CLASS

the Cosmological Linear Anisotropy Solving System[1]

Julien Lesgourgues
TTK, RWTH Aachen University

CCA, New York, 15-16.07.2019

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

Lecture 3: coding with class

1. overall structure of class
2. dynamical indexing rules
3. input parameters
4. error management rules
5. adding features
6. adding parameters in the wrapper
7. interface with samplers

Overall structure of class

In CLASS, what is a module?

- a file include/xxx.h containing some declarations
- a file source/xxx.c containing some functions
- each module is a associated with a structure xx, containing all what *other* modules need to know, and nothing else
- some fields in this structure are filled in the input.c module (input parameters relevant for this module)
- all other fields are filled by a function xxx_init(...)
- "executing a module" ≡ calling xxx_init(...)

In include/background.h: localise struct background
In source/background.c: localise background_init()

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0 - \tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0 - \tau))$,
and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of class

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0 - \tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ
thermodynamics.c	thermo	th	pth	thermo. quantities as funct. of z

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0 - \tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
`input.c`	`precision`	`pr`	`ppr`	all precision parameters
`background.c`	`background`	`ba`	`pba`	background quantities as funct. of τ
`thermodynamics.c`	`thermo`	`th`	`pth`	thermo. quantities as funct. of z
`perturbations.c`	`perturbs`	`pt`	`ppt`	source functions $S^i(k,t)$

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0-\tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
`input.c`	`precision`	`pr`	`ppr`	all precision parameters
`background.c`	`background`	`ba`	`pba`	background quantities as funct. of τ
`thermodynamics.c`	`thermo`	`th`	`pth`	thermo. quantities as funct. of z
`perturbations.c`	`perturbs`	`pt`	`ppt`	source functions $S^i(k,t)$
`primordial.c`	`primordial`	`pm`	`ppm`	primordial spectra $\mathcal{P}(k)$

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0-\tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of class

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ
thermodynamics.c	thermo	th	pth	thermo. quantities as funct. of z
perturbations.c	perturbs	pt	ppt	source functions $S^i(k,t)$
primordial.c	primordial	pm	ppm	primordial spectra $\mathcal{P}(k)$
nonlinear.c	nonlinear	nl	pnl	non-linear corrections $\alpha_{\rm NL}(k,\tau)$

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0-\tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ
thermodynamics.c	thermo	th	pth	thermo. quantities as funct. of z
perturbations.c	perturbs	pt	ppt	source functions $S^i(k,t)$
primordial.c	primordial	pm	ppm	primordial spectra $\mathcal{P}(k)$
nonlinear.c	nonlinear	nl	pnl	non-linear corrections $\alpha_{\rm NL}(k,\tau)$
transfer.c	transfers	tr	ptr	harmonic transfer functions $\Delta_l^i(k)$

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0 - \tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of class

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ
thermodynamics.c	thermo	th	pth	thermo. quantities as funct. of z
perturbations.c	perturbs	pt	ppt	source functions $S^i(k,t)$
primordial.c	primordial	pm	ppm	primordial spectra $\mathcal{P}(k)$
nonlinear.c	nonlinear	nl	pnl	non-linear corrections $\alpha_{\rm NL}(k,\tau)$
transfer.c	transfers	tr	ptr	harmonic transfer functions $\Delta_l^i(k)$
spectra.c	spectra	sp	psp	linear and/or non-linear $P(k,z)$, C_ℓ's

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0-\tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ
thermodynamics.c	thermo	th	pth	thermo. quantities as funct. of z
perturbations.c	perturbs	pt	ppt	source functions $S^i(k,t)$
primordial.c	primordial	pm	ppm	primordial spectra $\mathcal{P}(k)$
nonlinear.c	nonlinear	nl	pnl	non-linear corrections $\alpha_{\rm NL}(k,\tau)$
transfer.c	transfers	tr	ptr	harmonic transfer functions $\Delta_l^i(k)$
spectra.c	spectra	sp	psp	linear and/or non-linear $P(k,z)$, C_ℓ's
lensing.c	lensing	le	ple	lensed CMB C_ℓ's

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0 - \tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

List of structures associated to modules:

module	structure	ab.	*	main content
input.c	precision	pr	ppr	all precision parameters
background.c	background	ba	pba	background quantities as funct. of τ
thermodynamics.c	thermo	th	pth	thermo. quantities as funct. of z
perturbations.c	perturbs	pt	ppt	source functions $S^i(k,t)$
primordial.c	primordial	pm	ppm	primordial spectra $\mathcal{P}(k)$
nonlinear.c	nonlinear	nl	pnl	non-linear corrections $\alpha_{\rm NL}(k,\tau)$
transfer.c	transfers	tr	ptr	harmonic transfer functions $\Delta_l^i(k)$
spectra.c	spectra	sp	psp	linear and/or non-linear $P(k,z)$, C_ℓ's
lensing.c	lensing	le	ple	lensed CMB C_ℓ's
output.c	output	op	pop	description of output format

In a flat universe, line-of-sight integrals read $\Delta_l^i(k) = \int d\tau S^i(k,\tau) j_l(k(\tau_0-\tau))$, and harmonic spectra are given by $C_l^{ij} = 4\pi \int \frac{dk}{k} \mathcal{P}(k) \Delta_l^i(k) \Delta_l^j(k)$.

Overall structure of `class`

Each module contains:

- a function `xxx_init(...)` filling the structure `xx`
- a function `xxx_free(...)` freeing all the memory allocated to this structure
- some functions `xxx_external_1(...)`, ..., `xxx_external_n(...)` that can be called from other modules (e.g. to read correctly or interpolate the content of the structure `xx`)
- some functions `xxx_internal_1(...)`, ..., `xxx_internal_m(...)` that are called only inside the module, within `xxx_init(...)`

Overall structure of `class`

Each module contains:

- a function `xxx_init(...)` filling the structure `xx`
- a function `xxx_free(...)` freeing all the memory allocated to this structure
- some functions `xxx_external_1(...)`, ..., `xxx_external_n(...)` that can be called from other modules (e.g. to read correctly or interpolate the content of the structure `xx`)
- some functions `xxx_internal_1(...)`, ..., `xxx_internal_m(...)` that are called only inside the module, within `xxx_init(...)`

Following order always respected in `xxx.c`:

```
xxx_external_1(...)
...
xxx_external_n(...)
xxx_init(...)
xxx_free(...)
xxx_internal_1(...)
...
xxx_internal_m(...)
```

Overall structure of class

Each module contains:

- a function xxx_init(...) filling the structure xx
- a function xxx_free(...) freeing all the memory allocated to this structure
- some functions xxx_external_1(...), ..., xxx_external_n(...) that can be called from other modules (e.g. to read correctly or interpolate the content of the structure xx)
- some functions xxx_internal_1(...), ..., xxx_internal_m(...) that are called only inside the module, within xxx_init(...)

Following order always respected in xxx.c:

```
xxx_external_1(...)
...
xxx_external_n(...)
xxx_init(...)
xxx_free(...)
xxx_internal_1(...)
...
xxx_internal_m(...)
```

Remark: a module in the CLASS code is very similar to a "class" in C++. We enjoy the structure of C++ with the speed and readability of C.

Overall structure of `class`

Following order always respected in `xxx.c`:

```
xxx_external_1(...)
...
xxx_external_n(...)
xxx_init(...)
xxx_free(...)
xxx_internal_1(...)
...
xxx_internal_m(...)
```

Count number of external and internal functions in `source/background.c`:
Search "`int background_`" starting from top

Overall structure of class

The main() function of CLASS located in main/class.c could only contain:

```
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)
 /* all calculations done, free the structures */
 lensing_free(ple);
 spectra_free(psp);
 transfer_free(ptr);
 nonlinear_free(pnl);
 primordial_free(ppm);
 perturb_free(ppt);
 thermodynamics_free(pth);
 background_free(pba);
}
```

Dynamical indexing rules in `class`

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

Dynamical indexing rules in class

- 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 rules in `class`

- 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 rules in class

- 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 `include/background.h` (more precisely, inside the structure background, because these indices are necessary for manipulating the background table).

Dynamical indexing rules in class

- 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 `include/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 rules in class

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 rules in class

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 rules in class

This logic is followed everywhere for all groups 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 approximations that may be used
- etc. etc.

Check in your include/*.h files!

Input management in class

Terminal	Python wrapper
file `xxx.ini` ↓ `input_init_from_argument(...)` (parser) ↓	`.set(...)` ↓

`struct file_content fc;` (all parameter names/values stored as arrays of strings)

↓

`input_init(...)`

↓

`input_read_parameters(...)`

(assign all default values + interprete input + update some parameters)

↓

relevant parameters only get stored in the structures of each module

For special parameters requiring a shooting method: repeated calls of `input_read_parameters(...)` from `input_init(...)` until shooting target is met.

Input management in class

For normal parameters (no shooting): example of CDM density:

```
/** - Omega_0_cdm (CDM) */
class_call(parser_read_double(pfc,"Omega_cdm",&param1,&
    flag1,errmsg),
           errmsg,
           errmsg);
class_call(parser_read_double(pfc,"omega_cdm",&param2,&
    flag2,errmsg),
           errmsg,
           errmsg);
class_test(((flag1 == _TRUE_) && (flag2 == _TRUE_)),
           errmsg,
           "In input file, you can only enter one of
               Omega_cdm or omega_cdm, choose one");
if (flag1 == _TRUE_)
  pba->Omega0_cdm = param1;
if (flag2 == _TRUE_)
  pba->Omega0_cdm = param2/pba->h/pba->h;
```

For shooting parameters, establish mapping between *target parameter*, *unknown parameter* and *level*. Currently:

target parameter	unknown parameter	level
$100 \times \theta_s$	h	thermodynamics
σ_8	A_s	spectra
$\Omega_{\rm dcdm}$	$\rho^{\rm ini}_{\rm dcdm}$	background
...	...	...

... plus a few others (alternative parametrizations of decaying CDM, quintessence parameters).

If you need to add such parameters: see how it is done e.g. for `100*theta_s` and replicate the structure!

Error management rules in class

Run with an input file containing only

```
omega_b = 0.07
```

Error management rules 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.e-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 rules 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 rules 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 rules 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 rules 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 rules 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 rules 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 rules in class

Few special cases:

- 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_;
    }
```

Error management rules in class

Few special cases:

- 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_;
    }
```

- the input module does not have an error message attached to its structure, and just uses the local variable errmsg. So inside this module, the calls read e.g.:

```
class_call(background_ncdm_init(ppr,pba),
           pba->error_message,
           errmsg);
class_call(parser_read_file(...,errmsg),
           errmsg,
           errmsg);
```

Error management rules in class

Few special cases:

- 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_;
    }
```

- the input module does not have an error message attached to its structure, and just uses the local variable errmsg. So inside this module, the calls read e.g.:

```
class_call(background_ncdm_init(ppr,pba),
           pba->error_message,
           errmsg);
class_call(parser_read_file(...,errmsg),
           errmsg,
           errmsg);
```

- when calling external functions not in the 10 modules we must pass the error message as an argument:

```
class_call(array_interpolate(...,pba->error_message),
           pba->error_message,
           pba->error_message);
```

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside “redefine”, “needed”, etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some “`if (has_fld){ ...}`” and you can search directly for occurences of `has_fld`)

Implementing new features class

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)
4. duplicate these occurences

Implementing new features class

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)
4. duplicate these occurences
5. change `fld` into `earde`

Implementing new features `class`

If you want to implement:

- a new species
- a new approximation scheme to simplify some equations in some regime
- a new mathematical description of an existing species (switching on more precise corrections, etc.)
- a new observable or output (new source function, new transfer function, new spectrum...)

the logic is always the same:

1. define an acronym easy to search in the C files (e.g. for early dark energy: `earde` is good, `ede` is bad because it is inside "redefine", "needed", etc.)
2. think of the feature closest to yours, and find its acronym (e.g. for fluid: `fld`)
3. grep for all occurences of `fld` in `include/*.h` and `source/*.c` (normally they are all within some "`if (has_fld){ ...}`" and you can search directly for occurences of `has_fld`)
4. duplicate these occurences
5. change `fld` into `earde`
6. change some equations to describe the specific properties of your feature

Adding parameters in the wrapper

Example of `python` wrapper:

```
# redeclaration of relevant CLASS variables in cython
python/cclassy.pxd
# wrapper's function (.set(), .compute(), .lensed_cl(), ...)
python/classy.py
```

Don't edit any other! (generated automatically at compilation, or for testing or module installation)

Adding parameters in the wrapper

In python/cclassy.pxd relevant variables redeclared inside the structure to which they belong:

```
cdef struct background:
    ...
    double age
    ...
cdef struct thermo
    ...
    double z_reio
    ...
```

Indeed, in the C code, `pba->age`, `pth->z_reio` exist...
When defining new parameter in C code that should be accessible from outside: redeclare them here!

Adding parameters in the wrapper

E.g.: new model of Presidential Dark Matter.

In `include/background.h`:

```
struct background{
  ...
  double rho_trump;
  ...
  }
```

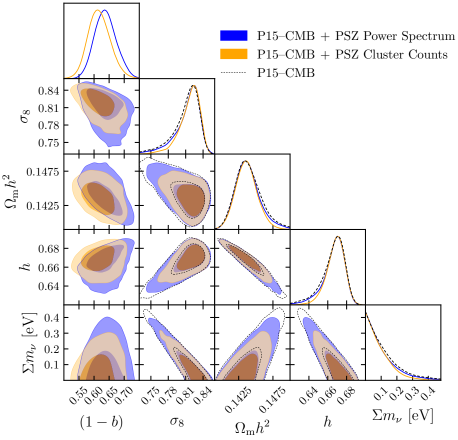
In `python/cclassy.pxd`:

```
cdef struct background:
  ...
  double rho_trump
  ...
```

Recompile after a `make clean` !

Many are compatible with CLASS! Non-exhaustive list:

- Bayesian samplers:
  - MontePython (B. Audren, T. Brinckmann, J.L. + others), python, https://github.com/brinckmann/montepython_public



Bolliet et al. 2019

Interface with sampler

Many are compatible with CLASS! Non-exhaustive list:

- Bayesian samplers:
  - MontePython (B. Audren, T. Brinckmann, J.L. + others), python,
    https://github.com/brinckmann/montepython_public
  - cobaya (J. Torrado, A. Lewis), python,
    https://cobaya.readthedocs.io
  - cosmosis (J. Zuntz), python,
    https://bitbucket.org/joezuntz/cosmosis
- Frequentist minimizers:
  - CAMEL (LAL Orsay), C++, uses MINUIT,
    http://camel.in2p3.fr

Interface with sampler

Many are compatible with CLASS! Non-exhaustive list:

- Bayesian samplers:
  - MontePython (B. Audren, T. Brinckmann, J.L. + others), python, https://github.com/brinckmann/montepython_public
  - cobaya (J. Torrado, A. Lewis), python, https://cobaya.readthedocs.io
  - cosmosis (J. Zuntz), python, https://bitbucket.org/joezuntz/cosmosis
- Frequentist minimizers:
  - CAMEL (LAL Orsay), C++, uses MINUIT, http://camel.in2p3.fr

At least in MontePython, Cobaya and CAMEL: no declaration of cosmological parameters in the sampler! No need to modify anything if you add new parameters! (whatever parameter `'name'` read in input file just passed directly through `class.set('name':,...)`