Indexing and error management

Program: Day 2

09:30-10:15	CLASS	Dynamical indexing and error management.	JL
10:15-11:00	CLASS	How the input module works.	TT
		Coffee	
11:30-12:15	CLASS	The python wrapper classy.py	TT
		Lunch	
13:30-14:15	MontePython	Basic runs.	BA
14:15-15:00	MontePython	Analyzing runs.	BA
		Tea	
15:45-16:30	Optional	Lecturers will answer questions	

Dynamical indexing

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

Dynamical indexing

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

Dynamical indexing

- Indexing is very generic in CLASS, same rules apply everywhere.
- Example: we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abreviation 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.
- Example: we want to define the indices of a vector of background quantities (stored in the background table).
- We choose an abreviation 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;
}
```

Dynamical indexing

This logic is followed everywhere for all ousts of indices! Examples:

- in `background.c`: `index_bg_...` for all background variables
- in `background.c`: `index_bi_...` subset of backg. var. integrated over time
- in `thermodynamics.c`: `index_th_...` for all thermodynamics variables
- in `perturbations.c`: `index_pt_...` perturbation var. integrated over time
- in `perturbations.c`: `index_mt_...` metric perturbations
- in `perturbations.c`: `index_md_...` list of modes (scalar, vector, tensor)
- in `perturbations.c`: `index_ic_...` list of initial conditions (AD, CDI, NID...)
- in `perturbations.c`: `index_tp_...` list of type of required source (temperature, polarisation, matter fluctuation...)
- in `perturbations.c`: `index_ap_...` list of approximation that may be used
- etc. etc.

Check in your `include/*.h` files!

Error management in `class`

Run with an input file containing only

```
omega_b = 0.07
```

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
    corrresponding 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=>%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

Useful CLASS macros:

```
class_call(function, errmsg_input, errmsg_output);
class_call_parallel(...);
class_call_except(...,[line of code;line of code;...;]);

class_test(condition, errmsg_output,"message"[,args]);
class_test_parallel(...);
class_test_except(...,[line of code;line of code;...;]);
class_stop(errmsg_ouput,"message"[,args]);

class_alloc(pointer,size);
class_alloc_parallel(...);
class_realloc(...);
class_calloc(...);
```

You can see them in `include/common.h` files!

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_;
    }
    ...
```

You can now read and understand the `main/class.c` !