CLASS@CosmoTools2018

Thomas Tram

Aarhus University

23/04/2018

Programme

Programme of the day

9.00-10.30	CMB physics and Boltzmann codes.
10.30-11.00	Coffee break.
11.00-12.30	CAMB.
12.30-14.00	Lunch break.
14.00-15.30	**CLASS**.
15.30-16.00	Coffee break.
16.00-18.00	Boltzmann code exercises.
18.00-20.00	Social event.

What is CLASS?

An Einstein-Boltzmann code!

- Solves the coupled **Einstein-Boltzmann** equations for many types of matter in the Universe to first order in perturbation theory.
- Computes **CMB observables** such as temperature and polarisation correlations C_ℓ^{TT}, C_ℓ^{TE}, C_ℓ^{EE}, C_ℓ^{BB}.
- Computes **LSS observables** such as the total matter power spectrum $P(k)$ and individual density and velocity transfer functions $\Delta_\delta^i(k,\tau)$, $\Delta_\theta^i(k,\tau)$.

Other Boltzmann codes

A history of Boltzmann codes

1995	COSMICS by Ed Bertschinger in Fortran77.
1996	Seljak&Zaldarriaga adds a few functions to COSMICS for implementing the line-of-sight formalism. The new code is much faster and is called CMBFAST.
'96–'99	CMBFAST improved with RECFAST and nonzero curvature.
1999	Antony Lewis translates CMBFAST into Fortran90, adds nonzero curvature and many other improvements. Released as CAMB.
2003	Similar restructuring and translation of CMBFAST done by Michael Doran in C++, called CMBEASY.
'03–'11	Only CAMB maintained.
2011	CLASS 1.0 released.

This lecture

Hands on, not a sales pitch!

- Focus on *using* CLASS through the Python interface.
- Performing very simple modifications.

Notable omissions

- CLASS from command line.
- Error management system.
- Numerical methods: `ndf15` (implicit, variable-order variable step-size), Modified Steed's method for (hyper-)spherical Bessel functions, Hermite interpolation, . . .
- No CAMB vs. CLASS comparison.

About CLASS

A few things I realised were missing 30 minutes ago

- No hard-coded constants: all technical precision parameters may be changed through the input.
- Different species in the Universe are named like `_b` (for baryons) in the code.
- 10 verbose parameters control output from 0–5, e.g. `background_verbose=1`. Default 0.
- When running in the *Jupyter notebook* , output will be written to the terminal from where you launched it.

The CLASSY module

classy, the CLASS wrapper

- All the functionality of **classy** is found in the **Python** class called **Class**.
- Import **Class** in In **Python** by:

```
from classy import Class
```

Running CLASS from Python

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

cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl','lensing':'
    yes','modes':'s,t','r':'0.2'})
cosmo.compute()
```

The Jupyter Notebook

Jupyter Notebook

Jupyter Notebook is a cell-based interface to IPython.

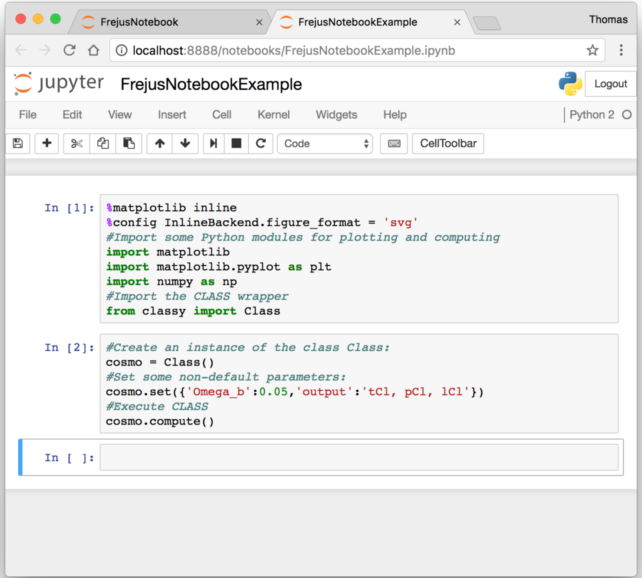
- Has Tab-completion of variables and function names.
- Nicely presents the documentation of each function.
- Easy way to get started on Python.

Launching Jupyter Notebook

Write the following command to launch the notebook:

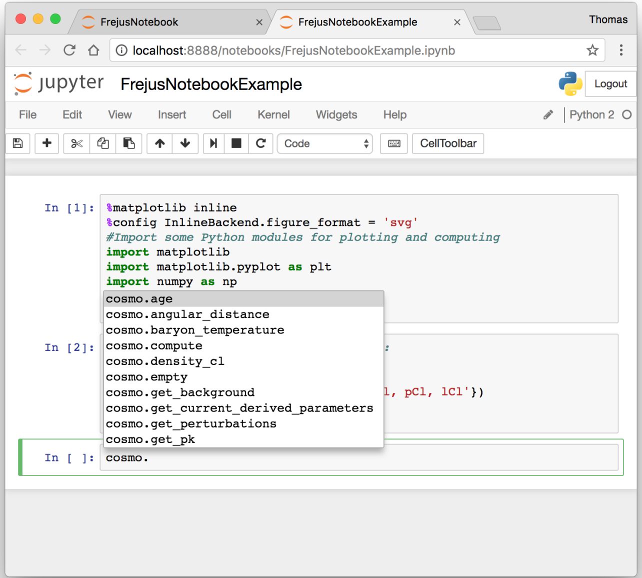
- `$> jupyter notebook`

The notebook



```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'svg'
        #Import some Python modules for plotting and computing
        import matplotlib
        import matplotlib.pyplot as plt
        import numpy as np
        #Import the CLASS wrapper
        from classy import Class

In [2]: #Create an instance of the class Class:
        cosmo = Class()
        #Set some non-default parameters:
        cosmo.set({'Omega_b':0.05,'output':'tCl, pCl, lCl'})
        #Execute CLASS
        cosmo.compute()

In [ ]:
```

Tab: Available class methods


FrejusNotebook
FrejusNotebookExample
Thomas
localhost:8888/notebooks/FrejusNotebookExample.ipynb
jupyter FrejusNotebookExample
Logout
File Edit View Insert Cell Kernel Widgets Help
Python 2
Code
CellToolbar

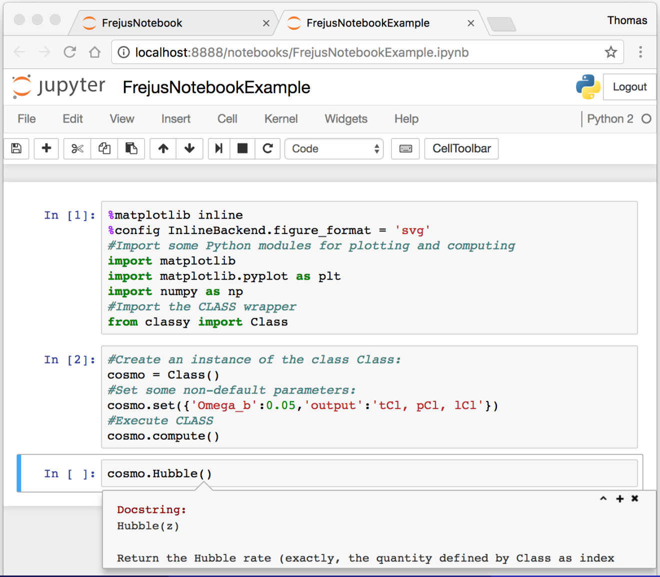
```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'svg'
        #Import some Python modules for plotting and computing
        import matplotlib
        import matplotlib.pyplot as plt
        import numpy as np
```

```
In [2]:                                                   :

                                                  l, pCl, lCl'})
```

```
cosmo.age
cosmo.angular_distance
cosmo.baryon_temperature
cosmo.compute
cosmo.density_cl
cosmo.empty
cosmo.get_background
cosmo.get_current_derived_parameters
cosmo.get_perturbations
cosmo.get_pk
```

```
In [ ]: cosmo.
```

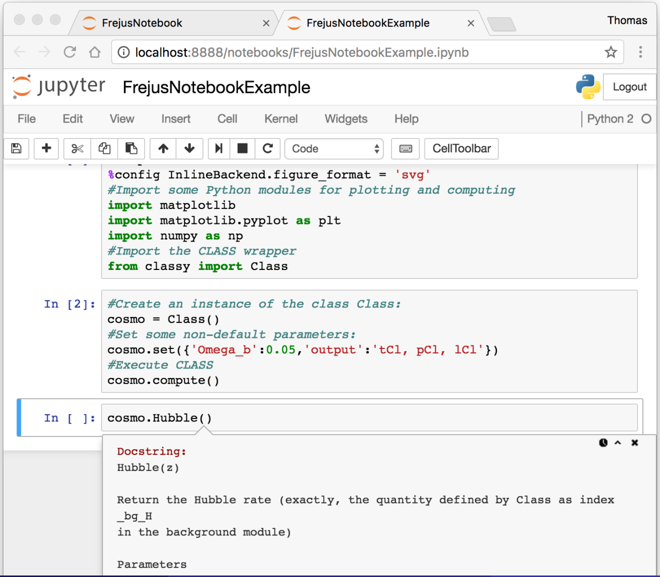
Shift+Tab: Help on method



```
In [1]: %matplotlib inline
        %config InlineBackend.figure_format = 'svg'
        #Import some Python modules for plotting and computing
        import matplotlib
        import matplotlib.pyplot as plt
        import numpy as np
        #Import the CLASS wrapper
        from classy import Class

In [2]: #Create an instance of the class Class:
        cosmo = Class()
        #Set some non-default parameters:
        cosmo.set({'Omega_b':0.05,'output':'tCl, pCl, lCl'})
        #Execute CLASS
        cosmo.compute()

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

```
Docstring:
Hubble(z)

Return the Hubble rate (exactly, the quantity defined by Class as index
```

Shift+Tab: More help


FrejusNotebook
FrejusNotebookExample
Thomas
localhost:8888/notebooks/FrejusNotebookExample.ipynb
jupyter FrejusNotebookExample
Logout
File Edit View Insert Cell Kernel Widgets Help
Python 2
Code
CellToolbar

```
%config InlineBackend.figure_format = 'svg'
#Import some Python modules for plotting and computing
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
#Import the CLASS wrapper
from classy import Class
```

```
In [2]: #Create an instance of the class Class:
        cosmo = Class()
        #Set some non-default parameters:
        cosmo.set({'Omega_b':0.05,'output':'tCl, pCl, lCl'})
        #Execute CLASS
        cosmo.compute()
```

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

```
Docstring:
Hubble(z)

Return the Hubble rate (exactly, the quantity defined by Class as index
_bg_H
in the background module)

Parameters
```

What can it do?

What is available in the wrapper?

- `get_background()` returns the information normally found in `_background.dat`.
- `get_thermodynamics()` returns the information of `_thermodynamics.dat`.
- `get_primordial()` corresponds to `_primordial_Pk.dat`.
- `get_perturbations()` returns everything found in `_perturbations*.dat`
- `get_transfer(z,format)` returns the density and velocity transfer functions at any redshift z. (Format can be either `'camb'` or `'class'`).

And even more...

What is available in the wrapper?

- `raw_cl()` returns unlensed C_ℓ.
- `lensed_cl()` returns lensed C_ℓ.
- `density_cl()` returns density C_ℓ.
- `pk(k, z)` returns the $P(k)$ at redshift z.
- Many other small functions.

Running a single instance of CLASS

Running CLASS

```
cosmo = Class()#Initiate one instance of the class Class
cosmo.set({'output':'tCl lCl mPk','Omega_cdm':0.25})
cosmo.compute()#Run CLASS
rawcl = cosmo.raw_cl()#Get C_l^XY from CLASS
print rawcl.keys()#Print keys in dictionary
```

Ressources

Useful links

- Exercises and slides for this course:
  `https://goo.gl/Feu7oi`
- GitHub repository for CLASS:
  `https://github.com/lesgourg/class_public`
- Help forum for CLASS:
  `https://github.com/lesgourg/class_public/issues`
- Official homepage, lecture notes and online documentation:
  `http://class-code.net/`
- Some Jupyter Notebooks using CLASS:
  `https://github.com/ThomasTram/iCLASS`
- Automatic online documentation (thanks to Deanna Hooper)
  `https://lesgourg.github.io/class_public/class_public-2.6.3/doc/manual/html/index.html`

The CLASS directory

Files and subdirectories

The directory `class/` contains subdirectories:

```
include/ # header files (*.h)
source/  # modules (*.c)
main/    # main CLASS function
tools/   # tools (*.c)
output/  # output files
python/  # python wrapper
cpp/     # C++ wrapper
```

plus examples of input files,`explanatory.ini`, `README.rst`, `Makefile`, and few other directories containing data files (`bbn/`) or external code (`hyrec/`)

CLASS modules

Structure of modules in CLASS

A module in CLASS is:

- a file `include/xxx.h` containing some declarations
- a file `source/xxx.c` containing some functions
- a structure `xx` of the information that must be propagated to other modules

Some fields in the structure are filled in the `input.c` module, and the rest are filled by a function `xxx_init(...)`.
"executing a module" $\equiv$ calling `xxx_init(...)`

All modules in CLASS

module	structure	ab.	*	main content
`input.c`	`precision`	`pr`	`ppr`	prec. params.
`background.c`	`background`	`ba`	`pba`	$a(\tau), \ldots$
`thermodynamics.c`	`thermodynamics`	`th`	`pth`	$x_e(z), \ldots$
`perturbations.c`	`perturbs`	`pt`	`ppt`	sources $S(k,t)$
`primordial.c`	`primordial`	`pm`	`ppm`	prim. spectra $\mathcal{P}(k)$
`nonlinear.c`	`nonlinear`	`nl`	`pnl`	NL corr. $\alpha_{\rm NL}(k,\tau)$
`transfer.c`	`transfers`	`tr`	`ptr`	trnsf. func. $\Delta_l(k)$
`spectra.c`	`spectra`	`sp`	`psp`	$P(k,z)$, C_ℓ's
`lensing.c`	`lensing`	`le`	`ple`	lensed C_ℓ's
`output.c`	`output`	`op`	`pop`	output format

The main CLASS function

```c
int main() {
 input_init_..(..,ppr,pba,pth,ppt,ptr,ppm,psp,pnl,ple,pop);
 background_init(ppr,pba);
 thermodynamics_init(ppr,pba,pth);
 perturb_init(ppr,pba,pth,ppt);
 primordial_init(ppr,ppt,ppm);
 nonlinear_init(ppr,pba,pth,ppt,ppm,pnl);
 transfer_init(ppr,pba,pth,ppt,pnl,ptr);
 spectra_init(ppr,pba,ppt,ppm,pnl,ptr,psp);
 lensing_init(ppr,ppt,psp,pnl,ple);
 output_init(pba,pth,ppt,ppm,ptr,psp,pnl,ple,pop);
 lensing_free(ple);
 spectra_free(psp);
 transfer_free(ptr);
 nonlinear_free(pnl);
 primordial_free(ppm);
 perturb_free(ppt);
 thermodynamics_free(pth);
 background_free(pba);
}
```

Background module

Unit system

We set $\hbar = c = k_B = 1$, and all dimensionful quantities have unit Mpc^n

Friedmann equation

$$a' = a^2 H = a^2 \sqrt{\sum_\alpha \rho_\alpha - \frac{K}{a^2}},$$

where we have defined $\rho_\alpha \equiv \frac{8\pi G}{3} \rho_\alpha^{\text{physical}}$.

Background module

background_functions()

Most quantities can be immediately inferred from a given value of a without integrating any differential equations:

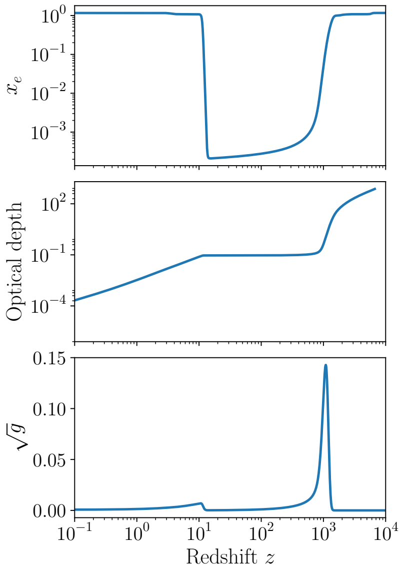
- $\rho_i = \Omega_i^0 H_0^2 \left(\frac{a}{a_0}\right)^{-3(1+w_i)}$
- $p_i = w_i \rho_i$
- $H = \left(\sum_i \rho_i - \frac{K}{a^2}\right)^{1/2}$
- $H' = \left(-\frac{3}{2}\sum_i(\rho_i + p_i) + \frac{K}{a^2}\right) a$
- $\rho_{\rm crit} = H^2$
- $\Omega_i = \rho_i / \rho_{\rm crit}$

Thermodynamics module

Free electron fraction

We must solve recombination and reionisation, to compute:

- the free electron fraction $x_e \equiv n_e/n_p$.
- the optical depth $\kappa(\tau)$ where $\kappa' = \sigma_T a n_p x_e$
- the visibility function $g(\tau) = \kappa' e^{-\kappa}$.



Thermodynamics module

Recombination

- Proper calculation requires out-of-equilibrium treatment of thousands of excited states in both hydrogen and helium.
- Fast computation by using effective 3-level atoms with fudged coefficients: `RECFAST` and `HyRec`.

Priomordial helium fraction Y_{He}

- Used to be a fixed input parameter for Boltzmann codes.
- BBN imposes a (non-analytic) relationship between N_{eff}, ω_b and Y_{He}.
- By default CLASS will find Y_{He} by interpolation in a table computed by the BBN code `Parthenope`.

Perturbations module

Einstein and Boltzmann equations

We must solve 2 of the 4 first order Einstein equations:

$$
\begin{aligned}
k^2\eta - \frac{1}{2}\frac{a'}{a}h' &= -4\pi G a^2 \delta\rho, \\
k^2\eta' &= 4\pi G a^2 (\rho + p)\theta, \\
h'' + 2\frac{a'}{a}h' - 2k^2\eta &= -24\pi G a^2 \delta p, \\
h'' + 6\eta'' + 2\frac{a'}{a}\left(h' + 6\eta'\right) - 2k^2\eta &= -24\pi G a^2 (\rho + p)\sigma
\end{aligned}
$$

together with the Boltzmann equation for each species present in the Universe.

Perturbations module

The Boltzmann equation

- At an abstract level we can write:

$$\mathcal{L}\left[f_\alpha(\tau, \mathbf{x}, \mathbf{p})\right] = \mathcal{C}\left[f_i, f_j\right](= 0). \qquad (1)$$

  The last equal sign is true for a **collisionless** species.

- We expand f_α to first order:

$$f_\alpha(\tau, \mathbf{x}, \mathbf{p}) \simeq f_0(q)(1 + \Psi(\tau, \mathbf{x}, q, \hat{n})). \qquad (2)$$

- Plugging equation (2) into equation (1) gives a Boltzmann equation for Ψ in Fourier space:

$$\frac{\partial \Psi}{\partial \tau} + i\frac{qk}{\epsilon}(\mathbf{k} \cdot \hat{n})\Psi + \frac{d \ln f_0}{d \ln q}\left[\dot{\eta} - \frac{\dot{h} + 6\dot{\eta}}{2}(\hat{k} \cdot \hat{n})^2\right] = \mathcal{C}$$

Perturbations module

A few missing definitions

We have defined the **comoving momentum** q and **comoving energy** ϵ by $q \equiv \frac{p}{T_\alpha}$ and $\epsilon \equiv \frac{\sqrt{p^2+m_\alpha^2}}{T_\alpha}$.

Perturbations module

A few missing definitions

We have defined the **comoving momentum** q and **comoving energy** ϵ by $q \equiv \frac{p}{T_\alpha}$ and $\epsilon \equiv \frac{\sqrt{p^2+m_\alpha^2}}{T_\alpha}$.

Why do we use Fourier-space?

- The Liouville operator $\mathcal{L}\left[f_i(\tau, \mathbf{x}, \mathbf{p})\right]$ contains the gradient operator ∂_j....
- ...so in real space, the Boltzmann equation is a Partial Differential Equation (PDE).
- But since $\partial_j e^{i\mathbf{k}\cdot\mathbf{x}} = ik_j e^{i\mathbf{k}\cdot\mathbf{x}}$, the PDE **decouples** and become a set of *ordinary* differential equations for each mode k.

Perturbations module

Legendre expansion of Ψ

Since $\hat{k} \cdot \hat{n} = \cos\theta$, the equation from before

$$\frac{\partial \Psi}{\partial \tau} + i\frac{qk}{\epsilon}(\mathbf{k} \cdot \hat{n})\Psi + \frac{d \ln f_0}{d \ln q}\left[\dot{\eta} - \frac{\dot{h} + 6\dot{\eta}}{2}(\hat{k} \cdot \hat{n})^2\right] = \mathcal{C}$$

has no dependence on the angle ϕ. Thus we can expand the angular dependence of Ψ in Legendre multipoles:

$$\Psi(\tau, k, q, \hat{k} \cdot \hat{n}) = \sum_{l}^{\infty}(2\ell + 1)\Psi_l(\tau, k, q)P_\ell(\hat{k} \cdot \hat{n}) \qquad (3)$$

Perturbations module

The Boltzmann hierarchy

$$\dot{\Psi}_0 = -\frac{qk}{\epsilon} + \frac{1}{6}\dot{h}\frac{d\ln f_0}{d\ln q} + \mathcal{C}_0,$$

$$\dot{\Psi}_1 = \frac{qk}{3\epsilon}(\Psi_0 - 2\Psi_2) + \mathcal{C}_1,$$

$$\dot{\Psi}_2 = \frac{qk}{5\epsilon}(2\Psi_1 - 3\Psi_3) - \left(\frac{1}{15}\dot{h} + \frac{2}{5}\dot{\eta}\right)\frac{d\ln f_0}{d\ln q} + \mathcal{C}_2,$$

$$\dot{\Psi}_\ell = \frac{qk}{(2\ell+1)\epsilon}\left(\ell\Psi_{\ell-1} - (\ell+1)\Psi_{\ell+1}\right), \quad \ell \leq 3.$$

Perturbations module

Simplifications for different species

- **Massive neutrinos** and **Dark Matter** usually have no interactions, so $\mathcal{C} = 0$.
- **Photons**, **massless neutrinos**, **baryons** and **Cold Dark Matter** have no momentum dependence, so we can integrate out q.
- **Baryons** and **Cold Dark Matter** have negligible shear, so we only need to consider $\ell = 0$ and $\ell = 1$.

Perturbations module

Simplifications for different species

- **Massive neutrinos** and **Dark Matter** usually have no interactions, so $\mathcal{C} = 0$.
- **Photons**, **massless neutrinos**, **baryons** and **Cold Dark Matter** have no momentum dependence, so we can integrate out q.
- **Baryons** and **Cold Dark Matter** have negligible shear, so we only need to consider $\ell = 0$ and $\ell = 1$.

Complications for photons

- **Photons** are not completely described by a single distribution function but the three Stoke's parameters Θ, Q and U. (Note that $\Theta \equiv \frac{\Delta T_\gamma}{T_\gamma} = \frac{1}{4}\delta_\gamma$)

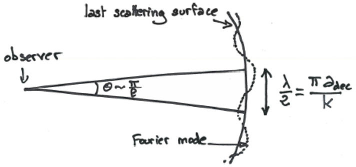
Perturbations module

The line-of-sight formalism

The Boltzmann equation has a formal solution in terms of an integral along the line-of-sight:

$$\Theta_l(\tau_0, k) = \int_{\tau_{\rm ini}}^{\tau_0} d\tau \; S_T(\tau, k) \; j_l(k(\tau_0 - \tau))$$

$$S_T(\tau, k) \equiv \underbrace{g\left(\Theta_0 + \psi\right)}_{\rm SW} + \underbrace{\left(g\, k^{-2}\theta_{\rm b}\right)'}_{\rm Doppler} + \underbrace{e^{-\kappa}(\phi' + \psi')}_{\rm ISW} + \text{pol.}\ .$$



last scattering surface
observer
$\theta \sim \frac{\pi}{\ell}$
$\frac{\lambda}{2} = \frac{\pi\, \partial_{\rm dec}}{k}$
Fourier mode

Perturbations module

Relevant sources

- sources for CMB **temperature** (decomposed in 1 to 3 terms)
- sources for CMB **polarisation** (only 1 term)
- **metric perturbations** ϕ and ψ and derivatives, used for lensing and galaxy number counts.
- **density** perturbations of all components $\{\delta_i\}$
- **velocity** perturbations of all components $\{\theta_i\}$

Transfer module

Purpose of the transfer module

The goal is to compute **harmonic transfer functions** by performing several integrals of the type

$$\Delta_l^X(q) = \int d\tau \;\; S_X(k(q), \tau) \;\; \phi_l^X(q, (\tau_0 - \tau))$$

for each mode, initial conditions, and several types of source functions. In flat space $k = q$.

Sparse ℓ-sampling

Calculation done for few values of ℓ (controlled by precision parameters). C_ℓ's are interpolated later.

Transfer module

Overview of `transfer_init`

- interpolate all sources along k. (k grid is finer in transfer module than in perturbation module.)
- compute all **flat Bessel functions**
- loop over q (or k in flat space).
- :   if non-flat, compute **hyperspherical Bessels** for this q
- :   loop over modes, initial conditions, types
- :   :   loop over ℓ
- :   :   :   integrate transfer function (or Limber approx.)

Primordial module

The primordial power spectrum

CLASS have a number of different ways to infer the primordial power spectrum of perturbations:

- Analytic parametrisation:

$$\mathcal{P}(k) = A \exp\left((n-1)\log(k/k_{\mathrm{pivot}}) + \alpha \log(k/k_{\mathrm{pivot}})^2\right).$$

- Taylor expansion of inflationary potential $V(\phi - \phi_*)$ or $H(\phi - \phi_*)$. (See astro-ph/0703625 and astro-ph/0710.1630 for details.)
- Parametrisation of $V(\phi)$ in the whole region between the observable part and the end of inflation.
- Primordial powerspectra from external code.

Spectra module

Linear matter power spectra

$$P(k,z) = \left(\delta_m(k,\tau(z))\right)^2 \mathcal{P}(k)$$

or in the case of several initial conditions,

$$P(k,z) = \sum_{ij} \delta_m^i(k,\tau(z))\, \delta_m^j(k,\tau(z)) \mathcal{P}_{ij}(k),$$

Angular power spectra

$$C_l^{XY} = 4\pi \sum_{ij} \int \frac{dk}{k} \Delta_l^X(k) \Delta_l^Y(k) \mathcal{P}(k)$$

(or its generalisation to several ICs). We have:

$$XY \in \left\{ \begin{array}{l} TT, TE, EE, BB, PP, TP, EP, \\ N_iN_j, TN_i, PN_i, L_iL_j, TL_i, N_iL_j \end{array} \right\}.$$

The input module

Main functions in the input module

- `input_init_from_arguments()` **or** `cosmo.set({})`.
  Read user input from `.ini`-file or from a Python dictionary.
- `input_read_parameters()`.
  Initialise all parameters with **default values** by caling `input_default_params()` and `input_default_precision()`, and then overwrites parameters that were passed as input.
- `input_init()`.
  Calls `input_read_parameters()` and runs a **shooting algorithm** if required.

Storing user input

The file_content structure

```
struct file_content fc;
fc.name[0] = "h";           fc.value[0] = "0.68";
fc.name[1] = "Omega_b";     fc.value[1] = "0.04";
fc.name[2] = "omega_cdm";   fc.value[2] = "0.12";
fc.name[3] = "modes";       fc.value[3] = "s,t";
...
```

Flexibility of input variables

Inside `input_read_parameters()`

Some simple analytical formulae, e.g. $\Omega_i = \omega_i/h^2$ allows for different input parametrisations.
Not always sufficient: E.g. $100 \times \theta_s$ cannot be converted analytically into h/H_0.

Non-trivial boundary conditions

target parameter	unknown parameter
$100 \times \theta_s$	h
Ω_{dcdm}	$\rho_{\mathrm{dcdm}}^{\mathrm{ini}}$
σ_8	A_s

Input module overview

Call sequence

Full sequence when CLASS is called with **input files** or from a generic **wrapper**:

- **Call** `input_init_from_argument(..)` **: read files and fill** `fc`
- **Call** `some_function(...)` **: Create and fill** `fc` **manually**
- Call `input_init(pfc,...)` :
- check if there are target/unknown parameters
- if yes, shooting: Run CLASS iteratively using "unknown parameters" until "target parameters" are reached
- all parameters are known at this point
- call `input_read_parameters(..)`

Specifying the Dark Energy sector

Λ, DE-fluid or scalar field

We have the closure relation

$$\sum_{\alpha \in \{\gamma, b, K, \ldots\}} \Omega_\alpha = 1 \,, \tag{4}$$

where we have included the contribution from curvature, Ω_K. This will be fullfilled using one of the following:

- **Scalar field**, if `Omega_Lambda=0`, `Omega_fld=0` and `Omega_scf<0`.
- **DE fluid**, if both `Omega_Lambda=0` and `Omega_fld` are *unspecified* .
- **Λ**, if `Omega_fld` is *unspecified* .

Expanding CLASS

Adding new physics

- Adding exotic physics to CLASS is **easy***.
- New features will be merged into the code and will be available in all future versions.
- This lecture will go through the implementation of decaying CDM, `dcdm`, step by step.

Footnote...

* If the rules of CLASS are carefully followed!

Example 1: Plot the CMB temperature power spectrum

Launch the Jupyter Notebook:

```
$> jupyter notebook
```

Run these lines in the notebook:

```
%matplotlib inline #Plots inside cells
%config InlineBackend.figure_format = 'svg'
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from classy import Class
cosmo = Class()
cosmo.set({'output':'tCl,pCl,lCl'})
cosmo.compute()
cl = cosmo.raw_cl()
l = cl['ell']
plt.loglog(l, l*(l+1)/(2.*np.pi)*cl['tt'])
```

Example 2: Baryon fraction as input parameter

Physics behind the example

The baryon density can be specified by either `Omega_b` or `omega_b`. Let us add the baryon fraction η_b as a third option:

$$\eta_b = \frac{n_N}{n_\gamma}. \tag{5}$$

The number density of photons are given by an integral over the Bose-Einstein distribution, so

$$n_\gamma = \frac{2\zeta(3)}{\pi^2}\left(\frac{k_B T_\gamma}{\hbar c}\right)^3, \tag{6}$$

while the density of nucleons is

$$n_N = \frac{\rho_b}{m_N} = \frac{3H_0^2\Omega_b}{8\pi G m_N} \simeq 1.346 \cdot 10^{-7}\mathrm{K}^3\Omega_b h^2. \tag{7}$$

Example 2: Baryon fraction as input parameter

Relation between Ω_b and η_b

Combining these equations give us

$$\Omega_b h^2 = 1.81 \cdot 10^6 \eta_b \left(\frac{T_{\gamma,0}}{K} \right)^3 \qquad (8)$$

Now open `input.c` and locate the function `input_read_parameters()`. Scroll down slowly until you find the point where the baryon density is read:

```
/* Omega_0_b (baryons) */
class_call(parser_read_double(pfc,"Omega_b",&param1,&flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_b",&param2,&flag2,errmsg),
           errmsg,
           errmsg);
class_test(((flag1 == _TRUE_) && (flag2 == _TRUE_)),
           errmsg,
           "In input file, you can only enter one of Omega_b or omega_b, choose one");
if (flag1 == _TRUE_)
  pba->Omega0_b = param1;
if (flag2 == _TRUE_)
  pba->Omega0_b = param2/pba->h/pba->h;
```

Example 2: Baryon fraction as input parameter

Locate the point where the baryon density is read.

```
/* Omega_0_b (baryons) */
class_call(parser_read_double(pfc,"Omega_b",&param1,&flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_b",&param2,&flag2,errmsg),
           errmsg,
           errmsg);
```

```
class_test(((flag1 == _TRUE_) && (flag2 == _TRUE_)),
           errmsg,
           "In input file, you can only enter one of Omega_b or omega_b, choose one");
```

```
if (flag1 == _TRUE_)
  pba->Omega0_b = param1;
if (flag2 == _TRUE_)
  pba->Omega0_b = param2/pba->h/pba->h;
```

Example 2: Baryon fraction as input parameter

Read the new input parameter eta_b:

```
/* Omega_0_b (baryons) */
class_call(parser_read_double(pfc,"Omega_b",&param1,&flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_b",&param2,&flag2,errmsg),
           errmsg,
           errmsg);
```

```
/* Read the baryon fraction eta_b! */
class_call(parser_read_double(pfc,"eta_b",&param3,&flag3,errmsg),
           errmsg,
           errmsg);
```

```
class_test(((flag1 == _TRUE_) && (flag2 == _TRUE_)),
           errmsg,
           "In input file, you can only enter one of Omega_b or omega_b, choose one");
```

```
if (flag1 == _TRUE_)
  pba->Omega0_b = param1;
if (flag2 == _TRUE_)
  pba->Omega0_b = param2/pba->h/pba->h;
```

Example 2: Baryon fraction as input parameter

Remove the old test and add a new one:

```
/* Omega_0_b (baryons) */
class_call(parser_read_double(pfc,"Omega_b",&param1,&flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_b",&param2,&flag2,errmsg),
           errmsg,
           errmsg);
```

```
/* Read the baryon fraction eta_b! */
class_call(parser_read_double(pfc,"eta_b",&param3,&flag3,errmsg),
           errmsg,
           errmsg);
```

```
/* Test that at most one flag has been set: */
class_test(class_at_least_two_of_three(flag1,flag2,flag3),
           errmsg,
           "In input file, you can only enter one of eta_b, Omega_b or omega_b, choose
                one");
```

```
if (flag1 == _TRUE_)
  pba->Omega0_b = param1;
if (flag2 == _TRUE_)
  pba->Omega0_b = param2/pba->h/pba->h;
```

Example 2: Baryon fraction as input parameter

Read the baryon fraction and convert to Ω_b using the formula:

```
/* Omega_0_b (baryons) */
class_call(parser_read_double(pfc,"Omega_b",&param1,&flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_b",&param2,&flag2,errmsg),
           errmsg,
           errmsg);
```

```
/* Read the baryon fraction eta_b! */
class_call(parser_read_double(pfc,"eta_b",&param3,&flag3,errmsg),
           errmsg,
           errmsg);
```

```
/* Test that at most one flag has been set: */
class_test(class_at_least_two_of_three(flag1,flag2,flag3),
           errmsg,
           "In input file, you can only enter one of eta_b, Omega_b or omega_b, choose
                one");
```

```
if (flag1 == _TRUE_)
  pba->Omega0_b = param1;
if (flag2 == _TRUE_)
  pba->Omega0_b = param2/pba->h/pba->h;
```

```
/* Set Omega_b in background structure. Formula hardcoded :( */
if (flag3 == _TRUE_)
  pba->Omega0_b = 1.81e6*param3*pow(pba->T_cmb,3)/pba->h/pba->h;
```

Implementation of the shooting method

Inside `input_init()` in `input.c`:

```
/** These two arrays must contain the strings of names to be searched
    for and the coresponding new parameter */
char * const target_namestrings[] = {"100*theta_s","Omega_dcdmdr","omega_dcdmdr",
                                     "Omega_scf","Omega_ini_dcdm","omega_ini_dcdm"};
char * const unknown_namestrings[] = {"h","Omega_ini_dcdm","Omega_ini_dcdm",
                                      "scf_shooting_parameter","Omega_dcdmdr","
                                           omega_dcdmdr"};
enum computation_stage target_cs[] = {cs_thermodynamics, cs_background, cs_background,
                                      cs_background, cs_background, cs_background};
```

And in the include file `input.h`:

```
enum target_names {theta_s, Omega_dcdmdr, omega_dcdmdr, Omega_scf, Omega_ini_dcdm,
     omega_ini_dcdm};
enum computation_stage {cs_background, cs_thermodynamics, cs_perturbations,
                        cs_primordial, cs_nonlinear, cs_transfer, cs_spectra};
#define _NUM_TARGETS_ 6 //Keep this number as number of target_names
```

Implementation of the shooting method

Inside `input_try_unknown_parameters()` in `input.c`:

```
for (i=0; i < pfzw->target_size; i++) {
  switch (pfzw->target_name[i]) {
  case theta_s:
    output[i] = 100.*th.rs_rec/th.ra_rec-pfzw->target_value[i];
    break;
```

Inside `input_get_guess()` in `input.c`:

```
/** Here we should right reasonable guesses for the unknown parameters.
    Also estimate dxdy, i.e. how the unknown parameter responds to the known.
    This can simply be estimated as the derivative of the guess formula.*/

for (index_guess=0; index_guess < pfzw->target_size; index_guess++) {
  switch (pfzw->target_name[index_guess]) {
  case theta_s:
    xguess[index_guess] = 3.54*pow(pfzw->target_value[index_guess],2)-5.455*pfzw->
         target_value[index_guess]+2.548;
    dxdy[index_guess] = (7.08*pfzw->target_value[index_guess]-5.455);
    /** Update pb to reflect guess */
    ba.h = xguess[index_guess];
    ba.H0 = ba.h *  1.e5 / _c_;
    break;
```

Installing CLASS and Jupyter Notebook

Obtaining the codes

- `$> git clone https://github.com/lesgourg/class_public class`
- If `git` is not installed: download from `class-code.net` or `github.com/lesgourg/class_public`.
- Download Anaconda Python 2.7: `https://store.continuum.io/cshop/anaconda/`

Installing CLASS

- Should be as easy as writing `make`, since CLASS does not require any external libraries.
- Note that the builtin C-compiler in OSX does not support OpenMP.
- CLASS is compatible with Python3, but not yet MontePython.

Installation details

Obtaining the codes

This PDF is available at https://goo.gl/Feu7oi as CLASS_overview.pdf.

```
$>git clone https://github.com/lesgourg/
    class_public class
$>cd class
$>which python #Is it Anaconda Python?
$>which gcc #Is it the correct compiler?
$>make clean; make -j
$>python -c 'from classy import Class'
```

Problems and solutions

- Not Anaconda Python: `$>source ANACONDA_FOLDER/bin/activate`
- `undefined symbol: _ZGVbN2v_sin` Follow the instructions at https://github.com/lesgourg/class_public/issues/99

The end?

Backup slides

- Adding a non-trivial species, or
- Spending some time on installation problems.

Expanding CLASS

Adding new physics

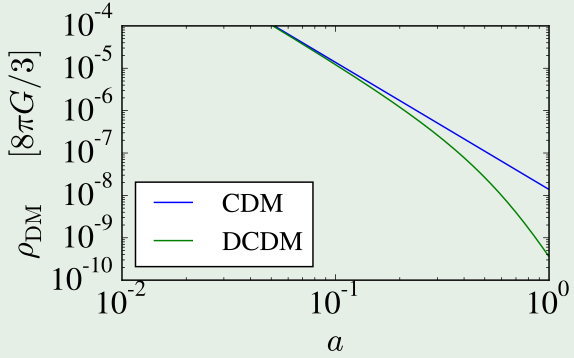
- Adding exotic physics to CLASS is **easy***.
- New features will be merged into the code and will be available in all future versions.
- This lecture will go through the implementation of decaying CDM, `dcdm`, step by step.

Footnote...

* If the rules of CLASS are carefully followed!

A bit about the physics

Background evolution



A bit about the physics

Background evolution

$$\rho_{\mathsf{dcdm}}{}' = -3\frac{a'}{a}\rho_{\mathsf{dcdm}} - a\,\Gamma_{\mathsf{dcdm}}\,\rho_{\mathsf{dcdm}}\ ,$$
$$\rho_{\mathsf{dr}}{}' = -4\frac{a'}{a}\rho_{\mathsf{dr}} + a\,\Gamma_{\mathsf{dcdm}}\,\rho_{\mathsf{dcdm}}\ .$$

Input parameters

`Omega_dcdmdr` $\equiv \Omega_{\rm dcdm} + \Omega_{\rm dr}$,

`omega_dcdmdr` $\equiv \omega_{\rm dcdm} + \omega_{\rm dr}$,

`Omega_ini_dcdm` $\equiv \Omega_{\rm dcdm,\ initial}$,

`omega_ini_dcdm` $\equiv \omega_{\rm dcdm,\ initial}$.

input.c modifications

input_init()

```
/** These two arrays must contain the strings of
    names to be searched
    for and the coresponding new parameter */
char * const target_namestrings[] = {"100*theta_s",
     "Omega_dcdmdr", "omega_dcdmdr", "Omega_scf", "
    Omega_ini_dcdm", "omega_ini_dcdm"};
char * const unknown_namestrings[] = {"h", "
    Omega_ini_dcdm", "Omega_ini_dcdm","
    scf_shooting_parameter", "Omega_dcdmdr", "
    omega_dcdmdr"};
enum computation_stage target_cs[] = {
    cs_thermodynamics, cs_background, cs_background
    , cs_background, cs_background, cs_background};
```

input.c modifications

input_init()

```
/** These two arrays must contain the strings of
    names to be searched
    for and the coresponding new parameter */
char * const target_namestrings[] = {"100*theta_s",
     "Omega_dcdmdr", "omega_dcdmdr", "Omega_scf", "
    Omega_ini_dcdm", "omega_ini_dcdm"};
char * const unknown_namestrings[] = {"h", "
    Omega_ini_dcdm", "Omega_ini_dcdm","
    scf_shooting_parameter", "Omega_dcdmdr", "
    omega_dcdmdr"};
enum computation_stage target_cs[] = {
    cs_thermodynamics, cs_background, cs_background
    , cs_background, cs_background, cs_background};
```

input_read_arguments()

I am skipping this part, since you have seen it many times already!

input.c modifications

input_try_unknown_parameters()

```
for (i=0; i < pfzw->target_size; i++) {
  switch (pfzw->target_name[i]) {
  case theta_s:
    output[i] = 100.*th.rs_rec/th.ra_rec-pfzw->
        target_value[i];
    break;
  case Omega_dcdmdr:
    rho_dcdm_today = ba.background_table[(ba.
        bt_size-1)*ba.bg_size+ba.index_bg_rho_dcdm
        ];
    if (ba.has_dr == _TRUE_)
      rho_dr_today = ba.background_table[(ba.
          bt_size-1)*ba.bg_size+ba.index_bg_rho_dr
          ];
    else
      rho_dr_today = 0.;
    output[i] = (rho_dcdm_today+rho_dr_today)/(ba.
        H0*ba.H0)-pfzw->target_value[i];
    break;
```

input.c modifications

input_get_guess()

```
/** Here we should right reasonable guesses for the unknown
    parameters. Also estimate dxdy, i.e. how the unknown
    parameter responds to the known. This can simply be
    estimated as the derivative of the guess formula.*/

for (index_guess=0; index_guess < pfzw->target_size;
    index_guess++) {
  switch (pfzw->target_name[index_guess]) {

  case Omega_dcdmdr:
    Omega_M = ba.Omega0_cdm+ba.Omega0_dcdmdr+ba.Omega0_b;

    if (gamma < 1)
      a_decay = 1.0;
    else
      a_decay = pow(1+(gamma*gamma-1.)/Omega_M,-1./3.);
    xguess[index_guess] = pfzw->target_value[index_guess]/
        a_decay;
    dxdy[index_guess] = 1./a_decay;
```

background.c modifications

background_functions()

```
/* dcdm */
if (pba->has_dcdm == _TRUE_) {
  /* Pass value of rho_dcdm to output */
  pvecback[pba->index_bg_rho_dcdm] = pvecback_B[pba->
      index_bi_rho_dcdm];
  rho_tot += pvecback[pba->index_bg_rho_dcdm];
  p_tot += 0.;
  rho_m += pvecback[pba->index_bg_rho_dcdm];
}

/* dr */
if (pba->has_dr == _TRUE_) {
  /* Pass value of rho_dr to output */
  pvecback[pba->index_bg_rho_dr] = pvecback_B[pba->
      index_bi_rho_dr];
  rho_tot += pvecback[pba->index_bg_rho_dr];
  p_tot += (1./3.)*pvecback[pba->index_bg_rho_dr];
  rho_r += pvecback[pba->index_bg_rho_dr];
}
```

background.c modifications

background_indices()

```
/* - index for dcdm */
class_define_index(pba->index_bg_rho_dcdm,pba->has_dcdm,index_bg
    ,1);

/* - index for dr */
class_define_index(pba->index_bg_rho_dr,pba->has_dr,index_bg,1);
```

```
/* - now, indices in vector of variables to integrate.
   First {B} variables, then {C} variables. */

index_bi=0;
```

```
/* -> energy density in DCDM */
class_define_index(pba->index_bi_rho_dcdm,pba->has_dcdm,index_bi
    ,1);

/* -> energy density in DR */
class_define_index(pba->index_bi_rho_dr,pba->has_dr,index_bi,1);
```

background.c modifications

background_initial_conditions()

```
/* Set initial values of {B} variables: */

if (pba->has_dcdm == _TRUE_){
  /* Remember that the critical density today in CLASS
      conventions is H0^2 */
  pvecback_integration[pba->index_bi_rho_dcdm] =
    pba->Omega_ini_dcdm*pba->H0*pba->H0*pow(pba->a_today/a,3);

if (pba->has_dr == _TRUE_){
  if (pba->has_dcdm == _TRUE_){

    f = 1./3.*pow(a/pba->a_today,6)*pvecback_integration[pba->
        index_bi_rho_dcdm]*pba->Gamma_dcdm/pow(pba->H0,3)/sqrt(
        Omega_rad);
    pvecback_integration[pba->index_bi_rho_dr] = f*pba->H0*pba
        ->H0/pow(a/pba->a_today,4);
  }
  else{
```

background.c modifications

background_output_titles()

```
class_store_columntitle(titles,"(.)rho_dcdm",pba->has_dcdm);
class_store_columntitle(titles,"(.)rho_dr",pba->has_dr);
```

background_output_data()

```
    class_store_double(dataptr, pvecback[pba->index_bg_rho_dcdm
        ], pba->has_dcdm, storeidx);
    class_store_double(dataptr, pvecback[pba->index_bg_rho_dr],
         pba->has_dr, storeidx);
```

background.c modifications

background_derivs()

```
if (pba->has_dcdm == _TRUE_){
  /** compute dcdm density rho' = -3aH rho - a Gamma rho*/
  dy[pba->index_bi_rho_dcdm] = -3.*y[pba->index_bi_a]*
      pvecback[pba->index_bg_H]*y[pba->index_bi_rho_dcdm]
      - y[pba->index_bi_a]*pba->Gamma_dcdm*y[pba->
      index_bi_rho_dcdm];
}

if ((pba->has_dcdm == _TRUE_) && (pba->has_dr == _TRUE_)){
  /** Compute dr density rho' = -4aH rho - a Gamma rho*/
  dy[pba->index_bi_rho_dr] = -4.*y[pba->index_bi_a]*
      pvecback[pba->index_bg_H]*y[pba->index_bi_rho_dr] +
      y[pba->index_bi_a]*pba->Gamma_dcdm*y[pba->
      index_bi_rho_dcdm];
}
```

perturbations.c modifications

perturb_indices_of_perturbs()

```
    if (ppt->has_density_transfers == _TRUE_) {

      if (pba->has_dcdm == _TRUE_)
        ppt->has_source_delta_dcdm = _TRUE_;

      if (pba->has_dr == _TRUE_)
        ppt->has_source_delta_dr = _TRUE_;

    if (ppt->has_velocity_transfers == _TRUE_) {

        ppt->has_source_theta_dcdm = _TRUE_;
      if (pba->has_fld == _TRUE_)

      if (pba->has_dr == _TRUE_)
        ppt->has_source_theta_dr = _TRUE_;
```

perturbations.c modifications

perturb_indices_of_perturbs()

```
class_define_index(ppt->index_tp_delta_dcdm, ppt->
    has_source_delta_dcdm,index_type,1);
```

```
class_define_index(ppt->index_tp_delta_dr,   ppt->
    has_source_delta_dr, index_type,1);
```

```
class_define_index(ppt->index_tp_theta_dcdm, ppt->
    has_source_theta_dcdm,index_type,1);
```

```
class_define_index(ppt->index_tp_theta_dr,   ppt->
    has_source_theta_dr,  index_type,1);
```

perturbations.c modifications

perturb_prepare_output()

```
    /* Decaying cold dark matter */
    class_store_columntitle(ppt->scalar_titles, "
        delta_dcdm", pba->has_dcdm);
    class_store_columntitle(ppt->scalar_titles, "
        theta_dcdm", pba->has_dcdm);
    /* Decay radiation */
    class_store_columntitle(ppt->scalar_titles, "delta_dr"
        , pba->has_dr);
    class_store_columntitle(ppt->scalar_titles, "theta_dr"
        , pba->has_dr);
    class_store_columntitle(ppt->scalar_titles, "shear_dr"
        , pba->has_dr);
```

perturbations.c modifications

perturb_vector_init()

```
/* dcdm */

class_define_index(ppv->index_pt_delta_dcdm,pba->
    has_dcdm,index_pt,1); /* dcdm density */
class_define_index(ppv->index_pt_theta_dcdm,pba->
    has_dcdm,index_pt,1); /* dcdm velocity */

/* ultra relativistic decay radiation */
if (pba->has_dr==_TRUE_){
  ppv->l_max_dr = ppr->l_max_dr;
  class_define_index(ppv->index_pt_F0_dr,_TRUE_,index_pt
      ,ppv->l_max_dr+1); /* all momenta in Boltzmann
      hierarchy  */
}
```

perturbations.c modifications

perturb_vector_init()

```
/** - case of switching approximation while a wavenumber
    is being integrated */

  if (_scalars_) {

    if (pba->has_dcdm == _TRUE_) {

      ppv->y[ppv->index_pt_delta_dcdm] =
        ppw->pv->y[ppw->pv->index_pt_delta_dcdm];

      ppv->y[ppv->index_pt_theta_dcdm] =
        ppw->pv->y[ppw->pv->index_pt_theta_dcdm];
    }

    if (pba->has_dr == _TRUE_){
      for (l=0; l <= ppv->l_max_dr; l++)
        ppv->y[ppv->index_pt_F0_dr+l] =
          ppw->pv->y[ppw->pv->index_pt_F0_dr+l];
    }
```

perturbations.c modifications

perturb_initial_conditions()

```
      if (pba->has_dcdm == _TRUE_) {
        ppw->pv->y[ppw->pv->index_pt_delta_dcdm] = 3./4.*ppw->pv->y
            [ppw->pv->index_pt_delta_g]; /* dcdm density */

    if (ppt->gauge == newtonian) {

      if (pba->has_dcdm == _TRUE_) {
        ppw->pv->y[ppw->pv->index_pt_delta_dcdm] += (-3.*
            a_prime_over_a - a*pba->Gamma_dcdm)*alpha;
        ppw->pv->y[ppw->pv->index_pt_theta_dcdm] = k*k*alpha;
      }

        if (pba->has_dr == _TRUE_)
          delta_dr += (-4.*a_prime_over_a + a*pba->Gamma_dcdm*ppw->
              pvecback[pba->index_bg_rho_dcdm]/ppw->pvecback[pba->
              index_bg_rho_dr])*alpha;

    } /* end of gauge transformation to newtonian gauge */
```

perturbations.c modifications

perturb_initial_conditions()

```
    if (pba->has_dr == _TRUE_) {

      f_dr = pow(pow(a/pba->a_today,2)/pba->H0,2)*ppw->
          pvecback[pba->index_bg_rho_dr];

      ppw->pv->y[ppw->pv->index_pt_F0_dr] = delta_dr*f_dr;

      ppw->pv->y[ppw->pv->index_pt_F0_dr+1] = 4./(3.*k)*
          theta_ur*f_dr;

      ppw->pv->y[ppw->pv->index_pt_F0_dr+2] = 2.*shear_ur*
          f_dr;

      ppw->pv->y[ppw->pv->index_pt_F0_dr+3] = l3_ur*f_dr;

    }
```

perturbations.c modifications

perturb_total_stress_energy()

```
    /* dcdm contribution */
    if (pba->has_dcdm == _TRUE_) {
      ppw->delta_rho += ppw->pvecback[pba->index_bg_rho_dcdm]*y
          [ppw->pv->index_pt_delta_dcdm];
      ppw->rho_plus_p_theta += ppw->pvecback[pba->
          index_bg_rho_dcdm]*y[ppw->pv->index_pt_theta_dcdm];
    }

    if (pba->has_dr == _TRUE_) {
      rho_dr_over_f = pow(pba->H0/a2,2);
      ppw->delta_rho += rho_dr_over_f*y[ppw->pv->index_pt_F0_dr
          ];
      ppw->rho_plus_p_theta += 4./3.*3./4*k*rho_dr_over_f*y[ppw
          ->pv->index_pt_F0_dr+1];
      ppw->rho_plus_p_shear += 2./3.*rho_dr_over_f*y[ppw->pv->
          index_pt_F0_dr+2];
      ppw->delta_p += 1./3.*rho_dr_over_f*y[ppw->pv->
          index_pt_F0_dr];
    }
```

perturbations.c modifications

perturb_sources()

```
    /* delta_dcdm */
    if (ppt->has_source_delta_dcdm == _TRUE_) {
      _set_source_(ppt->index_tp_delta_dcdm) = y[ppw->pv->
          index_pt_delta_dcdm];
    }

    /* delta_dr */
    if (ppt->has_source_delta_dr == _TRUE_) {
      f_dr = pow(a2_rel/pba->H0,2)*pvecback[pba->
          index_bg_rho_dr];
      _set_source_(ppt->index_tp_delta_dr) = y[ppw->pv->
          index_pt_F0_dr]/f_dr;
    }
```

perturbations.c modifications

perturb_sources()

```
    /* theta_dcdm */
    if (ppt->has_source_theta_dcdm == _TRUE_) {
      _set_source_(ppt->index_tp_theta_dcdm) = y[ppw->pv->
          index_pt_theta_dcdm];
    }
```

```
    /* theta_dr */
    if (ppt->has_source_theta_dr == _TRUE_) {
      f_dr = pow(a2_rel/pba->H0,2)*pvecback[pba->
          index_bg_rho_dr];
      _set_source_(ppt->index_tp_theta_dr) = 3./4.*k*y[ppw->pv
          ->index_pt_F0_dr+1]/f_dr;
    }
```

perturbations.c modifications

perturb_print_variables()

```
    if (pba->has_dcdm == _TRUE_) {
      delta_dcdm = y[ppw->pv->index_pt_delta_dcdm];
      theta_dcdm = y[ppw->pv->index_pt_theta_dcdm];
    }

    if (pba->has_dr == _TRUE_) {
      f_dr = pow(pvecback[pba->index_bg_a]*pvecback[pba->
          index_bg_a]/pba->H0,2)*pvecback[pba->index_bg_rho_dr
          ];
      delta_dr = y[ppw->pv->index_pt_F0_dr]/f_dr;
      theta_dr = y[ppw->pv->index_pt_F0_dr+1]*3./4.*k/f_dr;
      shear_dr = y[ppw->pv->index_pt_F0_dr+2]*0.5/f_dr;
    }
```

perturbations.c modifications

perturb_print_variables()

```
    /* converting synchronous variables to newtonian ones */
    if (ppt->gauge == synchronous) {

      if (pba->has_dr == _TRUE_) {
        delta_dr += (-4.*a*H+a*pba->Gamma_dcdm*pvecback[pba->
            index_bg_rho_dcdm]/pvecback[pba->index_bg_rho_dr])*
            alpha;
        theta_dr += k*k*alpha;
      }

      if (pba->has_dcdm == _TRUE_) {
        delta_dcdm += alpha*(-a*pba->Gamma_dcdm-3.*a*H);
        theta_dcdm += k*k*alpha;
      }

    }
```

perturbations.c modifications

perturb_print_variables()

```
    /* Decaying cold dark matter */
    class_store_double(dataptr, delta_dcdm, pba->has_dcdm,
        storeidx);
    class_store_double(dataptr, theta_dcdm, pba->has_dcdm,
        storeidx);
    /* Decay radiation */
    class_store_double(dataptr, delta_dr, pba->has_dr, storeidx
        );
    class_store_double(dataptr, theta_dr, pba->has_dr, storeidx
        );
    class_store_double(dataptr, shear_dr, pba->has_dr, storeidx
        );
```

perturbations.c modifications

perturb_derivs()

```
    /** -> dcdm and dr */
    if (pba->has_dcdm == _TRUE_) {
      dy[pv->index_pt_delta_dcdm] = -(y[pv->index_pt_theta_dcdm
          ]+metric_continuity) - a * pba->Gamma_dcdm / k2 *
          metric_euler;
      dy[pv->index_pt_theta_dcdm] = - a_prime_over_a*y[pv->
          index_pt_theta_dcdm] + metric_euler;
    }
    /** -> dr */
    if ((pba->has_dcdm == _TRUE_)&&(pba->has_dr == _TRUE_)) {
      f_dr = pow(pow(a/pba->a_today,2)/pba->H0,2)*pvecback[pba
          ->index_bg_rho_dr];
      fprime_dr = pba->Gamma_dcdm*pvecback[pba->
          index_bg_rho_dcdm]*pow(a,5)/pow(pba->H0,2);
      dy[pv->index_pt_F0_dr] = -k*y[pv->index_pt_F0_dr
          +1]-4./3.*metric_continuity*f_dr+fprime_dr*(y[pv->
          index_pt_delta_dcdm]+metric_euler/k2);
```

spectra.c modifications

spectra_indices

```
  class_define_index(psp->index_tr_delta_dcdm,ppt->
      has_source_delta_dcdm,index_tr,1);

  class_define_index(psp->index_tr_delta_dr,ppt->
      has_source_delta_dr,index_tr,1);

  class_define_index(psp->index_tr_theta_dcdm,ppt->
      has_source_theta_dcdm,index_tr,1);

  class_define_index(psp->index_tr_theta_dr,ppt->
      has_source_theta_ur,index_tr,1);
```

spectra.c modifications

spectra_output_tk_titles()

```
      class_store_columntitle(titles,"d_dcdm",pba->has_dcdm);
      class_store_columntitle(titles,"d_dr",pba->has_dr);
```

```
      class_store_columntitle(titles,"t_dcdm",pba->has_dcdm);
      class_store_columntitle(titles,"t_dr",pba->has_dr);
```

spectra.c modifications

spectra_output_tk_data()

```
            class_store_double(dataptr,tk[psp->
                index_tr_delta_dcdm],ppt->has_source_delta_dcdm
                ,storeidx);
            class_store_double(dataptr,tk[psp->
                index_tr_delta_dr],ppt->has_source_delta_dr,
                storeidx);
```

```
            class_store_double(dataptr,tk[psp->
                index_tr_theta_dcdm],ppt->has_source_theta_dcdm
                ,storeidx);
            class_store_double(dataptr,tk[psp->
                index_tr_theta_dr],ppt->has_source_theta_dr,
                storeidx);
```