Advanced usage of Monte Python

Benjamin Audren

École Polytechnique Fédérale de Lausanne

15/05/2014

Outline

Command line arguments

How to find the help

`python montepython/MontePython.py -h`, and
`python montepython/MontePython.py run -h`,
`python montepython/MontePython.py info -h`.

python montepython/MontePython.py run ...

Compulsory ones

- `-o`: output folder
- `-p`: input parameter file

python montepython/MontePython.py run ...

Compulsory ones

- `-o`: output folder
- `-p`: input parameter file

For Metropolis Hastings

- `-N`: number of steps **asked**.
- `-c`: covariance matrix (`.covmat` file)
- `-b`: best-fit file (`.bestfit` file)
- `-j` jumping method (`fast` for Cholesky)
- `-f` jumping factor (default `2.4`)

python montepython/MontePython.py run ...

Compulsory ones

- `-o`: output folder
- `-p`: input parameter file

For Metropolis Hastings

- `-N`: number of steps **asked**.
- `-c`: covariance matrix (`.covmat` file)
- `-b`: best-fit file (`.bestfit` file)
- `-j` jumping method (`fast` for Cholesky)
- `-f` jumping factor (default `2.4`)

Changing methods

- `-m` sampling method (`MH, NS, CH, IS`)

python montepython/MontePython.py info ...

Main argument

- **no selector**: `folder` or `list of files`

Polishing the output

- `--bins` number of bins to compute histogram
- `--no-mean` not showing mean likelihood
- `--extra` plot triangle plot for subset of params
- `--noplot` only text files
- `--all` output all subplots
- `--ext` choose the format of output (`png` or `pdf`)
- `--fontsize` and `--ticksize` adjust font

python montepython/MontePython.py info ...

Comparison

- `--comp` another folder
- `--plot-2d` set to `always` to have 2D comparison
- `--alpha` choose the transparency of 2nd posterior

Chains format

What is in there?

- Same format than CosmoMC chains
- name automatically generated `date_N__number.txt`
- `Multiplicity, -LogLkl, param1, param2, ...`

```
2   15.1909     2.242269e+00    6.982825e-01    7.275432e-01
1   15.8213     2.271929e+00    6.928746e-01    7.262034e-01
1   15.9302     2.232572e+00    6.920071e-01    7.269063e-01
1   16.4508     2.279289e+00    6.880627e-01    7.253825e-01
2   16.5249     2.256672e+00    6.875273e-01    7.257858e-01
2   16.2295     2.261342e+00    6.896998e-01    7.259885e-01
1   16.4418     2.250112e+00    6.881289e-01    7.260079e-01
2   16.9797     2.259634e+00    6.843527e-01    7.252778e-01
1   17.8757     2.209373e+00    6.785898e-01    7.255448e-01
2   16.0907     2.184355e+00    6.907565e-01    7.277476e-01
1   14.1885     2.174913e+00    7.089811e-01    7.302632e-01
1   13.3781     2.178098e+00    7.232123e-01    7.318917e-01
2   13.1839     2.227072e+00    7.354540e-01    7.323636e-01
2   13.2052     2.251143e+00    7.380229e-01    7.322084e-01
```

But I don't know which parameters I asked?

But I don't know which parameters I asked?

Look at the log.param:

```
data.experiments=['hst', 'timedelay']

# Cosmological parameters list
data.parameters['omega_b']    = [2.249,   1.8, 3, 0.016, 0.01, 'cosmo']
data.parameters['h']          = [0.703,   0.6,0.8, 0.0065,1,   'cosmo']

# Derived parameter list
data.parameters['Omega_Lambda'] = [0,        -1, -1, 0,1,  'derived']
```

Having mcmc parameters that are not known to CLASS

How to do it ?

- Need to modify the source code (**bad**)
- but it is **not so complicated**

Having mcmc parameters that are not known to CLASS

How to do it ?

- Need to modify the source code (**bad**)
- but it is **not so complicated**

Why doing it ?

Having mcmc parameters that are not known to CLASS

How to do it ?

- Need to modify the source code (**bad**)
- but it is **not so complicated**

Why doing it ?

- Using **CosmoMC** parameters (like `ln10^{10}A_s`)

Having mcmc parameters that are not known to CLASS

How to do it ?

- Need to modify the source code (**bad**)
- but it is **not so complicated**

Why doing it ?

- Using **CosmoMC** parameters (like `ln10^{10}A_s`)
- Using parameter **combinations**

Having mcmc parameters that are not known to CLASS

How to do it ?

- Need to modify the source code (**bad**)
- but it is **not so complicated**

Why doing it ?

- Using **CosmoMC** parameters (like `ln10^{10}A_s`)
- Using parameter **combinations**
- Dealing with **complicated parameters** in CLASS

How to do it?

```
    def update_cosmo_arguments(self):


        for elem in self.get_mcmc_parameters(['cosmo']):
            # Fill in the dictionnary with the current value of parameters
            self.cosmo_arguments[elem] = \
                self.mcmc_parameters[elem]['current'] *\
                self.mcmc_parameters[elem]['scale']
```

How to do it?

```
    def update_cosmo_arguments(self):

        for elem in self.get_mcmc_parameters(['cosmo']):
            # Fill in the dictionnary with the current value of parameters
            self.cosmo_arguments[elem] = \
                self.mcmc_parameters[elem]['current'] *\
                self.mcmc_parameters[elem]['scale']

        for elem in self.get_mcmc_parameters(['cosmo']):
```

How to do it?

```
    def update_cosmo_arguments(self):


        for elem in self.get_mcmc_parameters(['cosmo']):
            # Fill in the dictionnary with the current value of parameters
            self.cosmo_arguments[elem] = \
                self.mcmc_parameters[elem]['current'] *\
                self.mcmc_parameters[elem]['scale']


        for elem in self.get_mcmc_parameters(['cosmo']):


            if elem == 'ln10^{10}A_s':
                self.cosmo_arguments['A_s'] = math.exp(
                    self.cosmo_arguments[elem]) / 1.e10
                del self.cosmo_arguments[elem]
```

How to do it?

```
    def update_cosmo_arguments(self):


        for elem in self.get_mcmc_parameters(['cosmo']):
            # Fill in the dictionnary with the current value of parameters
            self.cosmo_arguments[elem] = \
                self.mcmc_parameters[elem]['current'] *\
                self.mcmc_parameters[elem]['scale']


        for elem in self.get_mcmc_parameters(['cosmo']):


            if elem == 'exp_m_2_tau_As':
                tau_reio = self.cosmo_arguments['tau_reio']
                self.cosmo_arguments['A_s'] = self.cosmo_arguments[elem] * \
                    math.exp(2.*tau_reio)
                del self.cosmo_arguments[elem]
```

How to do it?

```
    def update_cosmo_arguments(self):


        for elem in self.get_mcmc_parameters(['cosmo']):
            # Fill in the dictionnary with the current value of parameters
            self.cosmo_arguments[elem] = \
                self.mcmc_parameters[elem]['current'] *\
                self.mcmc_parameters[elem]['scale']


        for elem in self.get_mcmc_parameters(['cosmo']):


            if elem == 'M_tot':
                self.cosmo_arguments['m_ncdm'] = self.cosmo_arguments['M_tot']/3.
                del self.cosmo_arguments[elem]
```

Planck likelihood

For those of you who installed Planck

If you run `base.param` like this:

```
python montepython/MontePython.py -p base.param -o chains/planck -N 1000
```

the acceptance rate will be **dramatically** low. You **have to** use the given covariance matrix called `base.covmat`, like this:

```
python montepython/MontePython.py -p base.param -o chains/planck \
-N 1000 -c covmat/base.covmat -f 1.5 -j fast
```

Cholesky Decomposition with equations

Idea

Cosmological parameters are slow to update (CLASS), but **nuisance** parameters can be fast.
If **varied together**, this distinction is lost.
But, there are **correlations between them!**

Proposition

Instead of varying all the parameters at each step, we vary **both fast and slow** some of the time, and **only the fast one** the rest of the time.

Cholesky Decomposition with equations

Decomposition of the Proposal density

$C = \mathbf{L}\mathbf{L}^T$ with $\mathbf{L}$ a lower triangular matrix.
We define the new parameters $x' = \mathbf{L}^{-1}x$.

Blocks of parameters

$$\begin{pmatrix} S_1 \\ S_2 \\ F_1 \\ F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ * & * & * & * & 0 \\ * & * & * & * & * \end{pmatrix} \begin{pmatrix} S'_1 \\ S'_2 \\ F'_1 \\ F'_2 \\ F'_3 \end{pmatrix}$$

Cholesky Decomposition with equations

Decomposition of the Proposal density

$C = \mathbf{L}\mathbf{L}^T$ with $\mathbf{L}$ a lower triangular matrix.
We define the new parameters $x' = \mathbf{L}^{-1}x$.

Blocks of parameters Slow and Fast

$$\begin{pmatrix} \Delta S_1 \\ \Delta S_2 \\ \Delta F_1 \\ \Delta F_2 \\ \Delta F_3 \end{pmatrix} = \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ * & * & * & * & 0 \\ * & * & * & * & * \end{pmatrix} \begin{pmatrix} \Delta S'_1 \\ \Delta S'_2 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Cholesky Decomposition with equations

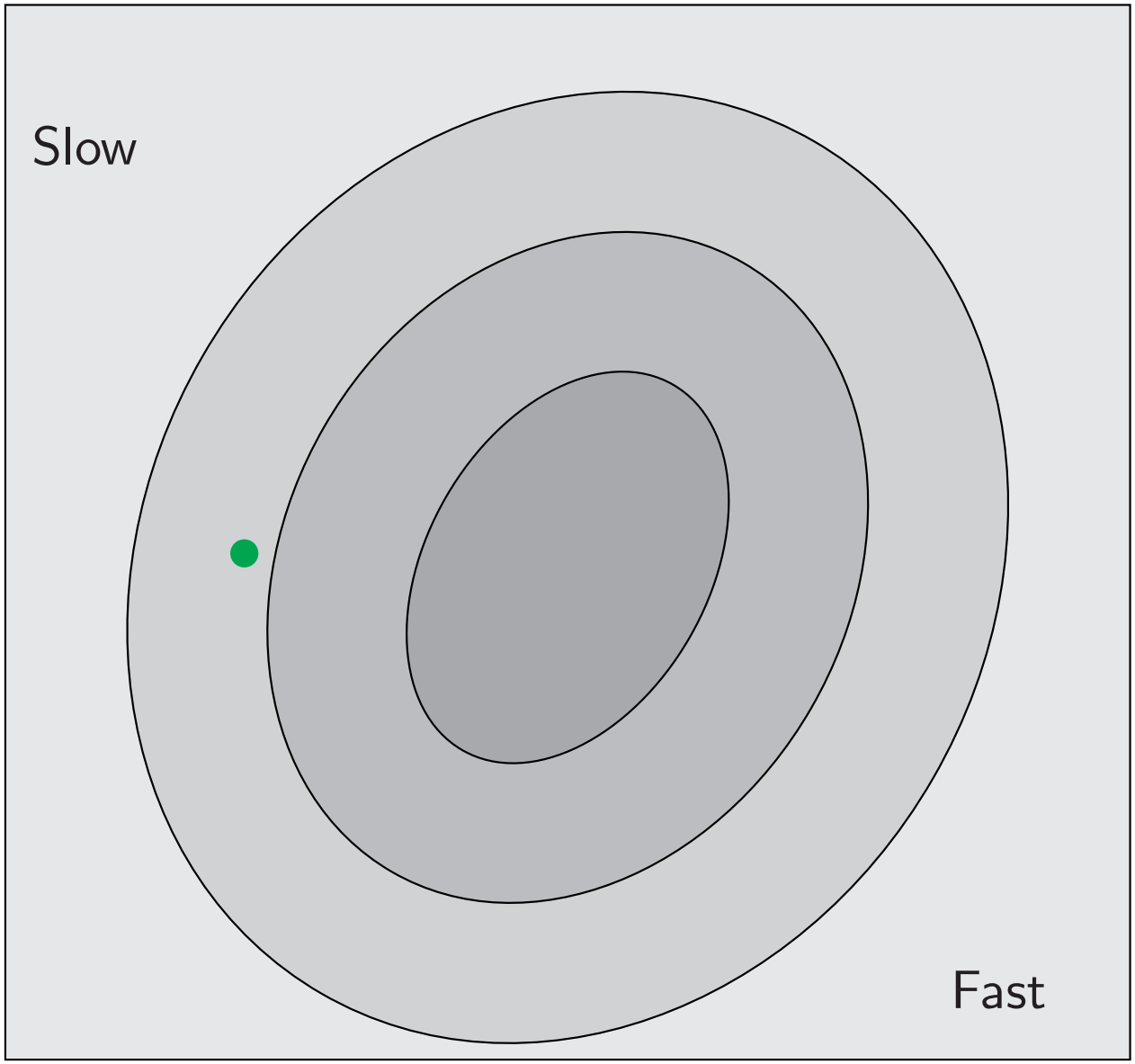
Decomposition of the Proposal density

$C = \mathbf{L}\mathbf{L}^T$ with $\mathbf{L}$ a lower triangular matrix.
We define the new parameters $x' = \mathbf{L}^{-1}x$.

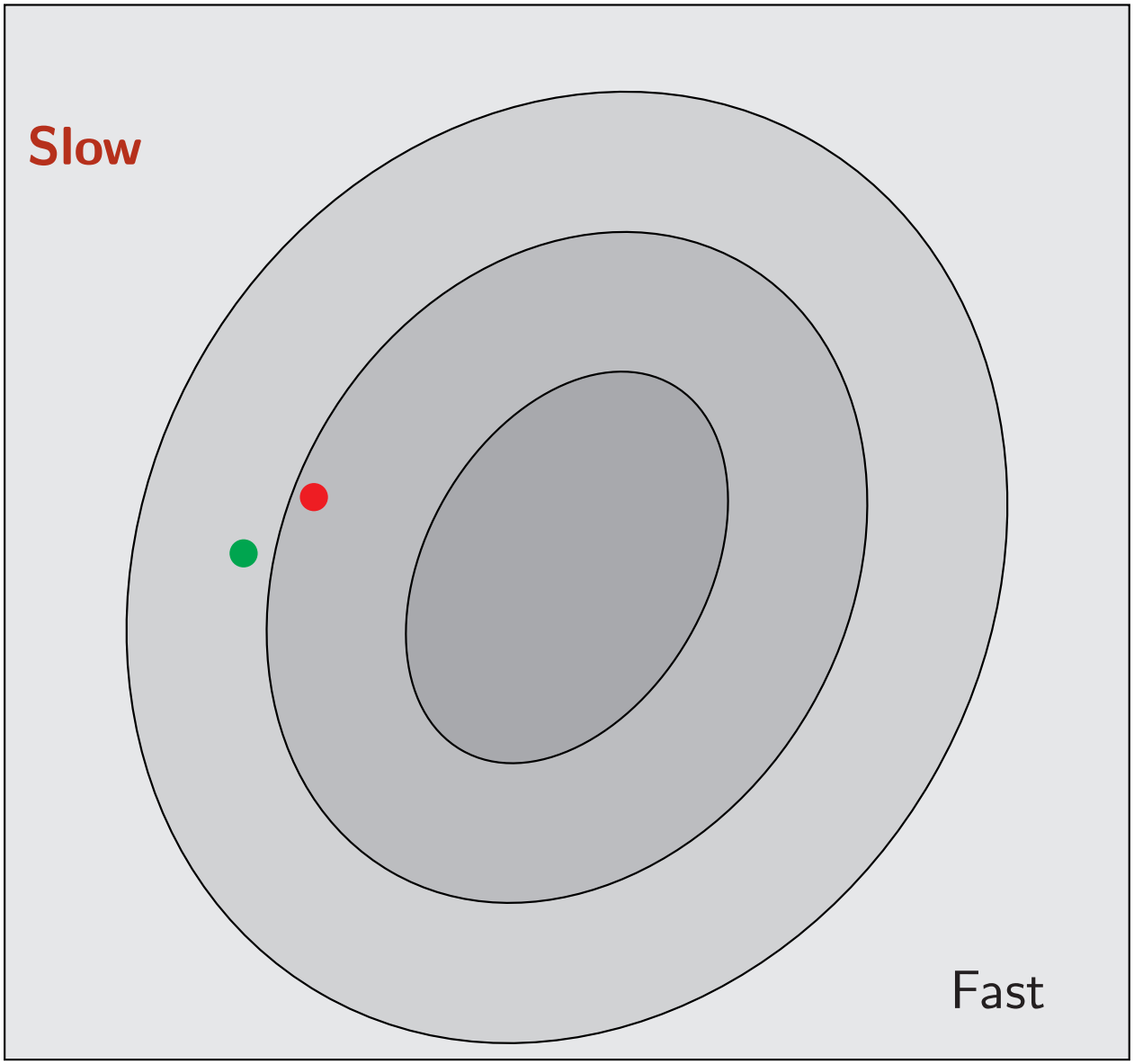
Blocks of parameters only Fast

$$\begin{pmatrix} 0 \\ 0 \\ \Delta F_1 \\ \Delta F_2 \\ \Delta F_3 \end{pmatrix} = \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ * & * & * & * & 0 \\ * & * & * & * & * \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ \Delta F_1' \\ \Delta F_2' \\ \Delta F_3' \end{pmatrix}$$

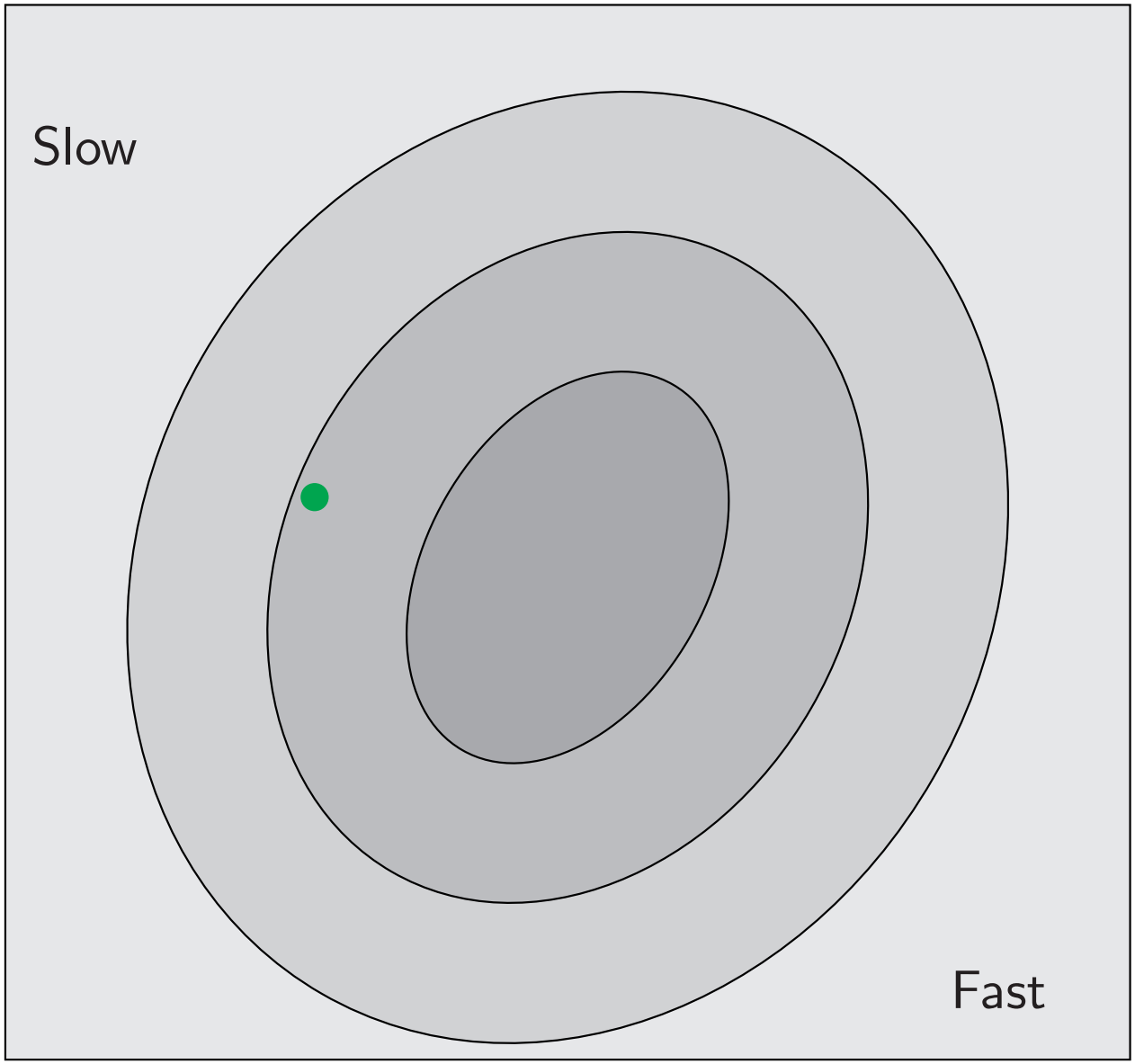
Cholesky Decomposition and Over-sampling



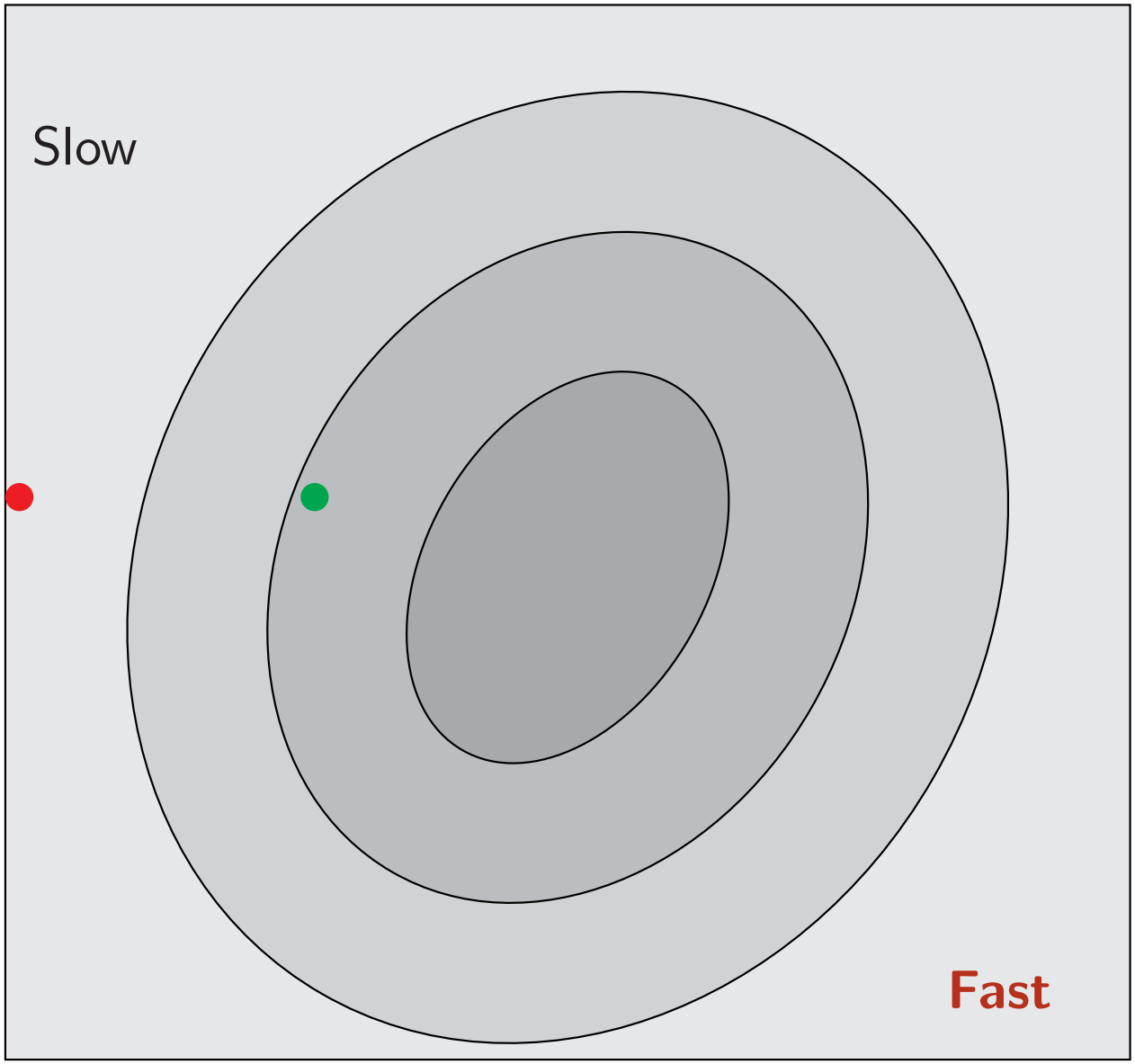
Cholesky Decomposition and Over-sampling



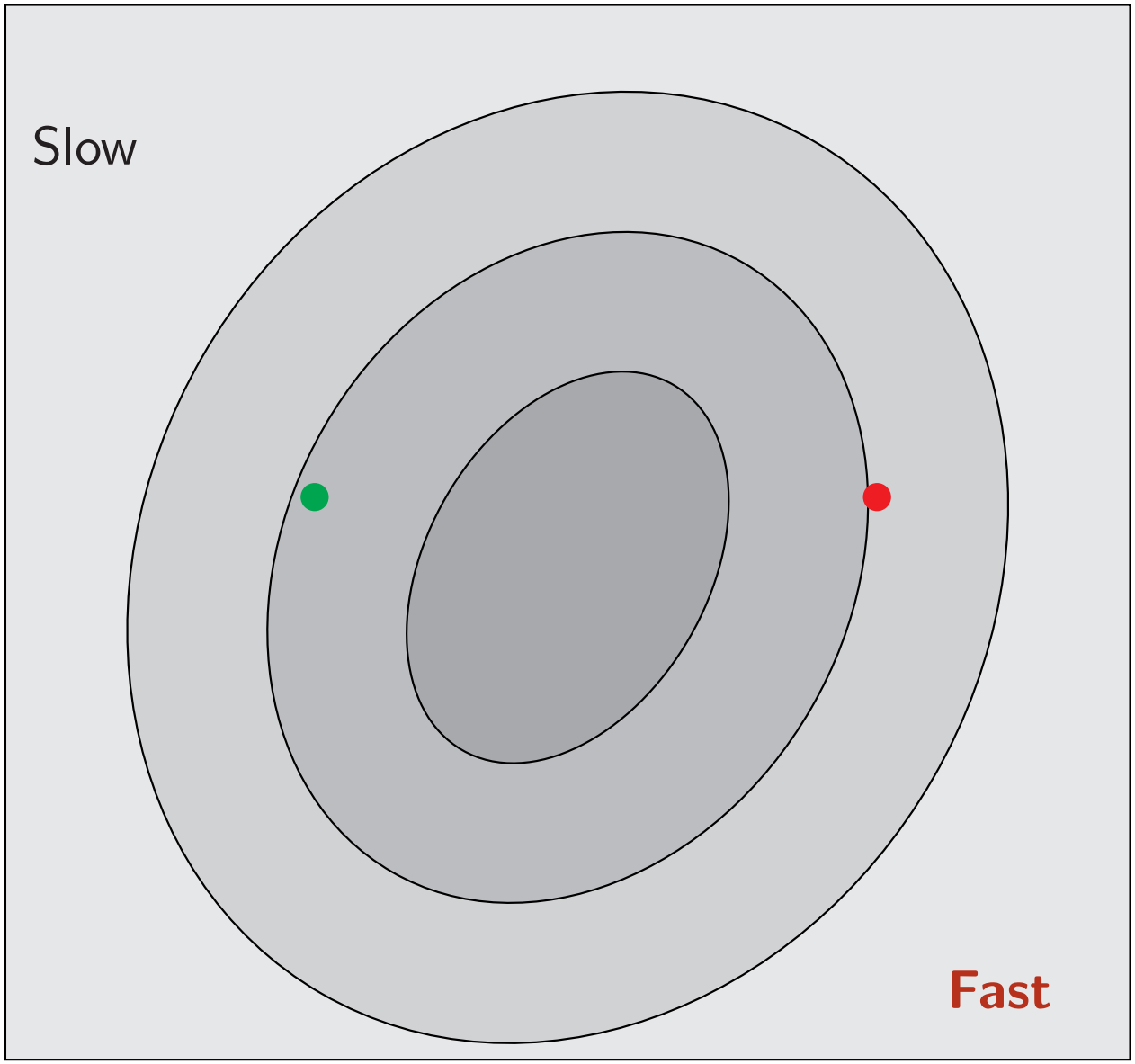
Cholesky Decomposition and Over-sampling



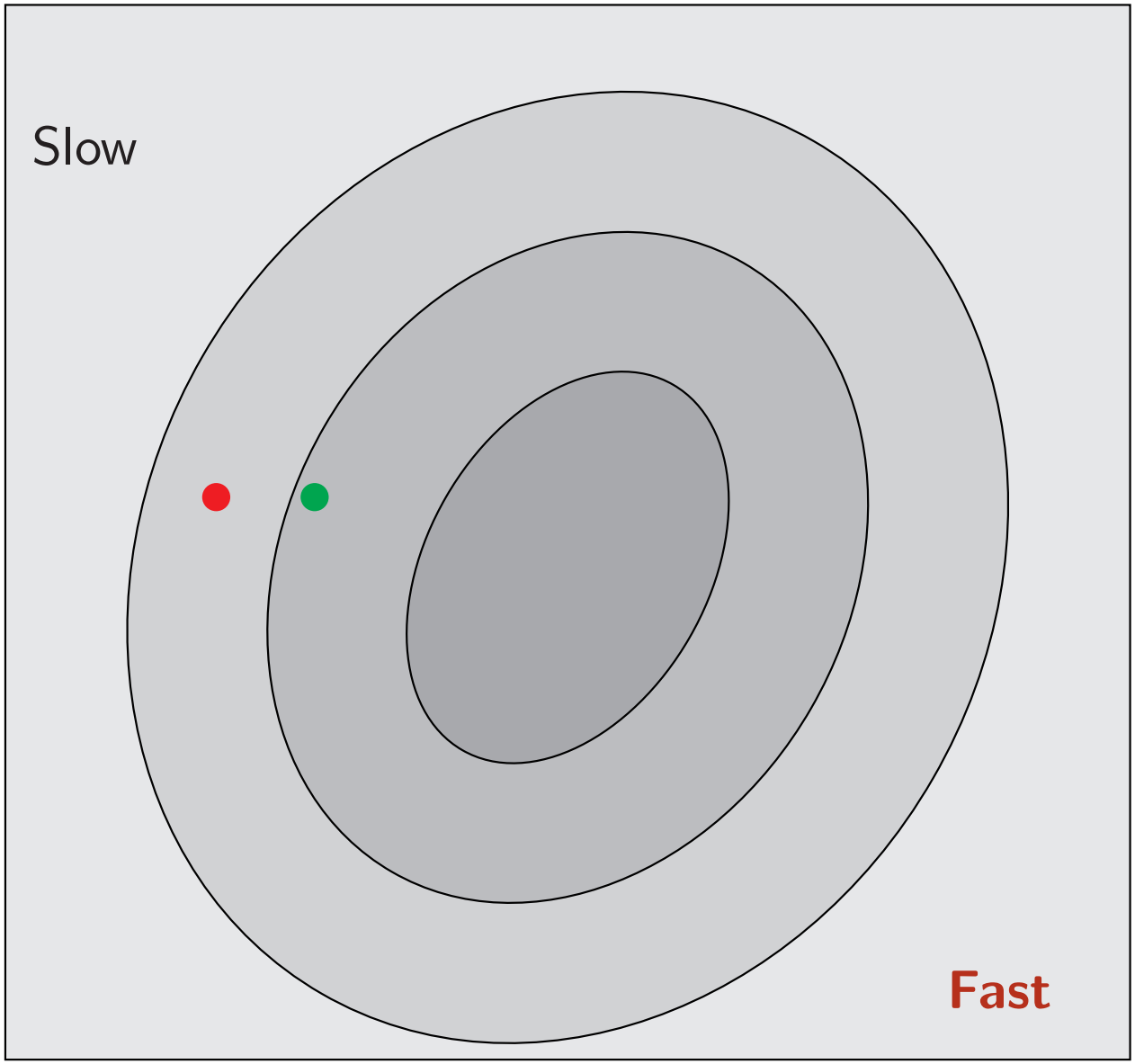
Cholesky Decomposition and Over-sampling



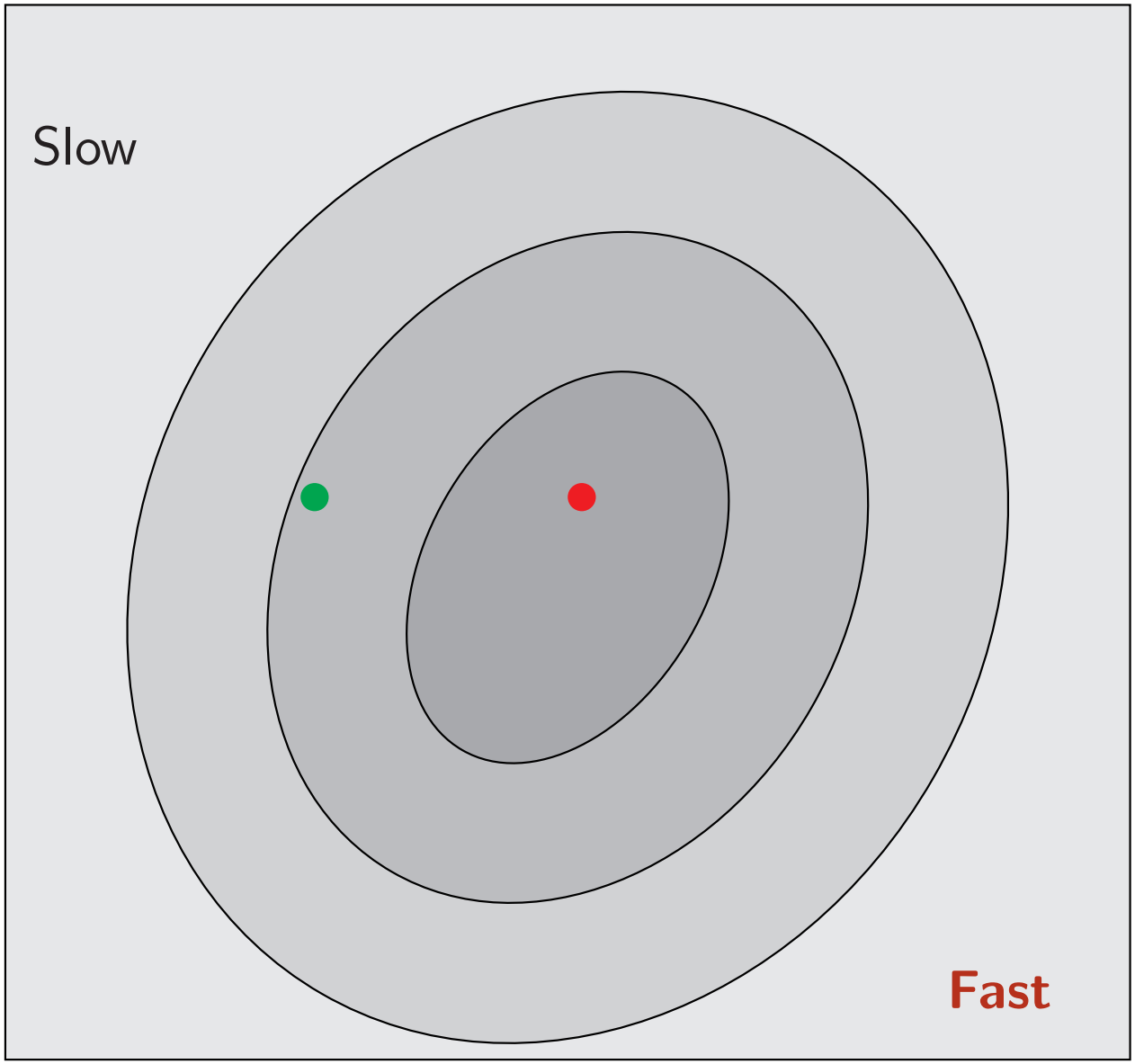
Cholesky Decomposition and Over-sampling



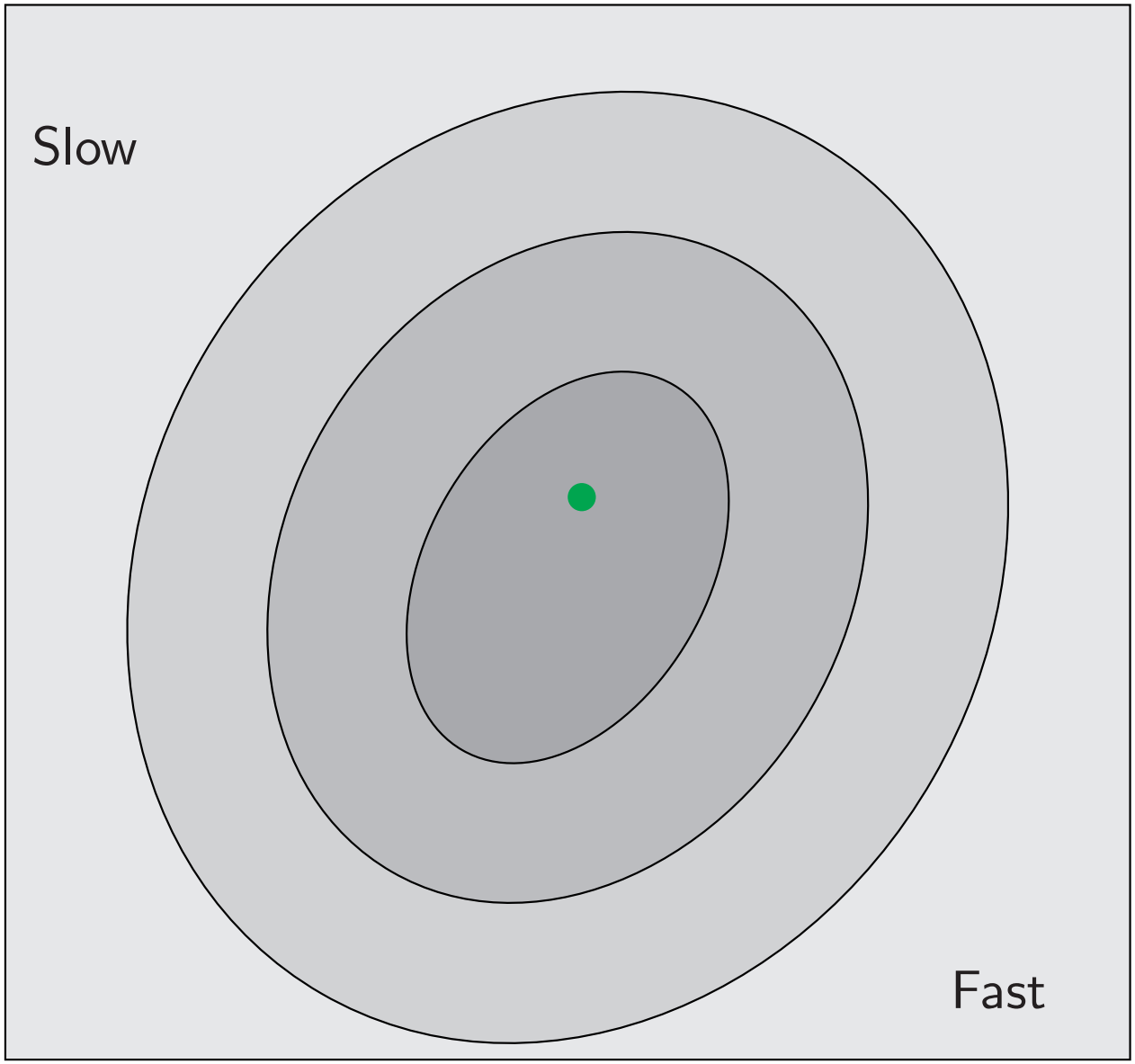
Cholesky Decomposition and Over-sampling



Cholesky Decomposition and Over-sampling



Cholesky Decomposition and Over-sampling



Outline

Wrapper around CLASS

Realisation

We wrote a **class** called **Class** wrapping CLASS !

```
cdef class Class:
```

Wrapper around CLASS

Sequence

CLASS **functions**, **wrapper functions**, **Monte Python functions**

- Set the parameters from the **data** dictionary, **data.cosmo_arguments** with the method **set**, which is equivalent to writing a **something.ini**

Wrapper around CLASS

Sequence

CLASS **functions**, **wrapper functions**, **Monte Python functions**

- Set the parameters from the **data** dictionary, **data.cosmo_arguments** with the method **set**, which is equivalent to writing a **something.ini**
- Run all **_init()** functions with **compute()**

Wrapper around CLASS

Sequence

CLASS **functions**, **wrapper functions**, **Monte Python functions**

- Set the parameters from the **data** dictionary, **data.cosmo_arguments** with the method **set**, which is equivalent to writing a **something.ini**
- Run all **_init()** functions with **compute()**
- Recover Cl **lensed_cl()**, Pk **pk()**, etc. . . equivalents to the **output_pk_at_k_and_z**

Wrapper around CLASS

Sequence

CLASS **functions**, **wrapper functions**, **Monte Python functions**

- Set the parameters from the **data** dictionary, **data.cosmo_arguments** with the method **set**, which is equivalent to writing a **something.ini**
- Run all **_init()** functions with **compute()**
- Recover Cl **lensed_cl()**, Pk **pk()**, etc. . . equivalents to the **output_pk_at_k_and_z**
- Free the structutes with **struct_cleanup**, which calls all the **_free()** functions.

Wrapper around CLASS

Sequence

CLASS **functions**, **wrapper functions**, **Monte Python functions**

- Set the parameters from the **data** dictionary, **data.cosmo_arguments** with the method **set**, which is equivalent to writing a **something.ini**
- Run all **_init()** functions with **compute()**
- Recover Cl **lensed_cl()**, Pk **pk()**, etc. . . equivalents to the **output_pk_at_k_and_z**
- Free the structutes with **struct_cleanup**, which calls all the **_free()** functions.

opt Clean the set of parameters to run with something completely different: **empty()**

Wrapper around CLASS: an example Python script

```
from classy import Class

# Define a cosmological scenario (CLASS default otherwise)
params = {'omega_b': 0.02, 'h': 0.7, 'output': 'mPk'}

# Create a Class instance
cosmo = Class()

# Set the instance to the cosmology
cosmo.set(params)

# Run the _init methods
cosmo.compute()

# Do something with the pk
pk = cosmo.pk(0, 0.1)

# Clean
cosmo.struct_cleanup(); cosmo.empty()
```

BICEP2 from CLASS and Monte Python

Open the IPython Notebook now

Compute details

```
    def compute(self, lvl=["lensing"]):
```

```
        if "input" in lvl:
            ierr = input_init(
                &self.fc,
                &self.pr,
                &self.ba,
                &self.th,
                &self.pt,
                &self.tr,
                &self.pm,
                &self.sp,
                &self.nl,
                &self.le,
                &self.op,
                errmsg)
            if ierr == _FAILURE_:
                raise CosmoSevereError(errmsg)
            self.ncp.add("input")
```

```
            problem_flag = False
            problematic_parameters = []
            for i in range(self.fc.size):
                if self.fc.read[i] == _FALSE_:
                    problem_flag = True
                    problematic_parameters.append(self.fc.name[i])
            if problem_flag:
                raise CosmoSevereError(
                    "Class did not read input parameter(s): %s\n" % ', '.join(
                    problematic_parameters))
```

Compute details

```
        if "background" in lvl:
            if background_init(&(self.pr),&(self.ba)) == _FAILURE_:
                self.struct_cleanup()
                raise CosmoComputationError(self.ba.error_message)
            self.ncp.add("background")

        if "thermodynamics" in lvl:
            if thermodynamics_init(&(self.pr),&(self.ba),&(self.th)) == _FAILURE_:
                self.struct_cleanup()
                raise CosmoComputationError(self.th.error_message)
            self.ncp.add("thermodynamics")
```

...

```
        if "lensing" in lvl:
            if lensing_init(&(self.pr),&(self.pt),&(self.sp),&(self.nl),&(self.le)) ==
                 _FAILURE_:
                self.struct_cleanup()
                raise CosmoComputationError(self.le.error_message)
            self.ncp.add("lensing")

        self.ready = True
```

What happens when a likelihood does not need the C_ℓ ?

What happens when a likelihood does not need the C_ℓ ?

Nothing !

- Each likelihood **defines its requirements** to the cosmological code.
- If it requires **no C_ℓ, or P_k**, the `output` variable in explanatory.ini is set to nothing.
- Background functions will be **lightning fast** (see exercises)

for instance, in a likelihood/__init__.py file:

```
self.need_cosmo_arguments(data, 'output': 'mPk')
```

What should I modify in classy if I modified CLASS ?

nothing !

If you respected the same structure as CLASS, then you are done, and you can start using this inside Monte Python.

What should I modify in classy if I modified CLASS ?

nothing !

If you respected the same structure as CLASS, then you are done, and you can start using this inside Monte Python.

For instance

You **solved exercise IIb** of Monday, implemented an **extra fluid**. You have two **new** CLASS **parameters**: `Omega_efld` and `w0_efld`. You can run Monte Python with these parameters, fixed or varying !

What should I modify in classy if I modified CLASS ?

nothing !

If you respected the same structure as CLASS, then you are done, and you can start using this inside Monte Python.

For instance

You **solved exercise IIb** of Monday, implemented an **extra fluid**. You have two **new** CLASS **parameters**: `Omega_efld` and `w0_efld`. You can run Monte Python with these parameters, fixed or varying !

Reminder

You will need to compile the wrapper for your new version, though!

Outline

What are they?

Idea

- Used to make forecast for future experiments (Euclid,. . .)

What are they?

Idea

- Used to make forecast for future experiments (Euclid,. . .)
- What is interesting is the expected **sensitivity** to data

What are they?

Idea

- Used to make forecast for future experiments (Euclid,. . .)
- What is interesting is the expected **sensitivity** to data
- We can **fix a fiducial cosmology model**, which we use, **given our knowledge of the experiment**, to **simulate an observation**.

What are they?

Idea

- Used to make forecast for future experiments (Euclid,. . .)
- What is interesting is the expected **sensitivity** to data
- We can **fix a fiducial cosmology model**, which we use, **given our knowledge of the experiment**, to **simulate an observation**.
- We then run a standard MCMC exploration

Outline

Design reminders

4 rules, for MyLikelihood

- `Mylikelihood` must be a folder in `montepython/likelihoods/`

Design reminders

4 rules, for MyLikelihood

- `Mylikelihood` must be a folder in `montepython/likelihoods/`
- `Mylikelihood` must contain two files, `__init__.py` and `Mylikelihood.data`

Design reminders

4 rules, for MyLikelihood

- `Mylikelihood` must be a folder in `montepython/likelihoods/`
- `Mylikelihood` must contain two files, `__init__.py` and `Mylikelihood.data`
- `__init__.py` must define a `class` called `Mylikelihood`, inheriting from `Likelihood`

Design reminders

4 rules, for MyLikelihood

- `Mylikelihood` must be a folder in `montepython/likelihoods/`
- `Mylikelihood` must contain two files, `__init__.py` and `Mylikelihood.data`
- `__init__.py` must define a `class` called `Mylikelihood`, inheriting from `Likelihood`
- `class Mylikelihood` must define a function called `loglkl` that returns the **log likelihood**

Likelihood file

```
# import the python package of the BICEP2 collaboration
import bicep_util as bu


class bicep2(Likelihood):
```

```
    def __init__(self, path, data, command_line):
```

```
        # Require tensor modes from Class
        arguments = {
            'output': 'tCl pCl lCl',
            'lensing': 'yes',
            'modes': 's, t',
            'l_max_scalars': self.l_max,
            'l_max_tensors': self.l_max,}
        self.need_cosmo_arguments(data, arguments)
```

Likelihood file

```
# import the python package of the BICEP2 collaboration
import bicep_util as bu


class bicep2(Likelihood):

    def loglkl(self, cosmo, data):

        dict_Cls = self.get_cl(cosmo, self.l_max)


        # Convert the dict to the same format expected by BICEP
        # that is:
        # 0: TT
        # 1: TE
        # 2: EE
        # 3: BB
        # 6: ET, and the rest to 0 (must be an array of width 9)
```

Likelihood file

```
# import the python package of the BICEP2 collaboration
import bicep_util as bu


class bicep2(Likelihood):
```

...

```
        # Get the expectation value of the data considering this
            theoretical
        # model
        expectation_value = bu.calc_expvals(
            ell, cosmo_Cls,
            self.bpwf_l, self.bpwf_Cs_l)
```

Likelihood file

```
# import the python package of the BICEP2 collaboration
import bicep_util as bu


class bicep2(Likelihood):
```

...

```
        # Add the noise
        self.C_l += self.N_l

        # Actually compute the likelihood
        loglkl = bu.evaluateLikelihood(
            self.C_l, self.C_l_hat, self.C_fl, self.M_inv)

        return loglkl
```

List of existing likelihoods in Monte Python

ls montepython/likelihoods

- `Planck_highl`, `Planck_lowl`
- `Planck_actstp`, `Planck_lensing`
- `Planck_SZ`, `lowlike`
- `clik_wmap_full` and `lowl`
- `bicep`, `bicep2`, `acbar`
- `boomerang`, `cbi`, `quad`
- `spt` and `spt_2500`

- `WiggleZ`, `sdss_lrgDR4`
- `euclid_lensing`, `euclid_pk`
- `sn`, `hst`, `timedelay`
- `bao`, `bao_boss`, ...

Outline

Exercise: Implementing a new likelihood

Simplest example

I) HST-like likelihood

An experiment called `hubble_2013` measured

$$h = 0.712 \pm 0.012$$

Create this likelihood and use it in a run.

II) Use the classy wrapper

to plot Cl_ℓ^{BB} with Planck bestfit and $r = 0.2$ (think about $n_t = 0$ and pivot scale)

III) Use the BICEP2 likelihood

At your own risk...

Hints for Exercice I

- create all the needed files/folder first
- define the data associated to the measurement in the `.data` file
- inherit from `Likelihood`
- get inspiration from `hst`