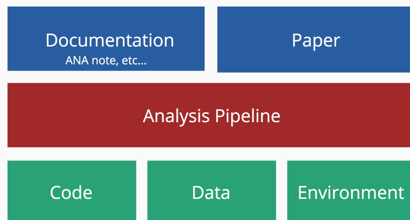
Analysis pipelines

Chris Burr
University of Manchester

May 17, 2017

What is an analysis pipeline?

- Defines the steps required to run each stage of an analysis
- Ties together data, analysis code and the analysis environment



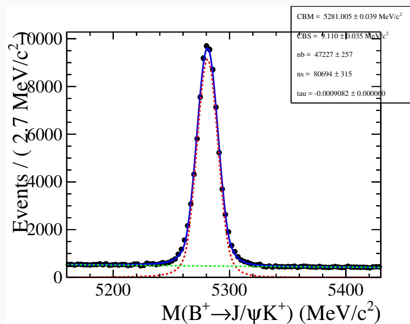
- Why are they useful?
  - Makes analysis easier
  - Source documentation that can be guaranteed to be up-to-date
  - Prevents some classes of bugs
  - Makes it easier for anyone to rerun an analysis

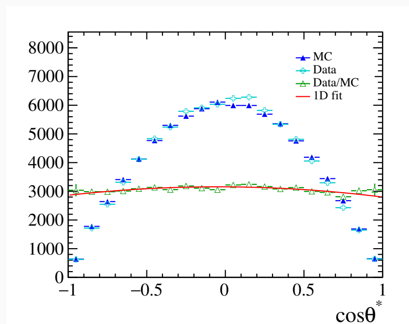
What features do we want from pipeline?

- Easy to implement
    - Make use of already common knowledge
    - Integrate with what we already have
- Automation
    - Minimise ambiguity about how an analysis is done
    - Able to compress everything from “ntuples” $\rightarrow$ paper into a single command
- Dependency tracking
    - Both local and remote (eos, bookkeeping, PDG)
- Caching of intermediate results
    - Nobody wants to wait for hours to change an axis label
- Make debugging easier rather than harder
    - It’s unhelpful if the pipeline becomes a black box
    - Should be possible to run everything independent of the pipeline

Example analysis

- Measurement of the polarisation of J/psi from B-decays
- Intended to be used as a standard candle for checking MC productions
- Several example pipelines have been implemented
    - Additional contributions welcome!
- More details available at `https://gitlab.cern.ch/lhcb-bandq/JpsiPolarisation`





Available options

Bash

Example workflow in bash

```bash
#!/bin/bash
# Set bash strict mode - http://redsymbol.net/articles/unofficial-bash-strict-mode/
set -euo pipefail
IFS=$'\n\t'

# Load the output directories from config.py
DATA_DIR=$(PYTHONPATH="${PWD}/python" python -c 'import config; print(config.data_dir)')
RESULTS_DIR=$(PYTHONPATH="${PWD}/python" python -c 'import config; print(config.results_dir)')

printf 'Preliminary step: compilation\n'
make compile

printf "STEP (1)\nPROCESSING MC EVENTS (selections and required variables)\n"
python python/CalVariables.py "BuJpsiK2012MC"

printf "PROCESSING Data EVENTS (selections and required variables)\n"
python python/CalVariables.py "BuJpsiK2012Data"

printf "STEP (2)\nGet normalization parameters (using GetNorm.py), and sweights from mass fits\n"
python python/GenerateInput.py

printf "STEP (3)\nExtract the polarization parameters\n"
bin/fitsweight "${DATA_DIR}" "${RESULTS_DIR}"

printf "STEP (-1)\n1D fit cross-check (check \"cosThetaFit1D.pdf\")\n"
python python/Compare1D.py
```

Bash

Other example analyses:

- Λ_c ΔA_{CP} `https://gitlab.cern.ch/apearce/lc2pxx-dacp`

Advantages:

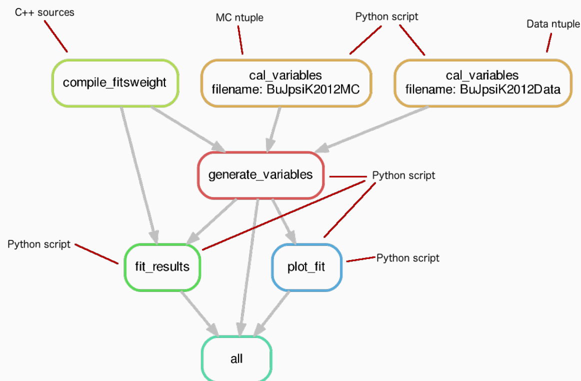
- Already installed on most systems
- Easy to interface to almost everything
- Excellent documentation

Disadvantages:

- Writing robust scripts can be hard
- No easy way to version inputs or intermediate results
- No native concept of multiprocessing

A better way - DAGs

- Directed acyclic graphs (DAGs) can be used schedule the execution of tasks
  - Each vertex represents some processing
  - Each edge represents an external dependency
  - Allows parallelised and partial execution
- Most commonly seen use of a DAG is in `make`
- For the example analysis the graph might look like:



Make

Example workflow

```
1  # Load the output directories from config.py
2  DATA_DIR := $(shell PYTHONPATH=$(PWD)/python python -c 'import config; print(config.data_dir)')
3  RESULTS_DIR := $(shell PYTHONPATH=$(PWD)/python python -c 'import config; print(config.results_dir)')
4
5  all: run
6
7  # Include the existing Makefile for bin/fitresults and the RooFit classes
8  include Makefile
9
10 output/logs:
11     @mkdir -p $@
12
13 $(DATA_DIR)/BuJpsiK2012MCUpdate.root: python/CalVariables.py python/config.py | output/logs
14     @echo -e '\e[93mSTEP (1) PROCESSING MC EVENTS (selections and required variables)\e[0m'
15     @python $< "BuJpsiK2012MC" > output/logs/mc.out
16
17 $(DATA_DIR)/BuJpsiK2012DataUpdate.root: python/CalVariables.py python/config.py | output/logs
18     @echo -e '\e[93mSTEP (1) PROCESSING Data EVENTS (selections and required variables)\e[0m'
19     @python $< "BuJpsiK2012Data" > output/logs/data.out
20
21 $(DATA_DIR)/BuJpsiK2012DataUpdate_sw.root: python/GenerateInput.py python/config.py python/GetNorm.py python/Fit
22     @echo -e '\e[93mSTEP (2) Get normalization parameters (using GetNorm.py), and sweights from mass fits (using
23     @python $< > output/logs/data_sw.out
24
25 # Phony rule to deal with multiple output files in a single rule.
26 .PHONY: fitresults
27 fitresults: bin/fitsweight $(DATA_DIR)/BuJpsiK2012DataUpdate_sw.root | output/logs
28     @echo -e '\e[93mSTEP (3) Extract the polarization parameters\e[0m'
29     @$< $(DATA_DIR) $(RESULTS_DIR) > output/logs/fitsweight.out
30
```

Make

Other example analyses:

- None that I know of

Advantages:

- Already install on most systems
- Allows caching
- Schedules jobs to run on multiple cores

Disadvantages:

- Inputs and outputs of jobs are limited (or complex to implement)
- Makefile syntax is difficult to understand for non trivial pipelines
- Easy to leave corrupt files which cause later runs to fail for obfuscated reasons

Snakemake

Snakemake

- Python with a few added pieces of syntax (compiles to pure python)
- Aimed towards active research (with preservation in mind)
- Widely used for bioinformatics research (has at least 61 citations)[1]

Tell Snakemake what files you want to be created

```
rule:
  input: "A.txt", "B.txt", "C.txt"
```

Produce the files you want to have from some intermediate result

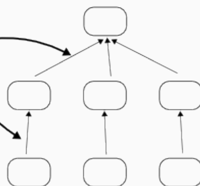
```
rule:
  input: "{sample}.inter"
  output: "{sample}.txt"
  shell: "somecommand {input} {output}"
```

Create a needed intermediate result

```
rule:
  input:  "{sample}.in"
  output: "{sample}.inter"
  run:
    somepythoncode()
```

Use wildcards to write general rules for all samples

Snakemake determines the dependencies for you



[1] http://bioinformatics.oxfordjournals.org/content/28/19/2520
https://snakemake.readthedocs.io

Example workflow - Preamble

```
1   import sys
2   from glob import glob
3   from os.path import join, relpath
4
5   from snakemake.remote.XRootD import RemoteProvider
6
7   # We have to modify sys.path to be able to import the analysis
8   sys.path.insert(0, join(os.getcwd(), 'python'))
9   import config as ana_config
10
11  # Load the config from the config module
12  config['eos_dir'] = ana_config.eos_dir
13  config['data_dir'] = ana_config.data_dir
14  config['results_dir'] = ana_config.results_dir
15
16  # Setup XRootD
17  XRootD = RemoteProvider(stay_on_remote=True)
18
19  # Set shell to move to the Python 2.7 environment each time
20  shell.prefix('source activate python-2.7; ')
21
22  rule all:
23      input:
24          join(config["results_dir"], 'cosThetaFit1D.pdf'),
25          join(config["results_dir"], 'Normalization_parameters.txt'),
26          join(config["results_dir"], 'B_mass.pdf'),
27          join(config["results_dir"], 'FitResult.txt'),
28          join(config["results_dir"], 'FitResultCov.txt')
29
```

Example workflow - Rules

```
30  rule compile_fitsweight:
31      input:
32          makefile='Makefile',
33          source=['src/LLFitSWeight.cc', 'src/RooDCBPdf.cc', 'src/lhcbStyle.cc', 'src/nobanner.cc'],
34          includes=glob('include/*.hh')
35      output:
36          binary='bin/fitsweight',
37          libs=['lib/libModels.so', 'lib/libnobanner.so']
38      log: 'output/logs/build.log'
39      threads: 2
40      shell:
41          'make -j{threads} > {log} 2>&1'
42
43  rule cal_variables:
44      input:
45          script='python/CalVariables.py',
46          modules=['python/config.py'],
47          data=XRootD.remote(config["eos_dir"]+'/{filename}.root')
48      output:
49          join(config["data_dir"], '{filename}Update.root')
50      log: 'output/logs/{filename}.out'
51      shell:
52          'python {input.script} {wildcards.filename} > {log} 2>&1'
53
54  rule generate_variables:
55      input:
56          script='python/GenerateInput.py',
57          modules=['python/GetNorm.py', 'python/FitMass.py', 'python/config.py'],
58          config='python/config.py',
59          libs=['lib/libModels.so', 'lib/libnobanner.so'],
60          data=[join(config["data_dir"], 'BuJpsiK2012MCUpdate.root'),
```

Snakemake

Other example analyses:

- Pentaquarks in $\Lambda_b^0 \to \Lambda_c^+ \bar{D}^0 K^-$ `https://gitlab.cern.ch/lhcb-bandq-exotics/Lb2LcD0K`
- $D_{(s)}^+ \to h^\pm l^- l'^\mp$ `https://gitlab.cern.ch/cburr/d2hll-analysis`

Advantages:

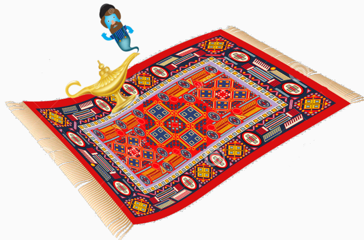
- Python based - Can run any valid python code
- Support for specifying environment per-rule
- Advanced scheduling with support for specifying resources (GPUs, RAM)
- Some support for cluster based execution
- Open to domain specific contributions:
  - XRootD support already added
  - Could add support for pulling in information from other remote resources

Disadvantages:

- Currently difficult to use at CERN (Python 3 only)

fabricate

fabricate



- Python library primarily focused on building software
- Finds dependencies automagically using `strace` (primarily)
- Parallel running is supported
- Similar to writing a series of shell commands and caching comes for free
- Uses MD5 to check for changes in files (by default)

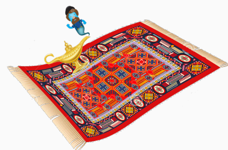
---

https://github.com/SimonAlfie/fabricate

fabricate - Example workflow

```
1   import os
2   from os.path import relpath, join
3   import sys
4   from fabricate import main, run, after
5   sys.path.insert(0, join(os.getcwd(), 'python'))
6   import config
7
8   def build():
9       compile_fitsweight()
10      cal_variables('BuJpsiK2012MC')
11      cal_variables('BuJpsiK2012Data')
12      after()  # Execution must be paused here
13      generate_input()
14      after()  # Execution must be paused here
15      fitsweight()
16      compare_1D()
17
18  def compile_fitsweight():
19      run('make', '-j2')
20
21  def cal_variables(data_type):
22      run('python', 'python/CalVariables.py', data_type)
23
24  def generate_input():
25      run('python', 'python/GenerateInput.py')
26
27  def fitsweight():
28      run('bin/fitsweight', relpath(config.data_dir), relpath(config.results_dir))
29
30  def compare_1D():
31      run('python', 'python/Compare1D.py')
32
33  if __name__ == '__main__':
34      main(build_dir=os.getcwd(), parallel_ok=True)
```

fabricate



Other example analyses:

- None that I know of

Advantages:

- Python based
- Very easy to install (single .py file)
- Automatic dependency generation
- Caches using file hashes

Disadvantages:

- Difficult to run in parallel
- Inherently limited to a single machine
- Care has to be taken to avoid subtle issues (for example running make)

Other pipelines

Luigi

- Created by spotify (used by many enterprises)
- Defines parametrised pipelines using python

Advantages:

- Deeply integrated with Python (but not exclusively)
- Good support for caching and clusters
- Excellent interface and documentation

Disadvantages:

- More aimed towards rerunning identical jobs with different inputs
- No caching of intermediate steps

https://github.com/spotify/luigi

Yadage

- Dynamic workflow engine (most must be fully defined at runtime)
- Defines parametrised workflows using yaml/json that connected together
- Connects several DIANA/HEP packages together

Advantages:

- Good support for containerised environments
- Part of DIANA/HEP so can be made very hep specific
- Could ultimately be helpful to get full reproducibility (backup)

Disadvantages:

- Error messages and documentation could be better
- No caching of intermediate rules

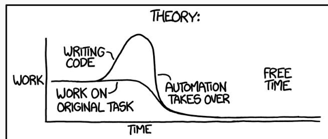
---

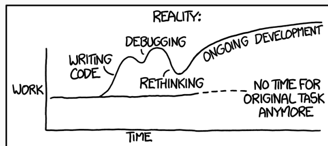
http://yadage.readthedocs.io/en/latest/

Conclusions

- There are lots of tools available
    - Many are complex and/or poorly suited to our needs
    - There are even more options available than shown here
    - People are free to use anything than can be containerised
- In my opinion:
    - Most solutions aren’t well suited to typical research needs
    - Snakemake is good for most of our needs (once installed)
    - For simple cases bash/make/fabricate have benefits

“I SPEND A LOT OF TIME ON THIS TASK. I SHOULD WRITE A PROGRAM AUTOMATING IT!”





Backup

The road to full reproducibility

- Here I focus on partial reproducibility
  - ntuples → paper
- Full reproducibility can come from integrating this into an overarching pipeline