Tools for Cosmology: The CLASS and Monte Python codes

Benjamin Audren[(a)], Julien Lesgourgues[(a,b,c)], Thomas Tram[(a)]

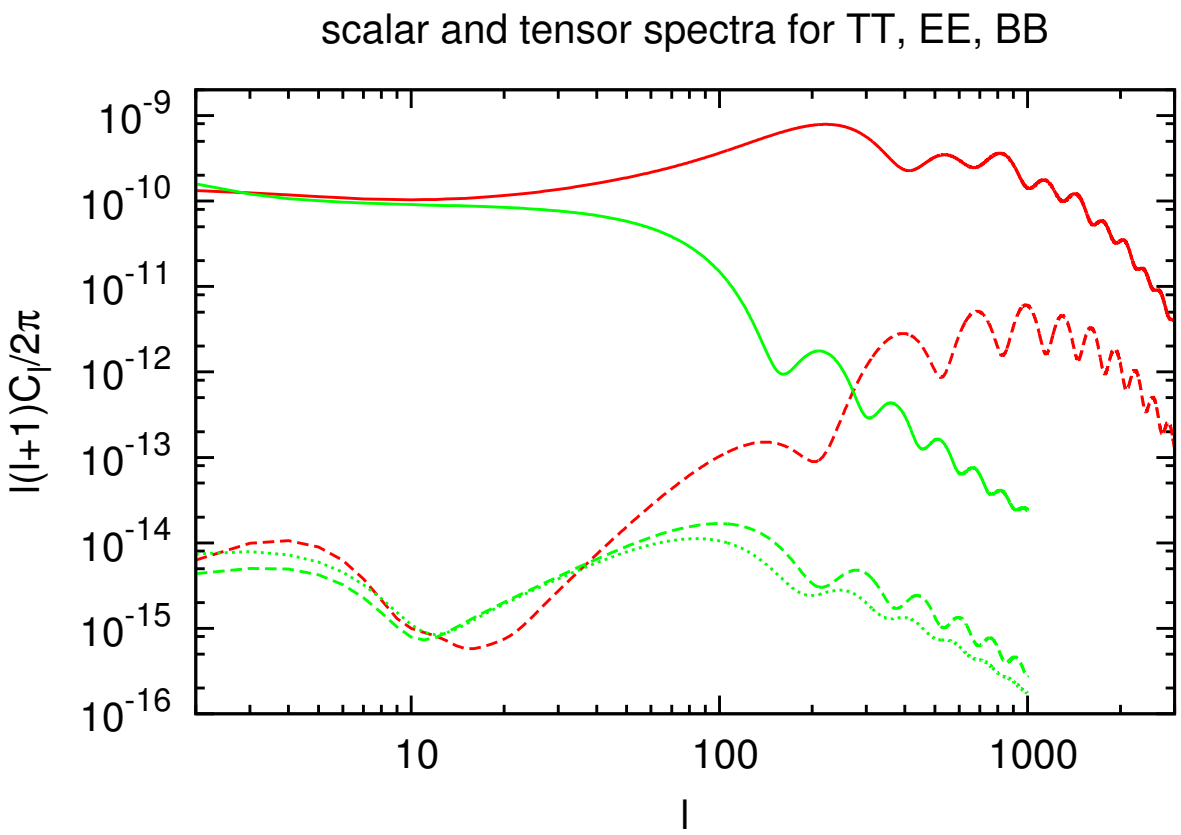
[(a)] EPFL, [(b)] CERN, [(c)] LAPTh

Geneva, 31.03.2014

UNIVERSITÉ DE GENÈVE

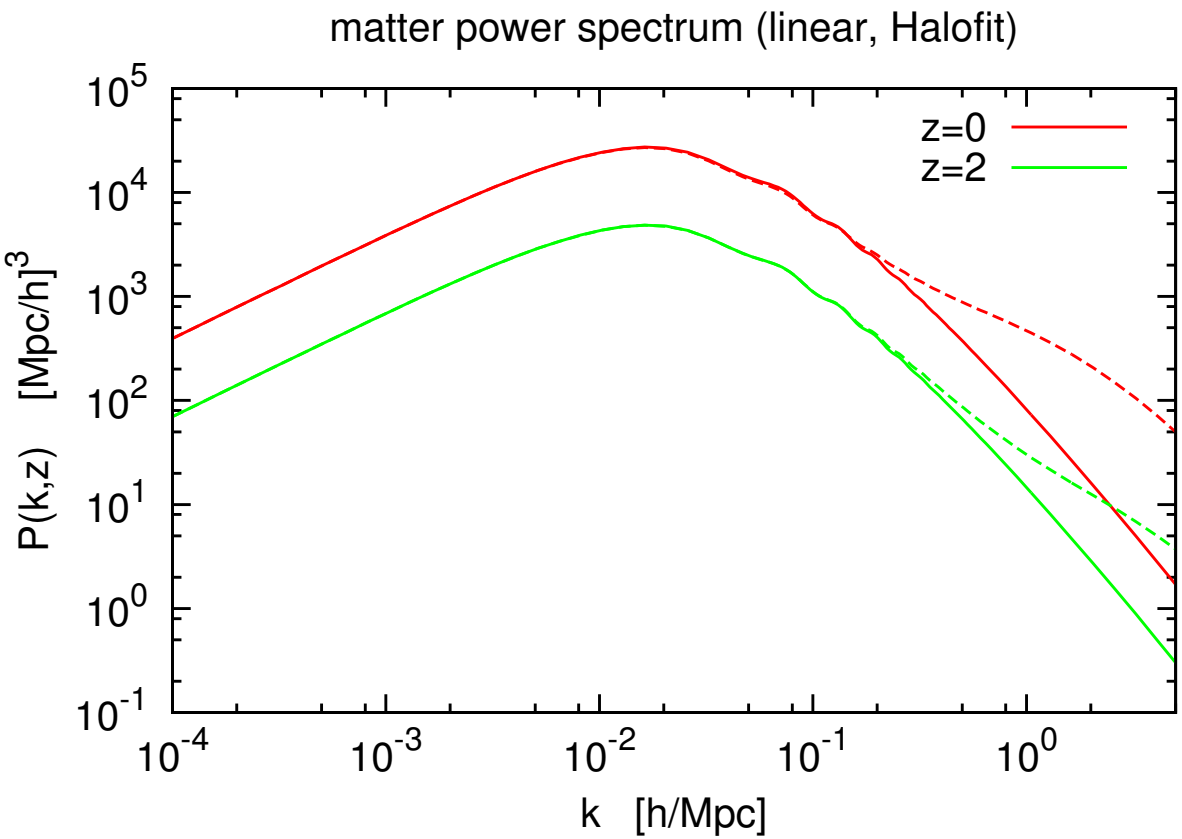
Almost any type of research activity in cosmology will use a Boltzmann code at some point

Computing CMB anisotropy spectra:



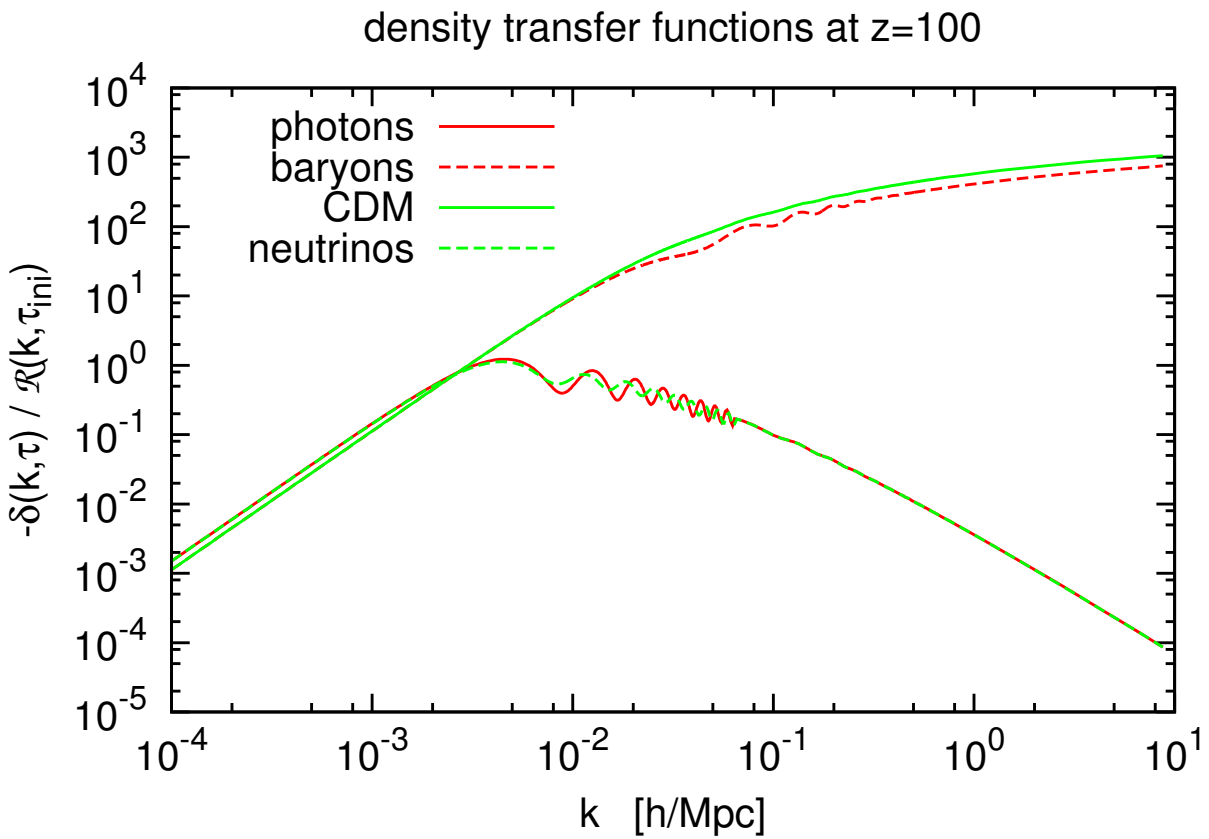
Almost any type of research activity in cosmology will use a Boltzmann code at some point

Computing matter power spectrum:



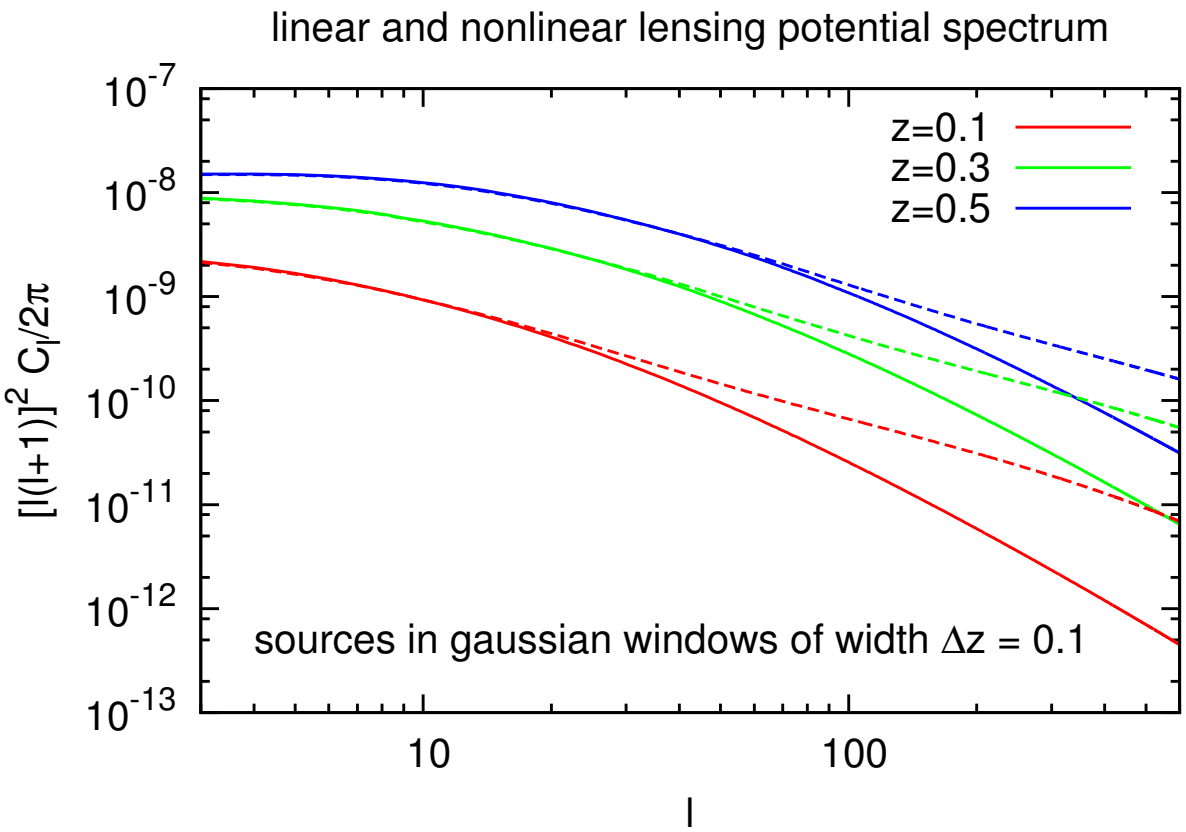
Almost any type of research activity in cosmology will use a Boltzmann code at some point

Computing transfer functions (e.g. initial conditions for N-body):



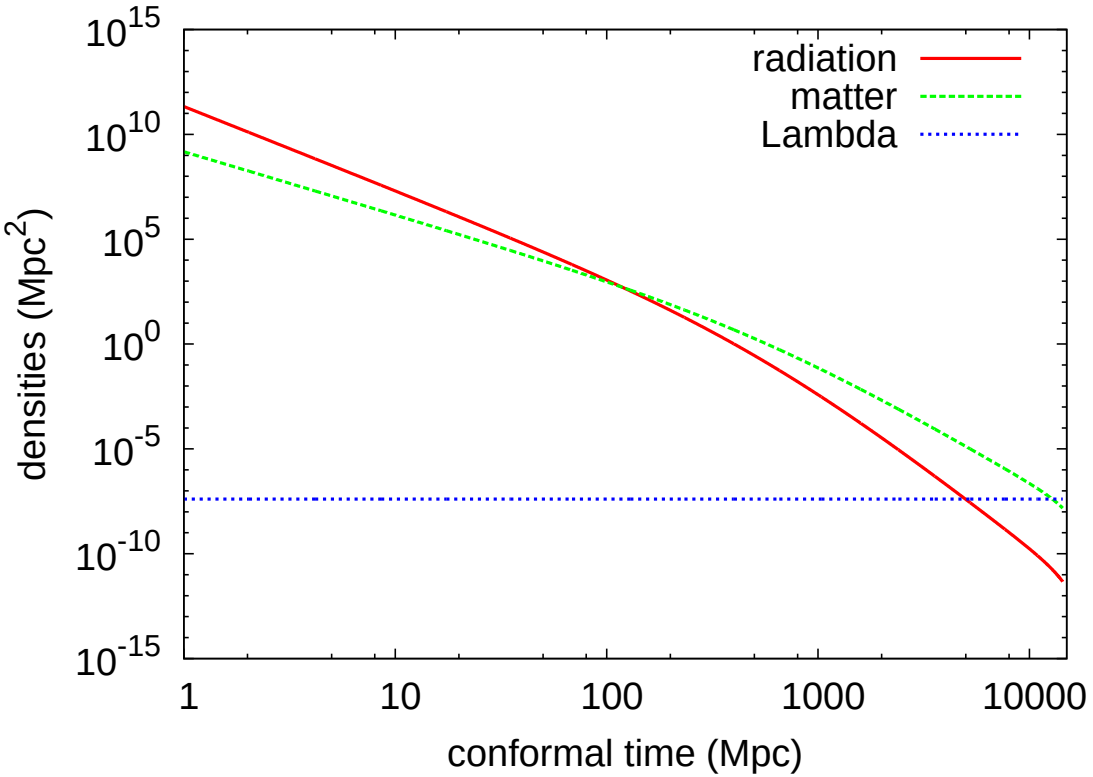
Targets

Computing matter density (number count) spectra, or lensing angular spectra:



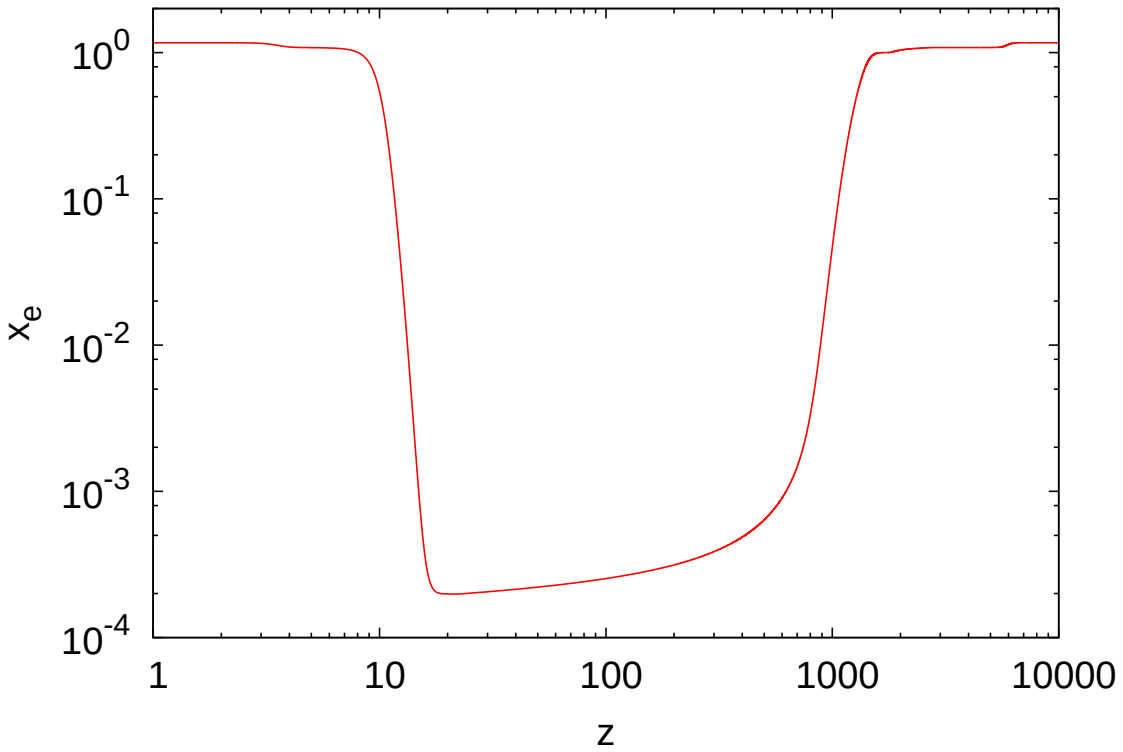
Almost any type of research activity in cosmology will use a Boltzmann code at some point

Computing background evolution in a given cosmological model:



Almost any type of research activity in cosmology will use a Boltzmann code at some point

Computing thermal history in a given cosmological model:



Almost any type of research activity in cosmology will use a Boltzmann code at some point

As an observer...

... you want to compute these quantities easily and efficiently
... you may want to develop the code for outputting new observables

Almost any type of research activity in cosmology will use a Boltzmann code at some point

As an observer...

... you want to compute these quantities easily and efficiently
... you may want to develop the code for outputting new observables

As a theorist...

... you may want to develop the code for incorporating new physics
and compute/understand the effects of your favourite model on observables

Almost any type of research activity in cosmology will use a Boltzmann code at some point

As an observer...

... you want to compute these quantities easily and efficiently
... you may want to develop the code for outputting new observables

As a theorist...

... you may want to develop the code for incorporating new physics
and compute/understand the effects of your favourite model on observables

As both...

... you want to infer constraints on cosmological parameters from a new dataset
... you want to test your own favorite model, given existing data
... you want to predict the sensitivity of a future experiment to a given parameter

Targets

- we will see how to reach the first two targets with the Cosmic Linear Anisotropy Solving System (CLASS)
- we will see how to reach the third target with a Monte Carlo code in Python, Monte Python
- good occasion to refresh our mind or understand better the underlying theory! We will do that on-the-fly, since the structure of the CLASS code is the same as the sections of a cosmology text book.

Targets

- we will see how to reach the first two targets with the Cosmic Linear Anisotropy Solving System (CLASS)
- we will see how to reach the third target with a Monte Carlo code in Python, Monte Python
- good occasion to refresh our mind or understand better the underlying theory! We will do that on-the-fly, since the structure of the CLASS code is the same as the sections of a cosmology text book.

So there will be:

1. lectures on CLASS (numerics + underlying physics) by JL, some aspects will be developed by TT
2. lectures on Monte Pyhton (underlying statistics + use of the code) by BA
3. exercise sessions on both codes, tutored by the 3 of us

Targets

Not difficult from numerical point of view.
Ideally, basic knowledge of C and Python required.
If not, no problem: for the exercises, we will mainly copy existing structures in the code.
Moreover C is very similar to fortran, and Python has similarities with Matlab, Mathematica or IDL.

Program: Day 1

	DAY I : Monday 31th March	
10:00-11:00	Introduction to CLASS. *Brief history of Boltzmann codes.* *Goals and philosophy of CLASS.* *Structure of the code.*	JL
11:00-11:20	Coffee	
11:20-11:50	Introduction to CLASS. *Basic input and output.* *Plotting facilities.*	JL
11:50-12:50	CLASS Practise. *Looking at all possible outputs of CLASS.* *Vizualizing them using the CLASS Plotting Unit, Gnuplot, Matlab, or one's own favorite plotting software.*	BA, JL, TT
12:50-14:00	Lunch break	
14:00-15:00	Cosmological parameter extraction from data. *Overview of the main methods and existing codes.*	BA
15:00-15:15	pause	
15:15-16:15	Introduction to Monte Python I. *Goals of Monte Python.* *Installation and basic use.*	BA

CLASS: Lecture I

Introduction to `class`

- Motivations and goals of `class`
- The philosophy of `class`: how to achieve friendliness and flexibility
- Overall structure of the code
- Input file/parameters
- Output files

Next lectures will describe the modules one-by-one: background, thermodynamics, perturbations, etc.

Brief history of Boltzmann codes

- 1995: Bertschinger releases the COSMICS package in f77. Contains Ma & Bertschinger (`astro-ph/9506072`) equations in synchronous gauge, Peebles recombination. Integration of Boltzmann eq. for photons/neutrinos till $\ell \sim 2500$.

Brief history of Boltzmann codes

- 1995: Bertschinger releases the COSMICS package in f77. Contains Ma & Bertschinger (`astro-ph/9506072`) equations in synchronous gauge, Peebles recombination. Integration of Boltzmann eq. for photons/neutrinos till $\ell \sim 2500$.
- 1996: Seljak & Zaldarriaga add a few functions for computing the source functions and convolve them with Bessel functions. New code much faster, released as CMBFAST.

Brief history of Boltzmann codes

- 1995: Bertschinger releases the COSMICS package in f77. Contains Ma & Bertschinger (`astro-ph/9506072`) equations in synchronous gauge, Peebles recombination. Integration of Boltzmann eq. for photons/neutrinos till $\ell \sim 2500$.
- 1996: Seljak & Zaldarriaga add a few functions for computing the source functions and convolve them with Bessel functions. New code much faster, released as CMBFAST.
- CMBFAST improved: RECFAST recombination, open/closed, lensing, but structure of the code becomes complicated.

Brief history of Boltzmann codes

- 1995: Bertschinger releases the COSMICS package in f77. Contains Ma & Bertschinger (`astro-ph/9506072`) equations in synchronous gauge, Peebles recombination. Integration of Boltzmann eq. for photons/neutrinos till $\ell \sim 2500$.
- 1996: Seljak & Zaldarriaga add a few functions for computing the source functions and convolve them with Bessel functions. New code much faster, released as CMBFAST.
- CMBFAST improved: RECFAST recombination, open/closed, lensing, but structure of the code becomes complicated.
- 1999: Lewis et al. cut CMBFAST in pieces and reorganize them differently in f90 in CAMB. Improved expressions for sources, initial conditions, lensing, etc.

Brief history of Boltzmann codes

- 1995: Bertschinger releases the COSMICS package in f77. Contains Ma & Bertschinger (`astro-ph/9506072`) equations in synchronous gauge, Peebles recombination. Integration of Boltzmann eq. for photons/neutrinos till $\ell \sim 2500$.
- 1996: Seljak & Zaldarriaga add a few functions for computing the source functions and convolve them with Bessel functions. New code much faster, released as CMBFAST.
- CMBFAST improved: RECFAST recombination, open/closed, lensing, but structure of the code becomes complicated.
- 1999: Lewis et al. cut CMBFAST in pieces and reorganize them differently in f90 in CAMB. Improved expressions for sources, initial conditions, lensing, etc.
- 2003: Doran does a similar work of reorganization in C++: CMBEASY

Brief history of Boltzmann codes

- 1995: Bertschinger releases the COSMICS package in f77. Contains Ma & Bertschinger (`astro-ph/9506072`) equations in synchronous gauge, Peebles recombination. Integration of Boltzmann eq. for photons/neutrinos till $\ell \sim 2500$.
- 1996: Seljak & Zaldarriaga add a few functions for computing the source functions and convolve them with Bessel functions. New code much faster, released as CMBFAST.
- CMBFAST improved: RECFAST recombination, open/closed, lensing, but structure of the code becomes complicated.
- 1999: Lewis et al. cut CMBFAST in pieces and reorganize them differently in f90 in CAMB. Improved expressions for sources, initial conditions, lensing, etc.
- 2003: Doran does a similar work of reorganization in C++: CMBEASY
- later: CAMB maintained and improved over the years; others not.

What would be expected from a new Boltzmann code?

- Friendly and flexible: should be easy to compile, to pass input parameters, to understand the code, and to modify it (extended cosmological scenarios, new observables).

What would be expected from a new Boltzmann code?

- Friendly and flexible: should be easy to compile, to pass input parameters, to understand the code, and to modify it (extended cosmological scenarios, new observables).
- Accurate: need more and more precision. Analyzing Planck and WMAP data required very different accuracy settings. Before, CAMB precision could only be calibrated w.r.t itself. CLASS played important role in pushing precision to Planck level. Similar efforts in the future (LSS, next CMB satellite, 21cm, etc.)

What would be expected from a new Boltzmann code?

- Friendly and flexible: should be easy to compile, to pass input parameters, to understand the code, and to modify it (extended cosmological scenarios, new observables).
- Accurate: need more and more precision. Analyzing Planck and WMAP data required very different accuracy settings. Before, CAMB precision could only be calibrated w.r.t itself. CLASS played important role in pushing precision to Planck level. Similar efforts in the future (LSS, next CMB satellite, 21cm, etc.)
- Fast: for parameter extraction (Metropolis-Hastings, Multinest, Cosmo Hammer, grid-base methods). Typical project: 10'000 to 1'000'000 executions

What would be expected from a new Boltzmann code?

- Friendly and flexible: should be easy to compile, to pass input parameters, to understand the code, and to modify it (extended cosmological scenarios, new observables).
- Accurate: need more and more precision. Analyzing Planck and WMAP data required very different accuracy settings. Before, CAMB precision could only be calibrated w.r.t itself. CLASS played important role in pushing precision to Planck level. Similar efforts in the future (LSS, next CMB satellite, 21cm, etc.)
- Fast: for parameter extraction (Metropolis-Hastings, Multinest, Cosmo Hammer, grid-base methods). Typical project: 10'000 to 1'000'000 executions

... three goals of the Cosmic Linear Anisotropy Solving System (CLASS)

The philosophy of `class`

Our efforts for ensuring flexibility and friendliness in CLASS, summarised in 14 key points

Friendliness and flexibility in 14 points

1. Written in plain C with no external libraries

C is free, diffuse, easy, fast (more than C++). Self-contained and ready to install, straightforward to compile.

Friendliness and flexibility in 14 points

1. Written in plain C with no external libraries

C is free, diffuse, easy, fast (more than C++). Self-contained and ready to install, straightforward to compile.

2. Input parameters are "interpreted"

Some basic logic has been incorporated in the code. Easy to elaborate further.

Examples:
- if H_0 is passed, do not expect h, otherwise, complain; and vice-versa.
- If $\{H_0$ or $h\} + \{T_{\rm cmb}$ or Ω_γ or $\omega_\gamma\}$ are passed, infer the missing ones; if more are passed, complain.

Friendliness and flexibility in 14 points

1. Written in plain C with no external libraries

C is free, diffuse, easy, fast (more than C++). Self-contained and ready to install, straightforward to compile.

2. Input parameters are "interpreted"

Some basic logic has been incorporated in the code. Easy to elaborate further.

Examples:
- if H_0 is passed, do not expect h, otherwise, complain; and vice-versa.
- If $\{H_0$ or $h\} + \{T_{\rm cmb}$ or Ω_γ or $\omega_\gamma\}$ are passed, infer the missing ones; if more are passed, complain.

3. Perturbation equations and notations taken literally from well-known Ma & Bertschinger (`astro-ph/9506072`) paper ...

... rather than specific notations of one given group, or mixed notations from various origins.

For non-flat universes we found and published the simplest possible generalisation of Ma & Bertschinger notations, (`arXiv:1305.3261`).

Friendliness and flexibility in 14 points

4. Code intensively documented

As many comment lines as C lines

Friendliness and flexibility in 14 points

4. Code intensively documented

As many comment lines as C lines

5. Easy units

All important variables are either dimensionless or in Mpc^n

Friendliness and flexibility in 14 points

4. Code intensively documented

As many comment lines as C lines

5. Easy units

All important variables are either dimensionless or in Mpc^n

6. No hard coding, for example:

- Never write a sampling step scheme in physical units; code infers sampling as given fraction of dimensionless physical quantites;
- Never write the index of an array as an integer; indexing down automatically and internally by the code; use symbolic index names;

Friendliness and flexibility in 14 points

4. Code intensively documented

As many comment lines as C lines

5. Easy units

All important variables are either dimensionless or in Mpc^n

6. No hard coding, for example:

- Never write a sampling step scheme in physical units; code infers sampling as given fraction of dimensionless physical quantites;
- Never write the index of an array as an integer; indexing down automatically and internally by the code; use symbolic index names;

7. No global variables

All variables passed as arguments of functions. Important for readability and parallelisation.

Friendliness and flexibility in 14 points

8. Clear modular structure

Dinstinct modules with separate physical tasks. No duplicate equations.

E.g.: Friedmann equation appears in one single place. Same for linearised Einstein equations. Ideal for implementing modified gravity theories.

```
 1. input.c
 2. background.c
 3. thermodynamics.c
 4. perturbations.c
 5. primordial.c
 6. nonlinear.c
 7. transfer.c
 8. spectra.c
 9. lensing.c
10. output.c
```

Friendliness and flexibility in 14 points

8. Clear modular structure

Dinstinct modules with separate physical tasks. No duplicate equations.

E.g.: Friedmann equation appears in one single place. Same for linearised Einstein equations. Ideal for implementing modified gravity theories.

```
 1. input.c
 2. background.c
 3. thermodynamics.c
 4. perturbations.c
 5. primordial.c
 6. nonlinear.c
 7. transfer.c
 8. spectra.c
 9. lensing.c
10. output.c
```

9. All precision variables grouped in one single place (`input.c`), and even inside a single structure 'precision'

There are... many. True for any code, but they are usually hidden and spread!

Friendliness and flexibility in 14 points

10. Given "ingredient" always implemented between brackets, in zone switched by a flag

- adding new physics does not slow down the code or compromise its readability.
- incentive to add lots of new things even if rarely used, with no drawback.
- with a search, one can localise all the parts of the code related to a given ingredient.

Examples:

```
if (has_fld == TRUE) {...}
if (has_cmb_lensing == TRUE) {...}
```

Friendliness and flexibility in 14 points

10. Given "ingredient" always implemented between brackets, in zone switched by a flag

- adding new physics does not slow down the code or compromise its readability.
- incentive to add lots of new things even if rarely used, with no drawback.
- with a search, one can localise all the parts of the code related to a given ingredient.

Examples:

```
if (has_fld == TRUE) {...}
if (has_cmb_lensing == TRUE) {...}
```

11. Adding new ingredient...

... can be done by searching for occurrence of another similar ingredient, copy/pasting, and adapting the new lines.

Example: if you want to add a new Dark Energy component, you may search for '_fld', duplicate all corresponding lines, change '_fld' into e.g. '_myde', and adapt the physical equations.

Friendliness and flexibility in 14 points

12. Error management

In principle CLASS never crashes. In case of problem, it returns an error message, with a well-documented error (line, function, what caused the crash, suggestions on how to avoid it). Most of this message is generated automatically by the code.

Friendliness and flexibility in 14 points

12. Error management

In principle CLASS never crashes. In case of problem, it returns an error message, with a well-documented error (line, function, what caused the crash, suggestions on how to avoid it). Most of this message is generated automatically by the code.

13. Version history

Old versions can always be downloaded. In most cases, new versions feature new ingredients and avoid (whenever possible) to modify or erase the old ones. Try to develop in such way that modifications to an old version can still be pasted in a new version (as much as possible).

Friendliness and flexibility in 14 points

12. Error management

In principle CLASS never crashes. In case of problem, it returns an error message, with a well-documented error (line, function, what caused the crash, suggestions on how to avoid it). Most of this message is generated automatically by the code.

13. Version history

Old versions can always be downloaded. In most cases, new versions feature new ingredients and avoid (whenever possible) to modify or erase the old ones. Try to develop in such way that modifications to an old version can still be pasted in a new version (as much as possible).

14. Git repository and GitHub website.

The code can be downloaded as a `.tar.gz`, or as a git repository. Then, user can develop his own modification with the advantage of git (branching, memory of changes...); or merge his changes with a newer version almost automatically; or submit his modifications to the CLASS team in view of an easy merging with the public version.

Structure of the code

The directory `class/` contains subdirectories:

```
include/  # header files (*.h) containing declarations
source/   # the 10 important modules of CLASS
main/     # main CLASS function: short, just calls 10 modules
test/     # other main functions for testing part of the code
tools/    # auxiliary pieces of codes (numerical methods)
output/   # output files
python/   # python wrapper of CLASS
cpp/      # C++ wrapper of CLASS
build/    # binary files created at compilation
```

plus examples of input files, `README`, `Makefile`, and few other directories containing ancillary data or external code

Structure of the code

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" $\equiv$ calling `xxx_init(...)`

Structure of the code

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	thermodynamics	th	pth	thermo. quantities as funct. of z
perturbations.c	perturbs	pt	ppt	source functions $S(k,t)$
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	transfer functions $\Delta_l(k)$
spectra.c	spectra	sp	psp	linear and/or non-linear $P(k,z)$, C_ℓ's
lensing.c	lensing	le	ple	lensed C_ℓ's
output.c	output	op	pop	description of output format

Structure of the code

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(...)`

Structure of the code

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(...)
```

Structure of the code

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++ and the speed of C.

Structure of the code

The `main()` function of CLASS located in `main/class.c` only contains:

```
int main() {
 input_init(pfc,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);
 /* 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);
}
```

The input module

source/input.c

Input

./class can take two input files *.ini and *.pre:

```
>./class my_model.ini some_precision.pre
```

But one is enough. Syntax:

```
h = 0.7
T_cmb = 2.726 # comment
output = tCl, pCl
more comments, ignored because there is no equal sign
# comment with an =, still ignored thanks to the sharp
```

Input

`./class` can take two input files `*.ini` and `*.pre`:

```
>./class my_model.ini some_precision.pre
```

But one is enough. Syntax:

```
h = 0.7
T_cmb = 2.726 # comment
output = tCl, pCl
more comments, ignored because there is no equal sign
# comment with an =, still ignored thanks to the sharp
```

- Order of lines doesn't matter at all.
- All parameters not passed fixed to default, i.e. the most reasonable or minimalistic choice
- All possible input parameters and details on the syntax explained in `explanatory.ini`
- This is only a reference file; we advise you *never* to modify it, but rather to copy it and reduce it to a shorter and more friendly file.
- For *basic* usage: `explanatory.ini` $\equiv$ full documentation of the code

Input

For instance, we can create a very short file `lcdm.ini`:

```
******************************
* CLASS input parameter file *
******************************
----> background parameters:
H0 = 72.
omega_b = 0.0266691
omega_cdm = 0.110616
----> thermodynamics parameters:
z_reio = 10.
----> define which perturbations should be computed:
output = tCl, pCl
----> define primordial perturbation spectra:
A_s = 2.3e-9
n_s = 1.
----> parameters for the output spectra:
l_scalar_max = 2500
root = output/lcdm_
```

Input

How does the input module works?

1. "parse" (reads) the input file(s), and store their content in a list of names and values (corresponding to a structure called `file_content`)

```
struct file_content fc;
fc.name[0] = "Omega_b"; fc.value[0] = 0.04;
...
```

Input

How does the input module works?

1. "parse" (reads) the input file(s), and store their content in a list of names and values (corresponding to a structure called `file_content`)

```
struct file_content fc;
fc.name[0] = "Omega_b"; fc.value[0] = 0.04;
...
```

2. pass the structure `file_content` to the function `input_init(...)` that interprets the input parameters, set all other parameters to default values, and fills the input parameter fields of each structure.

Input

How does the input module works?

1. "parse" (reads) the input file(s), and store their content in a list of names and values (corresponding to a structure called `file_content`)

```
struct file_content fc;
fc.name[0] = "Omega_b"; fc.value[0] = 0.04;
...
```

2. pass the structure `file_content` to the function `input_init(...)` that interprets the input parameters, set all other parameters to default values, and fills the input parameter fields of each structure.

These two steps are performed inside:

```
input_init_from_arguments(argc,argv,&pr,&ba,&th,&pt,&tr,&pm
    ,&sp,&nl,&le,&op,errmsg)
input_init(&fc,&pr,&ba,&th,&pt,&tr,&pm,&sp,&nl,&le,&op,
    errmsg)
```

Input

How does the input module works?

1. "parse" (reads) the input file(s), and store their content in a list of names and values (corresponding to a structure called `file_content`)

```
struct file_content fc;
fc.name[0] = "Omega_b"; fc.value[0] = 0.04;
...
```

2. pass the structure `file_content` to the function `input_init(...)` that interprets the input parameters, set all other parameters to default values, and fills the input parameter fields of each structure.

These two steps are performed inside:

```
input_init_from_arguments(argc,argv,&pr,&ba,&th,&pt,&tr,&pm
    ,&sp,&nl,&le,&op,errmsg)
input_init(&fc,&pr,&ba,&th,&pt,&tr,&pm,&sp,&nl,&le,&op,
    errmsg)
```

But when CLASS is called by another code, the other code (the CLASS wrapper) fills a `file_content` structure `fc`, and calls directly `input_init`

Input

Essential input parameters controlling the output (see details in `explanatory.ini`):

```
modes = s,t
ic = ad, cdi, bi, nid, niv
lensing = yes
nonlinear = halofit
output = tCl, pCl, lCl, mPk, mTk, vTk, nCl, sCl

l_max_scalars=2500
l_max_tensors=500
l_max_lss = 1000
P_k_max_h/Mpc = 0.2
#P_k_max_1/Mpc =
z_pk = 0 #or 1,2,10

root = output/test_

headers = yes, no
format = class, camb
write parameters = yes, no
write warnings = no

verbose_xxx = 1
```

The output module

source/output.c

The output module

Called in the last place by `main/class.c` to write all requested output in files. Only writing, no physics, no manipulation of tables stored in other modules. Uses external interpolation functions of other modules, e.g.

```
spectra_cl_at_l(...);
lensing_cl_at_l(...);
spectra_pk_at_z(...);
spectra_pk_nl_at_z(...);
...
```

If CLASS embedded in another code, same information is obtained by directly calling such functions.

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested
- `test_tk.dat` density and/or velocity transfer functions

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested
- `test_tk.dat` density and/or velocity transfer functions
- `test_tk_ad.dat`, `test_tk_cdi.dat`, `test_tk_ad_cdi.dat` etc. when different i.c. requested

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested
- `test_tk.dat` density and/or velocity transfer functions
- `test_tk_ad.dat`, `test_tk_cdi.dat`, `test_tk_ad_cdi.dat` etc. when different i.c. requested
- if `pk` or `tk` requested at different redhsift, several files, with extra suffix `_z0`, `_z1`, etc.

The output module

Following files created (or not) automatically (here we assume that `root=test_`):

- `test_cl.dat` total unlensed C_l's
- `test_cl_lensed.dat` total lensed C_l's
- `test_cls.dat` scalar C_l's when two modes
- `test_clt.dat` tensor C_l's when two modes
- `test_pk.dat` matter power spectrum
- `test_pk_nl.dat` non-linear matter power spectrum
- `test_cl_ad.dat`, `test_cl_cdi.dat`, `test_cl_ad_cdi.dat` etc. when different i.c. requested
- `test_pk_ad.dat`, `test_pk_cdi.dat`, `test_pk_ad_cdi.dat` etc. when different i.c. requested
- `test_tk.dat` density and/or velocity transfer functions
- `test_tk_ad.dat`, `test_tk_cdi.dat`, `test_tk_ad_cdi.dat` etc. when different i.c. requested
- if `pk` or `tk` requested at different redhsift, several files, with extra suffix `_z0`, `_z1`, etc.

Number of columns in these files can vary a lot depending on input parameters. Always indicated in the header.

Plotting

- Whatever software is fine: `gnuplot`, `python`, `matlab`, `IDL`...

Plotting

- Whatever software is fine: `gnuplot`, `python`, `matlab`, `IDL`...
- We provide scripts in `python` (the Class Plotting Unit `class/CPU`) and `matlab` (`Geneva_Tools/Auxiliary codes/plot_CLASS_output.m`)

Plotting

E.g. to plot total E-mode polarisation in `output/test_cl_lensed.dat`:

```
# dimensionless total [l(l+1)/2pi] C_l's
# for l=2 to 3000, i.e. number of multipoles equal to 2999
#
# -> if you prefer output in CAMB/HealPix/LensPix units/
    order, set 'format' to 'camb' in input file
# -> if you don't want to see such a header, set 'headers'
    to 'no' in input file
#
#  l TT     EE       TE       BB      phiphi      Tphi     Ephi
```

you could do:

```
python CPU -h
python CPU -colnum 3 -x output/test_cl.dat
```

or:

```
plot_CLASS_output('output/test.dat','EE')
```

Exercises I

All these exercises consist in running CLASS with the correct set of input parameters (cosmological parameters are unimportant), and plotting its different outputs with gnuplot, or CPU, or plot_CLASS_output.m (or with your own favorite software).

Ia

Check the difference between the lensed and unlensed C_l^{TT} of scalars, to see effect of smoothing of the peak contrast, and extra damping.

Ib

Check the difference between the lensed and unlensed C_l^{BB} of tensors, to see that B modes are dominated by lensing at least on small scales. Use $r = 0.2$ like in BICEP results!

Exercises I

Ic

Check the difference between the unlensed C_l^{TT} of scalar modes for adiabatic and CDM isocurvature (CDI) initial conditions (with index $n_{cdi} = 1$) , to check that peaks are suppressed in amplitude and shifted in scale. Do the same with NID isocurvature modes (with index $n_{nid} = 1$) to check that the suppression in amplitude is less pronounced and the phase of NID and CDI are different.

Id

Check the difference between the linear and non-linear matter power spectrum at $z = 0$ and $z = 2$, to see that at low redshift non-linear corrections are present on larger scales.