The PIPELET software (1.0)

Maude Le Jeune, Marc Betoule

November, 26th, 2010

Context

How it works
- Building a pipeline
- Writing segment scripts
- Running a pipeline
- Browsing a pipeline

A CMB demo pipeline
- Problematic
- A pipelet solution
- Zooming the code
- Browsing the result
- Deployment

Getting started

Context

How it works
Building a pipeline
Writing segment scripts
Running a pipeline
Browsing a pipeline

A CMB demo pipeline
Problematic
A pipelet solution
Zooming the code
Browsing the result
Deployment

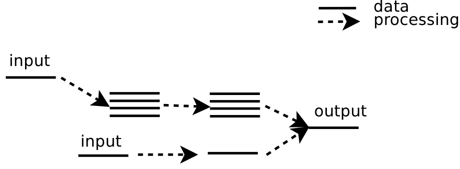
Getting started

Context and needs

Usually in scientific data processing:

- Big data sets and/or big CPU time
- Optimal parameters unknown
- Complex processing (multiple interdependant steps)

$\rightarrow$ Computation **and development** cost a lot.



The **PIPELET** framework helps with these 3 points:

- Native parallelisation and CPU time saving
  (recompute only the needed parts)
- Offer comparison facilities
- Take care of traceability

Usual issues

1. The processing is cut in 2/3 steps, intermediate products are saved on disk with approximate filenames

```
map-nside2048-ps5sigma-masksmall
map-nside2048-ps5sigma-masksmall-nodip
map-nside2048-ps5sigma-masksmall-nodip-2
```

2. In the best case: low level routines are documented and can be reused by someone else, but pipelines are always trashed !
3. A prototype pipeline is built using an high level programming language, and smaller dimensions. What's next ?
   - use another programming language to perform real processing
   - use interfacing
   - what about code parallelisation ?
   - what about portability ? (smp machines, clusters, ...)

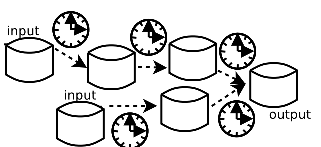
Pipe scheme and intermediate products

1. *The processing is cut in 2/3 steps, intermediate products are saved on disk with approximate filenames*

▷ Cut the whole processing into segments

▷ Save intermediate products on disk

▷ Use an unique indentifier wrt code, parameters and I/Os.



⟹ Filenames are provided by the PIPELET engine.
- key inputs (not the key itself) readable from the web interface.

⟹ The pipe scheme is defined by user (any directed acyclic graph allowed)
- small number of segments for disk saving
- right number of segments for readability

Permanence and collaborative work

2. *In the best case: low level routines are documented and can be reused by someone else, but pipelines are always trashed !*

▷ Pipeline scheme written and displayed using graphviz dot language.

```
"""
1->2->4;
3->4;
"""
```

→

▷ Segments can be documented using Python docstring syntax.

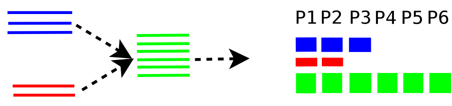
$\Longrightarrow$ Non trivial dependency scheme are easy to read.

▷ Collaborative work eased by a web interface and *code repositories*.

$\Longrightarrow$ Results can be accessed by different users with a min amount of indications (tags).

Prototyping and parallelization

3. *A prototype pipeline is built using an high level programming language, and smaller dimensions. What's next ?*



▷ Native parallelisation scheme applied on data to process (**tasks**).

⟹ Parallelisation at the highest level. No need to learn OpenMP/MPI.

▷ **Workers** empty the **task queue** in different modes (sequential for debugging, using process on smp machine, batch mode on clusters).

⟹ Scalability and portability offered by the different running modes (**launchers**).

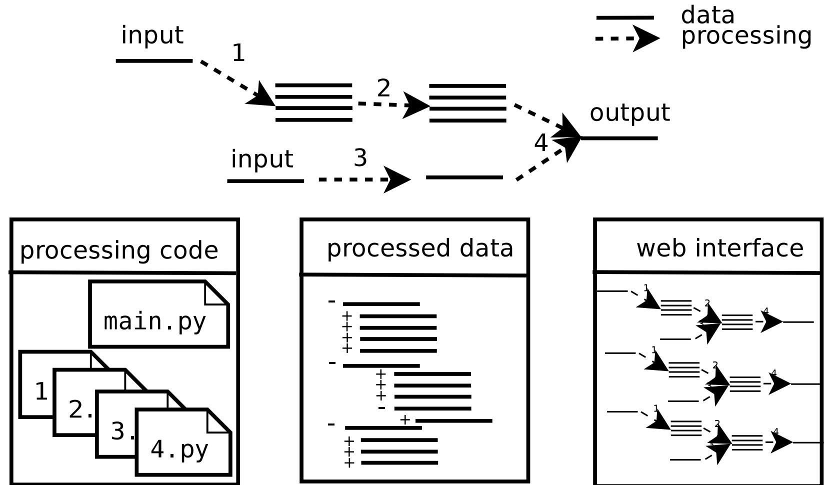
Context

How it works
- Building a pipeline
- Writing segment scripts
- Running a pipeline
- Browsing a pipeline

A CMB demo pipeline
- Problematic
- A pipelet solution
- Zooming the code
- Browsing the result
- Deployment

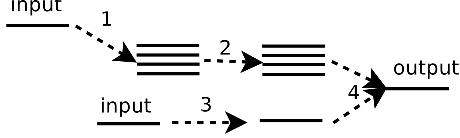
Getting started

The PIPELET big scheme



Building a pipeline

```
P = Pipeline(pipedot, codedir='./', prefix='/data/...')
```



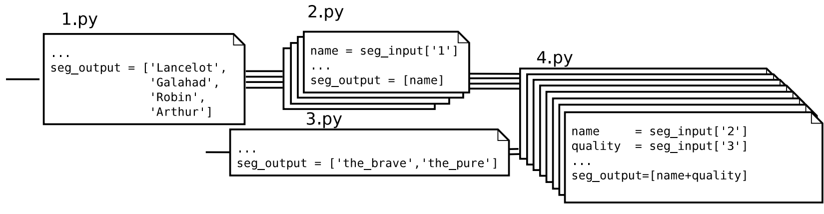
- `pipedot` is the string description of the pipeline

```
pipedot = """
1->2->4;
3->4;
"""
```

- `codedir` is the path of the processing code files (.py)
- `prefix` is the path of the processed data repository

Writing segment scripts

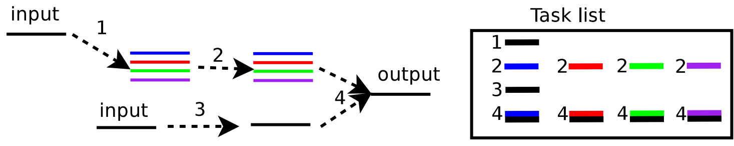
- A segment is a python script (.py file)



- It benefits from an improved namespace to:
  - provide filenames, save and load I/O's;
  - save and load parameters;
  - execute or include subprocess (and log);
  - control the pipe parallelisation scheme.

Running a pipeline

The pipe engine converts each pair of (processing code, data to process) into a task list.



One can empty the task list by launching workers in different modes:

- the process/thread mode (for smp machine)

```
python main.py -p 4
```

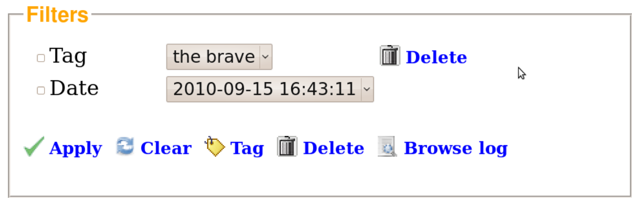
- the interactive mode (or debugging mode)

```
ipython: %pdb
ipython: run main.py -d
```

- the batch mode (for cluster)

```
python main.py -b
python main.py -a 8
```

Browsing a pipeline : http://localhost:8080



From the web interface one can:

• Filter/delete pipe instances	from the pipeline page
• Highlight dependencies	from the segment page
• Read code	from the segment page
• Read log files	from the log page
• Download/visualize/delete product files	from the product page

Context

How it works
- Building a pipeline
- Writing segment scripts
- Running a pipeline
- Browsing a pipeline

A CMB demo pipeline
- Problematic
- A pipelet solution
- Zooming the code
- Browsing the result
- Deployment

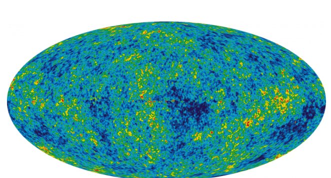
Getting started

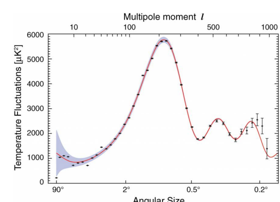
Problematic

Evaluate the performances of inverse noise weighting spectral estimation via simulations.

The wish list is:

- Design a prototype pipeline to get a first result.
- Perform Monte Carlo studies to get error bars.
  - Test different weighting masks.
  - Save coupling matrix computation as much as possible.





A pipelet solution

(nside,sim_id)

lcdm.txt

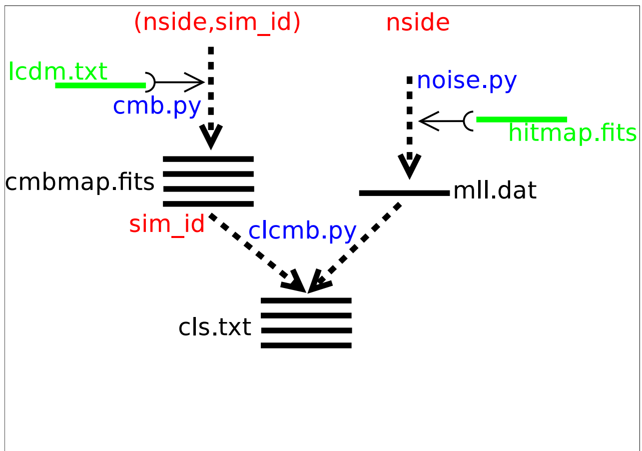
cmb.py

cmbmap.fits

A PIPELET solution

(nside,sim_id)
nside
lcdm.txt
noise.py
cmb.py
hitmap.fits
cmbmap.fits
mll.dat

A PIPELET solution

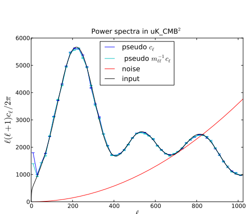


A PIPELET solution

(nside,sim_id)
nside
lcdm.txt
noise.py
cmb.py
hitmap.fits
cmbmap.fits
mll.dat
sim_id
clcmb.py
cls.txt
sim_ids
clplot.py
cls.png

A PIPELET solution

(nside,sim_id)
nside
lcdm.txt
cmb.py
noise.py
hitmap.fits
cmbmap.fits
mll.dat
sim_id
clcmb.py
clnoise.py
cls.txt
cls.txt
sim_ids
clplot.py
cls.png



Zooming the code (1/3): main.py

```
pipe_dot = """
cmb->clcmb->clplot;
noise->clcmb;
noise->clnoise->clplot;
"""
P = Pipeline(pipe_dot, code_dir='./', prefix='./cmb')

P.push(cmb =[1,2,3,4,5,6]) ## push as many inputs as CMB realizations

if options.debug:        ## Interactive mode
    w, t = launch_interactive(P)
    w.run()
elif options.process:    ## Process mode
    launch_process(P, options.process)
else:                    ## Batch mode
    launch_pbs(P, 10, job_name="pipelet_job",cpu_time="00:30:00")
```

Zooming the code (2/3): cmb.py

```
""" cmb.py
Generate a cmb map from lambda-CDM power spectrum.
"""
import healpy as hp
import pylab as pl

lst_par = ['lmax', 'nside']
sim_id  = seg_input[0] ## pushed from main
lmax    = 2*nside

input_cl = "lambda_best_fit.txt"
cmb_cl   = pl.loadtxt(input_cl)[0:lmax+1,0]     ## load cl
cmb_map  = hp.synfast(cmb_cl, nside, lmax=lmax) ## make a map

cmb_map_fn = get_data_fn ('map_cmb.fits')
hp.write_map(cmb_map_fn, cmb_map)

cmb_map_fig = cmb_map_fn.replace('.fits', '.png')
hp.mollview(cmb_map, title="cmb")
pl.savefig (cmb_map_fig)

seg_output = [sim_id] ## forward as many childs as sim ids
```

Zooming the code (3/3): clplot.py

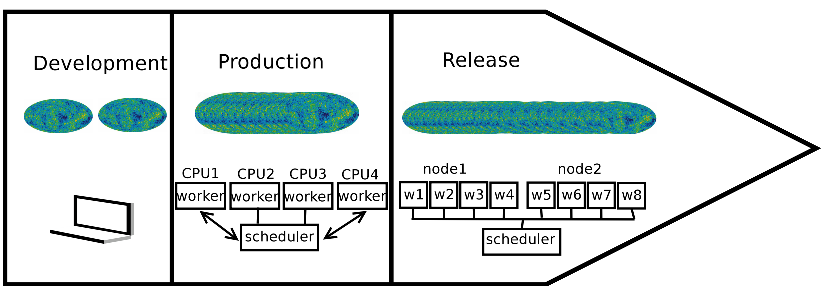
```
""" clplot.py
Make a plot.
"""
import pylab as pl

### Gather all sim_ids
#multiplex cross_prod group_by '0'

### Retrieve some global parameters
load_param('cmb',   globals(), ["lmax"])
load_param("noise", globals(), ["noise_power"])

## Get mean cl values and error bars
pseudo_cls = glob_seg('clcmb', 'cls.txt')
mll_cls    = glob_seg('clcmb', 'cl*mll*.txt')
nsims      = len(mll_cls)
for sim_id in range(nsims):
    ...

## make a plot
...
```

Browsing the result

http://localhost:8080

Deployment

1. Development phase : on your laptop.
   - use the **PIPELET** interactive mode and python debugger %pdb
2. Production phase : on a desktop machine / adamis cluster.
   - use the process or batch mode to dispatch tasks between cores
3. Release phase : @ CCin2p3 or Magique3.
   - use a cutosmized segment environment (DMC database)
   - use a customized batch launcher (BQS)



Context

How it works
Building a pipeline
Writing segment scripts
Running a pipeline
Browsing a pipeline

A CMB demo pipeline
Problematic
A pipelet solution
Zooming the code
Browsing the result
Deployment

Getting started

Getting PIPELET

▷ Download from http://gitorious.org/pipelet

- Git repository
  `git clone git://gitorious.org/pipelet/pipelet.git`
- Open wiki including documentation

▷ Features and bugs are tracked from the IN2P3 forge.
Don't hesitate to subscribe the PIPELET project to give your feedback and follow the project news.

▷ CMB demo pipeline under : `pipelet/test/cmb`

A word on Python

is a free programming language:

- high level language easy to use (as compared to C) http://mathesaurus.sourceforge.net
- runs on Linux/Unix (native), Windows, Mac OS X
- fast (as compared to Matlab, Octave, IDL)
- interfaces nicelly with other languages : C/C++ extensions or SWIG/Boost wrapping.
- Extensive standard library
- and most of all : Pythonic !

Python tools:

- interpreter: ipython
- numerical libraries: numpy + matplotlib *rightarrow* pylab
- CMB librairies: healpy, spherelib

Advanced usage and discussion

- Customizing the segment environment
  ▷ Segment environment is a namespace, provided with default utilities (filenames, parameters, subprocesses, ...)
  ▷ This namespace is loaded from a Python object which can be derived (heritage) or changed.
- Customizing launchers
  ▷ Right now available : interactive, process, batch (PBS, BQS)
- Using repositories for segment scripts:
  ▷ Git, CVS, SVN, ...
- Improving the web interface.
- Suggestions are welcomed !