

# Lecture II: The Background

Julien Lesgourgues

EPFL & CERN

Geneva, 01.04.2014

# Homogeneous cosmology

- treated by the module `background.c`. So this lecture will refer mainly to the content of `input/background.h`, `source/background.c`, and to the structure referred as `ba`:

```
struct background ba;
```

with fields `ba.blabla`, or through the pointer `pba`:

```
struct background * pba;
```

with fields `pba->blabla`.

# Homogeneous cosmology

- treated by the module `background.c`. So this lecture will refer mainly to the content of `input/background.h`, `source/background.c`, and to the structure referred as `ba`:

```
struct background ba;
```

with fields `ba.blabla`, or through the pointer `pba`:

```
struct background * pba;
```

with fields `pba->blabla`.

- the goal of this module is to solve the background evolution and store the results in a table. It should provide a function able to interpolate within this table at any value of time.

# Homogeneous cosmology

- treated by the module `background.c`. So this lecture will refer mainly to the content of `input/background.h`, `source/background.c`, and to the structure referred as `ba`:

```
struct background ba;
```

with fields `ba.blabla`, or through the pointer `pba`:

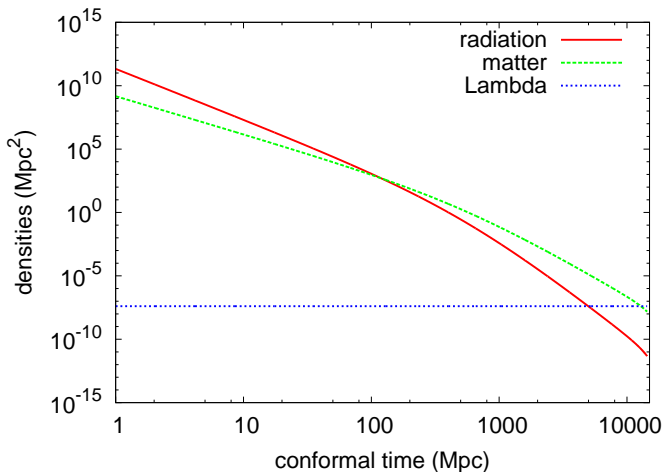
```
struct background * pba;
```

with fields `pba->blabla`.

- the goal of this module is to **solve the background evolution and store the results in a table**. It should provide a function able to interpolate within this table at any value of time.
- other modules should be able to know all background quantities (densities, pressures, Hubble rate, angular/luminosity distances, etc.) at any given time or redshift.

# Homogeneous cosmology

- at the end of this lecture, we also see how to plot the background evolution.



Units assume  $c = 1$  and all quantities in  $\text{Mpc}^n$

- times and distances are in Mpc: conformal time  $\tau$  in Mpc,  $H = \frac{a'}{a^2}$  in  $\text{Mpc}^{-1}$ .

Units assume  $c = 1$  and all quantities in  $\text{Mpc}^n$

- times and distances are in Mpc: conformal time  $\tau$  in Mpc,  $H = \frac{a'}{a^2}$  in  $\text{Mpc}^{-1}$ .
- all densities and pressures appearing in the code are in fact some rescaled variables:

$$\rho = \frac{8\pi G}{3}\rho_{\text{physical}}, \quad p = \frac{8\pi G}{3}p_{\text{physical}}, \quad \text{in } \text{Mpc}^{-2}.$$

Units assume  $c = 1$  and all quantities in **Mpc<sup>n</sup>**

- times and distances are in Mpc: conformal time  $\tau$  in Mpc,  $H = \frac{a'}{a^2}$  in  $\text{Mpc}^{-1}$ .
- all densities and pressures appearing in the code are in fact some rescaled variables:

$$\rho = \frac{8\pi G}{3}\rho_{\text{physical}}, \quad p = \frac{8\pi G}{3}p_{\text{physical}}, \quad \text{in } \text{Mpc}^{-2}.$$

So the **Friedmann equation** reads

$$H = \left( \sum_i \rho_i - \frac{K}{a^2} \right)^{1/2}$$

with the curvature  $K$  also in  $\text{Mpc}^{-2}$ .



# The function `background_functions()`

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

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

These quantities are all returned by a function `background_functions(pba,a,...)`

# What shall we integrate?

But to get them as a function of time we need to **integrate** one differential equation:

$$a' = a^2 H$$

Then we know  $a(\tau)$ , and hence all previous quantities as a function of  $\tau$ .

# What shall we integrate?

But to get them as a function of time we need to **integrate** one differential equation:

$$a' = a^2 H$$

Then we know  $a(\tau)$ , and hence all previous quantities as a function of  $\tau$ .

Other quantities requiring an **integration over time**:

- proper time:  $t' = a$  (since  $d\tau = dt/a$ )

# What shall we integrate?

But to get them as a function of time we need to **integrate** one differential equation:

$$a' = a^2 H$$

Then we know  $a(\tau)$ , and hence all previous quantities as a function of  $\tau$ .

Other quantities requiring an **integration over time**:

- proper time:  $t' = a$  (since  $d\tau = dt/a$ )
- comoving sound horizon:  $r'_s = c_s$ , since  $r_s = \int_{\tau_{\text{ini}}}^{\tau_0} c_s d\tau$ , with a squared sound speed in the photon+baryon+electron fluid

$$c_s^2 = \frac{\delta p_\gamma}{\delta \rho_\gamma + \delta \rho_b} = \frac{1}{3(1 + [3\rho_b/4\rho_\gamma])}.$$

# What shall we integrate?

But to get them as a function of time we need to **integrate** one differential equation:

$$a' = a^2 H$$

Then we know  $a(\tau)$ , and hence all previous quantities as a function of  $\tau$ .

Other quantities requiring an **integration over time**:

- proper time:  $t' = a$  (since  $d\tau = dt/a$ )
- comoving sound horizon:  $r'_s = c_s$ , since  $r_s = \int_{\tau_{\text{ini}}}^{\tau_0} c_s d\tau$ , with a squared sound speed in the photon+baryon+electron fluid

$$c_s^2 = \frac{\delta p_\gamma}{\delta \rho_\gamma + \delta \rho_b} = \frac{1}{3(1 + [3\rho_b/4\rho_\gamma])}.$$

- linear growth factor of density perturbations in minimal  $\Lambda$ CDM model (filled with dust),  $D' = 1/(aH^2)$  (such that  $\delta_m \propto D$ ).

# How can we classify background variables?

In general, three types of parameters:

- $\{A\} = \{\rho_i, p_i, H, \dots\}$  which can be expressed directly as a function of some variables  $\{B\}$ .

# How can we classify background variables?

In general, three types of parameters:

- $\{A\} = \{\rho_i, p_i, H, \dots\}$  which can be expressed directly as a function of some variables  $\{B\}$ .
- $\{B\} = \{a\}$ , which need to be integrated over time.

# How can we classify background variables?

In general, three types of parameters:

- $\{A\} = \{\rho_i, p_i, H, \dots\}$  which can be expressed directly as a function of some variables  $\{B\}$ .
- $\{B\} = \{a\}$ , which need to be integrated over time.
- $\{C\} = \{t, r_s, D\}$ , which also need to be integrated over time but are not used to compute  $\{A\}$ .



# How can we classify background variables?

In general, three types of parameters:

- $\{A\} = \{\rho_i, p_i, H, \dots\}$  which can be expressed directly as a function of some variables  $\{B\}$ .
- $\{B\} = \{a\}$ , which need to be integrated over time.
- $\{C\} = \{t, r_s, D\}$ , which also need to be integrated over time but are not used to compute  $\{A\}$ .

Exemple of extended cosmology with quintessence  $\phi$ : in this case there are some new parameters in the categories  $\{A\}$ ,  $\{B\}$ :

- $\{A\} = \{\rho_i, p_i, H, \dots, V_\phi, \rho_\phi, p_\phi\}$
- $\{B\} = \{a, \phi, \phi'\}$

# How can we classify background variables?

In general, three types of parameters:

- $\{A\} = \{\rho_i, p_i, H, \dots\}$  which can be expressed directly as a function of some variables  $\{B\}$ .
- $\{B\} = \{a\}$ , which need to be integrated over time.
- $\{C\} = \{t, r_s, D\}$ , which also need to be integrated over time but are not used to compute  $\{A\}$ .

Example of extended cosmology with quintessence  $\phi$ : in this case there are some new parameters in the categories  $\{A\}$ ,  $\{B\}$ :

- $\{A\} = \{\rho_i, p_i, H, \dots, V_\phi, \rho_\phi, p_\phi\}$
- $\{B\} = \{a, \phi, \phi'\}$

Example of decaying dark matter with non-trivial differential equation giving  $\rho_{dm}(t)$ : also in this case there are some new parameters in the category  $\{B\}$ :

- $\{A\} = \{\rho_i, p_i, H, \dots\}$
- $\{B\} = \{a, \rho_{dm}\}$

See dedicated lecture by Thomas!

# Input background parameters

In the \*.ini file, the user may pass:

- Hubble: `h` or `H0`
- Photons: `T_cmb` or `Omega_g` or `omega_g`
- Ultra-relativistic relics: `N_eff` or `Omega_ur` or `omega_ur`
- CDM: `Omega_cdm` or `omega_cdm`
- Non-cold DM: `ncdm` : lots of possible input, see dedicated lectures
- Curvature: `Omega_k`
- Cosmological constant: `Omega_Lambda`
- Fluid: `Omega_fld`, `w0_fld`, `wa_fld`, `cs2_fld` (assuming CLP:  
 $w = w_0 + w_a(1 - a/a_0)$  and  $\delta p = c_s^2 \delta \rho$ )

# Input background parameters

In the \*.ini file, the user may pass:

- Hubble: `h` or `H0`
- Photons: `T_cmb` or `Omega_g` or `omega_g`
- Ultra-relativistic relics: `N_eff` or `Omega_ur` or `omega_ur`
- CDM: `Omega_cdm` or `omega_cdm`
- Non-cold DM: `ncdm` : lots of possible input, see dedicated lectures
- Curvature: `Omega_k`
- Cosmological constant: `Omega_Lambda`
- Fluid: `Omega_fld`, `w0_fld`, `wa_fld`, `cs2_fld` (assuming CLP:  
 $w = w_0 + w_a(1 - a/a_0)$  and  $\delta p = c_s^2 \delta \rho$ )

Just one convention to remember !!!!

One of `Omega_Lambda` or `Omega_fld` **must be left unspecified**, to let the code match with  $H_0$ :

$$H_0^2 = \sum_i \rho_i^0 - K/a_0^2$$

If the two are passed, there is an error message.

All details on the syntax and on these rules are explicitly written in the comments of `explanatory.ini`

# Remark on the component called "fluid"

Remark on the **fluid**:

- $\rho_i = \Omega_i^0 H_0^2 \left( \frac{a}{a_0} \right)^{-3(1+w_i)}$  is only valid when  $w_i = \text{constant}$ .
- for  $w = w_0 + w_a(1 - a/a_0)$ , an analytic integration of the energy conservation equation gives

$$\rho_i = \Omega_i^0 H_0^2 \left( \frac{a}{a_0} \right)^{-3(1+w_0+w_a)} e^{3w_a(a/a_0-1)}$$

# Remark on the component called "fluid"

Remark on the **fluid**:

- $\rho_i = \Omega_i^0 H_0^2 \left( \frac{a}{a_0} \right)^{-3(1+w_i)}$  is only valid when  $w_i = \text{constant}$ .
- for  $w = w_0 + w_a(1 - a/a_0)$ , an analytic integration of the energy conservation equation gives

$$\rho_i = \Omega_i^0 H_0^2 \left( \frac{a}{a_0} \right)^{-3(1+w_0+w_a)} e^{3w_a(a/a_0-1)}$$

Finally there is a parameter `background_verbose=...` (one verbose parameter for each module). 0 gives no output at all, 1 the standard output that you see by default, etc.

# External function in the background module

- `background_init()` solves the differential equations once and for all, and stores in the background structure some tabulated values  $\tau_i$ ,  $z_i$ , and all quantities  $\{A_i\}$ ,  $\{B_i\}$ ,  $\{C_i\}$ .

# External function in the background module

- `background_init()` solves the differential equations once and for all, and stores in the `background` structure some tabulated values  $\tau_i$ ,  $z_i$ , and all quantities  $\{A_i\}$ ,  $\{B_i\}$ ,  $\{C_i\}$ .
- `background_free()` frees the memory allocated for this table.



# External function in the background module

- `background_init()` solves the differential equations once and for all, and stores in the background structure some tabulated values  $\tau_i$ ,  $z_i$ , and all quantities  $\{A_i\}$ ,  $\{B_i\}$ ,  $\{C_i\}$ .
- `background_free()` frees the memory allocated for this table.
- `background_at_tau(pba, tau, ...)` interpolates inside this table and returns  $\{A(\tau)\}$ ,  $\{B(\tau)\}$ ,  $\{C(\tau)\}$ .

# External function in the background module

- `background_init()` solves the differential equations once and for all, and stores in the background structure some tabulated values  $\tau_i$ ,  $z_i$ , and all quantities  $\{A_i\}$ ,  $\{B_i\}$ ,  $\{C_i\}$ .
- `background_free()` frees the memory allocated for this table.
- `background_at_tau(pba, tau, ...)` interpolates inside this table and returns  $\{A(\tau)\}$ ,  $\{B(\tau)\}$ ,  $\{C(\tau)\}$ .
- `background_tau_of_z(pba, z, &tau)` returns  $\tau(z)$ , can be useful just before calling `background_at_tau(pba, tau, ...)`

# Internal function in the background module

- `background_indices(...)` allocates dynamically all indices (see next slides)
- `background_functions(pba, a,...)` gets  $\{B\}$  and returns  $\{A\}$ .
- `background_solve(...)` integrates the differential system for  $\{B\}$ ,  $\{C\}$  variables
- `background_initial_conditions(...)` assigns initial conditions  $\{B_{\text{ini}}\}$ ,  $\{C_{\text{ini}}\}$  to this system, e.g.  $a(\tau_{\text{ini}})$  assuming pure radiation domination since Big Bang
- `background_derivs(z,...)` contains the differential eqns.  $dy[i] = f(y[j])$

# Internal function in the background module

- `background_indices(...)` allocates dynamically all indices (see next slides)
- `background_functions(pba, a,...)` gets  $\{B\}$  and returns  $\{A\}$ .
- `background_solve(...)` integrates the differential system for  $\{B\}$ ,  $\{C\}$  variables
- `background_initial_conditions(...)` assigns initial conditions  $\{B_{\text{ini}}\}$ ,  $\{C_{\text{ini}}\}$  to this system, e.g.  $a(\tau_{\text{ini}})$  assuming pure radiation domination since Big Bang
- `background_derivs(z,...)` contains the differential eqns.  $dy[i] = f(y[j])$

These functions call auxiliary functions in `tools/`:

- for array interpolation, `tools/array.c` (contains functions with all operations on arrays: interpolation, extrapolation, integration, derivation, smoothing, ...)
- for the integration of Ordinary Differential Equations, `generic_integrator()`, which can be set either to `rkck` or `ndf15` (see dedicated lecture). We will find this feature anytime we need to integrate ODE (also in thermodynamics and in perturbations).

# Dynamical indexing

- Indexing is very generic in CLASS, **same rules apply everywhere.**

# Dynamical indexing

- Indexing is very generic in CLASS, same rules apply everywhere.
- Here, we want to define the indices of a vector of background quantities (stored in the background table).

# Dynamical indexing

- Indexing is very generic in CLASS, same rules apply everywhere.
- Here, we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abbreviation of 2 letters for these indices, `_bg_`.

# Dynamical indexing

- Indexing is very generic in CLASS, same rules apply everywhere.
- Here, we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abbreviation of 2 letters for these indices, `_bg_`.
- Then we declare all possible indices `index_bg_<blabla>` in `common/background.h` (more precisely, inside the structure `background`, because these indices are necessary for manipulating the background table).



# Dynamical indexing

- Indexing is very generic in CLASS, **same rules apply everywhere.**
- Here, we want to define the **indices of a vector of background quantities** (stored in the background table).
- We choose an abbreviation of 2 letters for these indices, `_bg_`.
- Then we declare all possible indices `index_bg_<blabla>` in `common/background.h` (more precisely, inside the structure `background`, because these indices are necessary for manipulating the background table).
- We also declare **flags** saying whether these indices need to be defined or not.

# Dynamical indexing

In `include/background.h`:

```
struct background {  
    /** input parameters with assigned in the input module*  
    */  
    double Omega0_cdm;  
    ...  
    /** flags and indices */  
    int has_cdm;      // can take values _TRUE_ or _FALSE_  
    ....  
  
    int index_bg_rho_cdm;  
    ...  
  
    int bg_size;  
  
    /** interpolation table */  
    double * background_table;  
}
```

# Dynamical indexing

In source/background.c, the function `background_indices()` called at the beginning of `background_init()` assigns numerical value to indices, that the user will never need to know (quantities always written symbolically as `y[pba->index_bg_rho_cdm]`)

```
int background_indices(pba,...) {
    /* initialize all flags */
    if (pba->Omega0_cdm != 0.)
        pba->has_cdm = _TRUE_;
    ...
    /* initialize all indices */
    index_bg=0;
    class_define_index(pba->index_bg_rho_cdm,
                      pba->has_cdm,
                      index_bg,
                      1);
    class_define_index(pba->index_bg_rho_fld,
                      pba->has_fld,
                      index_bg,
                      1);
    ...
    pba->bg_size = index_bg;
}
```

# Getting background quantities from outside

Calling background quantities from another module is then very simple: e.g., in the perturbation module:

```
double * pvecback;
class_alloc(pvecback,
            pba->bg_size*sizeof(double),
            ppt->error_message);
...
class_call(background_at_tau(pba,tau,...,...,pvecback),
            pba->error_message,
            ppt->error_message);

/* We want here to compute the total background density*/
if (pba->has_cdm == _TRUE_) {
    rho_tot += pvecback[pba->index_bg_rho_cdm];
}
if (pba->has_fld == _TRUE_) {
    rho_tot += pvecback[pba->index_bg_rho_fld];
    p_tot += pvecback[pba->index_bg_p_fld];
}
...
```

# Getting background quantities from outside

To save a bit of time, we have coded three return formats:

```
class_alloc(pvecback, pba->bg_size_short*sizeof(double),...);  
...  
class_call(background_at_tau(pba, tau, ..., ...,  
                             pba->short_info, pvecback),  
            ...,...)
```

or

```
class_alloc(pvecback, pba->bg_size_normal*sizeof(double),...)  
    ;  
...  
class_call(background_at_tau(pba, tau, ..., ...,  
                             pba->normal_info, pvecback),  
            ...,...)
```

or

```
class_alloc(pvecback, pba->bg_size*sizeof(double),...);  
...  
class_call(background_at_tau(pba, tau, ..., ...,  
                             pba->long_info, pvecback),  
            ...,...)
```

# Full list of coded background quantities

Currently, the list of all available background quantities is:

short_info	index_bg_a	$a$
"	index_bg_H	$H$
"	index_bg_H_prime	$H'$
normal_info	index_bg_rho_<i>	$\rho$ for _b, _g, _cdm, _ur, _fld, _lambda
"	variables for ncdm	see dedicated lecture
"	index_bg_Omega_r	$\Omega_{\text{radiation}}$
long_info	index_bg_rho_crit	$\rho_{\text{crit}}$
"	index_bg_Omega_m	$\Omega_{\text{matter}}$
"	index_bg_conf_distance	$\tau_0 - \tau = \chi$
"	index_bg_ang_distance	$d_A = a r$
"	index_bg_lum_distance	$d_L = (1 + z)^2 d_A$
"	index_bg_time	proper time $t$
"	index_bg_rs	conformal sound horizon $r_s$
"	index_bg_D	density growth factor of $\Lambda$ CDM
"	index_bg_f	velocity growth factor of $\Lambda$ CDM

(with  $r = \chi$ , or  $\sin(\sqrt{K}\chi)/\sqrt{K}$ , or  $\sinh(\sqrt{-K}\chi)/\sqrt{-K}$ )

# To conclude on dynamical indexing in CLASS

These **rules for flags and indices** are followed everywhere in the code, with maybe 50 different lists of indices `index_ab_blabla`. In the background module, two such lists:

- for all variables in the table ( $\{A\}$ ,  $\{B\}$ ,  $\{C\}$ ),

```
int index_bg_a;  
...  
int bg_size;
```

declared in `include/background.h` inside the background structure, to be used in other modules.

# To conclude on dynamical indexing in CLASS

These **rules for flags and indices** are followed everywhere in the code, with maybe 50 different lists of indices `index_ab_blabla`. In the background module, two such lists:

- for all variables in the table ( $\{A\}$ ,  $\{B\}$ ,  $\{C\}$ ),

```
int index_bg_a;  
...  
int bg_size;
```

declared in `include/background.h` inside the background structure, to be used in other modules.

- for the ODE  $dy[i] = f(y[j])$ , i.e. for variables  $\{B\}$ ,  $\{C\}$ ,

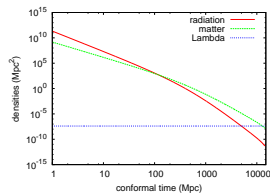
```
int index_bi_a;  
int index_bi_time;  
int index_bi_rs;  
int index_bi_growth;  
int bi_size;
```

declared in `include/background.h` outside the background structure, and erased/forgotten after the execution of `background_init()`.



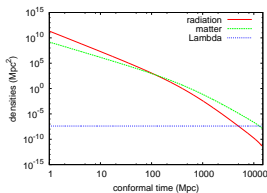
# Printing the background evolution

Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



# Printing the background evolution

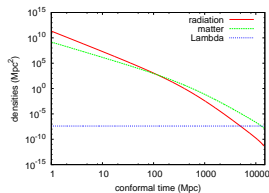
Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



1. **easiest:** execute e.g. `./class myinput.ini` including in the input file:  
write background = yes  
root = output/toto\_

# Printing the background evolution

Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



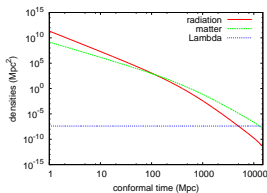
1. **easiest:** execute e.g. `./class myinput.ini` including in the input file:  
    `write background = yes`  
    `root = output/toto_`

The output module will also write a file `output/toto_background.dat` with header:

```
# Table of selected background quantities  
# All densities are multiplied by (8piG/3)  
# z, proper time [Gyr], conformal time * c [Mpc], H/c [1/Mpc] (etc.)
```

# Printing the background evolution

Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



1. **easiest:** execute e.g. `./class myinput.ini` including in the input file:  
    `write background = yes`  
    `root = output/toto_`

The output module will also write a file `output/toto_background.dat` with header:

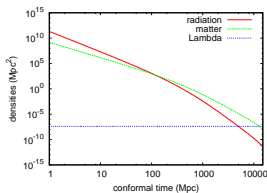
```
# Table of selected background quantities
# All densities are multiplied by (8piG/3)
# z, proper time [Gyr], conformal time * c [Mpc], H/c [1/Mpc] (etc.)
```

Output easy to customise in `output.c`, by editing:

`output_one_line_of_background(...)` for the quantities to plot in each line  
`output_open_background_file(...)` for the header (description of columns)

# Printing the background evolution

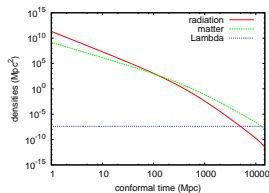
Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



2. good to know: `directory test/` contains several test codes executing only part of the `main(...)` function.

# Printing the background evolution

Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



2. good to know: directory `test/` contains several test codes executing only part of the `main(...)` function.

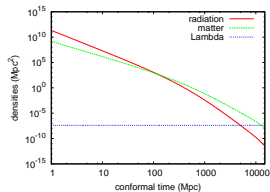
For instance: `test/test_background.c` only executes `input_init(...)`, `background_init(...)`, and then outputs a table of all background quantities. Useful also for quick debugging!

Usage:

```
> make test_background
> ./test_background myinput.ini
```

# Printing the background evolution

Two ways to produce such a plot with the quantities of your choice (maybe customised to your own model):



2. good to know: directory `test/` contains several test codes executing only part of the `main(...)` function.

For instance: `test/test_background.c` only executes `input_init(...)`, `background_init(...)`, and then outputs a table of all background quantities. Useful also for quick debugging!

Usage:

```
> make test_background
> ./test_background myinput.ini
```

Same with `test_thermodynamics.c`, `test_perturbations.c`, `test_nonlinear.c`, `test_transfer.c`...

# Background exercises (see exercise sheet for more details)

## Exercise IIa

Reproduce this plot from the Dodelson book *Modern Cosmology*, using the plotting software of your choice.

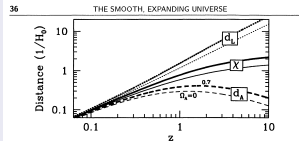


Figure 2.3. Three distance measures in a flat expanding universe. From top to bottom, the luminosity distance, the comoving distance, and the angular diameter distance. The pair of lines in each case is for a flat universe with matter only (light curves) and 70% cosmological constant  $\Lambda$  (heavy curves). In a  $\Lambda$ -dominated universe, distances out to fixed redshift are larger than in a matter-dominated universe.

## Exercise IIb

Add a new matter species with an equation of state  $p = w\rho$ . Visualise its evolution with time. Provoke an error on purpose, to check the error format.



# Error management in class

By following a few general rules, we get automatically some very informative error messages like:

```
Error in thermodynamics_init
=>thermodynamics_init(L:292) :error in
  thermodynamics_helium_from_bbn(ppr,pba,pth);
=>thermodynamics_helium_from_bbn(L:1031) :condition (omega_b
  > omegab[num_omegab-1]) is true; You have asked for an
  unrealistic high value omega_b = 7.350000e-02. The
  corresponding value of the primordial helium fraction
  cannot be found in the interpolation table. If you
  really want this value, you should fix YHe to a given
  value rather than to BBN
```

We only wrote the piece starting with “You have asked...”. All the rest was generated automatically by the code. This follows from following everywhere 5 rules.

# Error management in class

## Rule 1:

All functions are of type `int`, and return either `_SUCCESS_` or `_FAILURE_` (defined internally in `include/common.h`: `#define _SUCCESS_ 0` , `#define _FAILURE_ 1` )

```
int function(input, &output) {  
    ...  
    if (something goes wrong) return _FAILURE_;  
    ...  
    return _SUCCESS_;  
}
```

# Error management in class

## Rule 2:

All functions are called with the macro `class_call(.,.,.)` (all macros `class_xxx(...)` are defined in `include/common.h`):

```
class_call(function(input, &output),
            error_message_from_function,
            error_message_output);
```

This is simply a short-cut for

```
if (function == _FAILURE_) {
    ErrorMsg Transmit_Error_Message;
    sprintf(Transmit_Error_Message, "%s(L:%d) : error in %s;\n",
            n=>%s", __func__, __LINE__, #function,
            error_message_from_function);
    sprintf(error_message_output, "%s", Transmit_Error_Message);
    return _FAILURE_;
}
```

# Error management in class

## Rule 3:

Each of the 9 main structures `xx` has a field called `error_message`. Any function in the module `xxx.c` is called `xxx_something()` and writes its error message in `xx.error_message` (if `pxx` is a pointer to `xx`, in `pxx->error_message`).

So if we are in `perturb_init()` and we call `perturb_indices()` we write:

```
class_call(perturb_indices(...,ppt),
           ppt->error_message,
           ppt->error_message);
```

But if we are in `perturb_init()` and we call `background_at_tau()` we write:

```
class_call(background_at_tau(...,pba),
           pba->error_message,
           ppt->error_message);
```

# Error management in class

## Rule 4:

Whenever an error could occur, we first write a test with the macro

`class_test(.,.,.)`:

```
class_test(condition, error_message, "Some text");
```

or

```
class_test(condition, error_message, "Some text and numbers  
%d %e",n,x);
```

Example:

```
class_test(num_points == 0,  
          ppt->error_message,  
          "this might be caused by ...");  
step = (max-min)/((double)num_points);
```

In the text, no need to say in which function we are, or to write that the number of points is zero, or to put a `\n`, all this is done automatically.

# Error management in class

## Rule 5:

Always allocate memory with the macros `class_alloc()`, `class_calloc()`, `class_realloc()`.

Instead of

```
malloc(parray, N*sizeof(double));
```

use

```
class_alloc(parray, N*sizeof(double), pxx->error_message);
```

If allocation fails (N too big, null or negative), the function will automatically return a `_FAILURE_` and the code will return an appropriate error message:

```
Error running background_init
=>background_init(L:537):error in background_solve(ppr,pba);
=>background_solve(L:1303):could not allocate pvecback with
    size -8
```

# Error management in class

Final remark: in `main/class.c` there is no “higher level” so the 10 initialisation functions are called like e.g.:

```
int main(int argc, char **argv) {
    ....
    if (background_init(&pr,&ba) == _FAILURE_) {
        printf("\n\nError running background_init \n=>%s\n",ba.
            error_message);
        return _FAILURE_;
    }
    ...
}
```

But when `CLASS` is called as a function from another code (e.g. Monte Python or `test/test_loops.c`) we can use the standard way:

```
int class(...,ErrorMsg errmsg) {
    ....
    class_call(background_init(ppr,pba),
                pba->error_message,
                errmsg);
    ...
}
```

Then, `CLASS` never crashes (in principle...), it only returns `class(...)== _FAILURE_`