

The



toolkit

Ben Krikler for FAST-HEP

TOOLS2020

3rd October 2020

✉ b.krikler@cern.ch

🐦 [@benkrikler](https://twitter.com/benkrikler)



University of
BRISTOL



Software
Sustainability
Institute

F.A.S.T = Faster Analysis Software Taskforce

- Started around May 2017
- Explore ways to accelerate and improve our analysis code
- Use of 1 to 3-day “hack-shops” to test new ideas



Analysis Challenges ~3 Years Ago

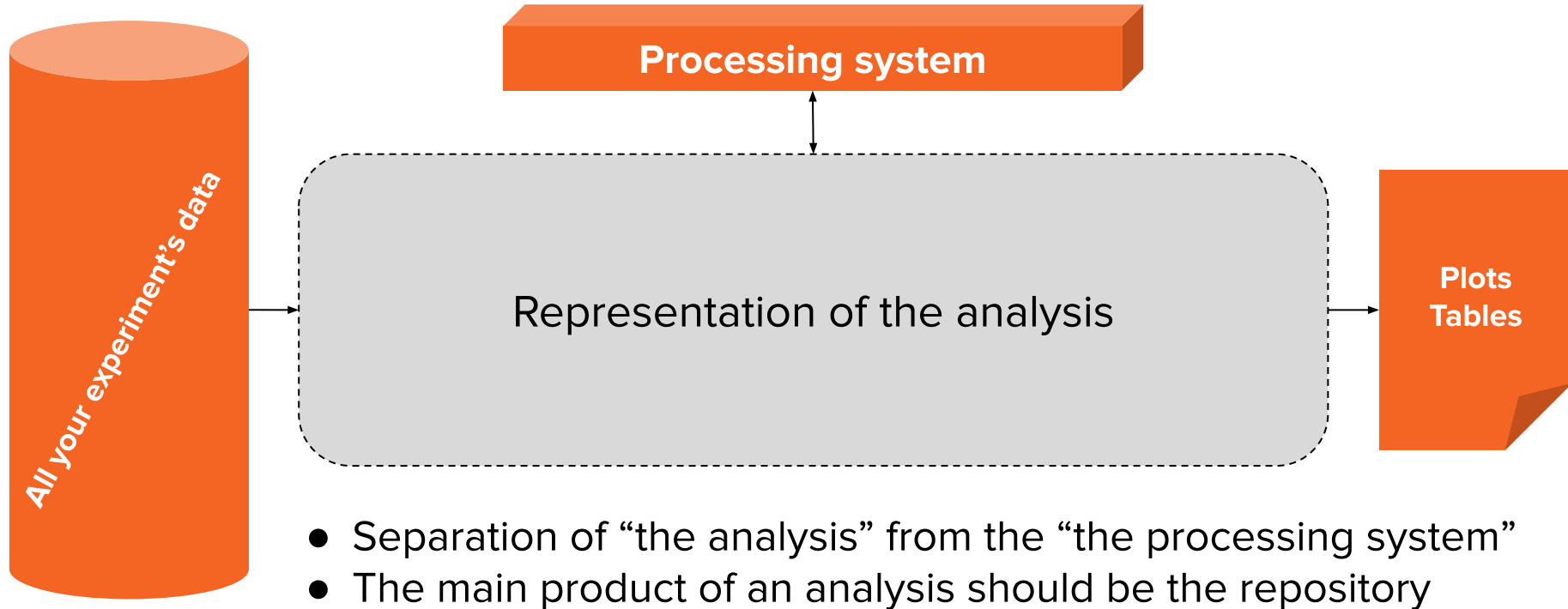
Development and processing
time slow

Brittle and inflexible

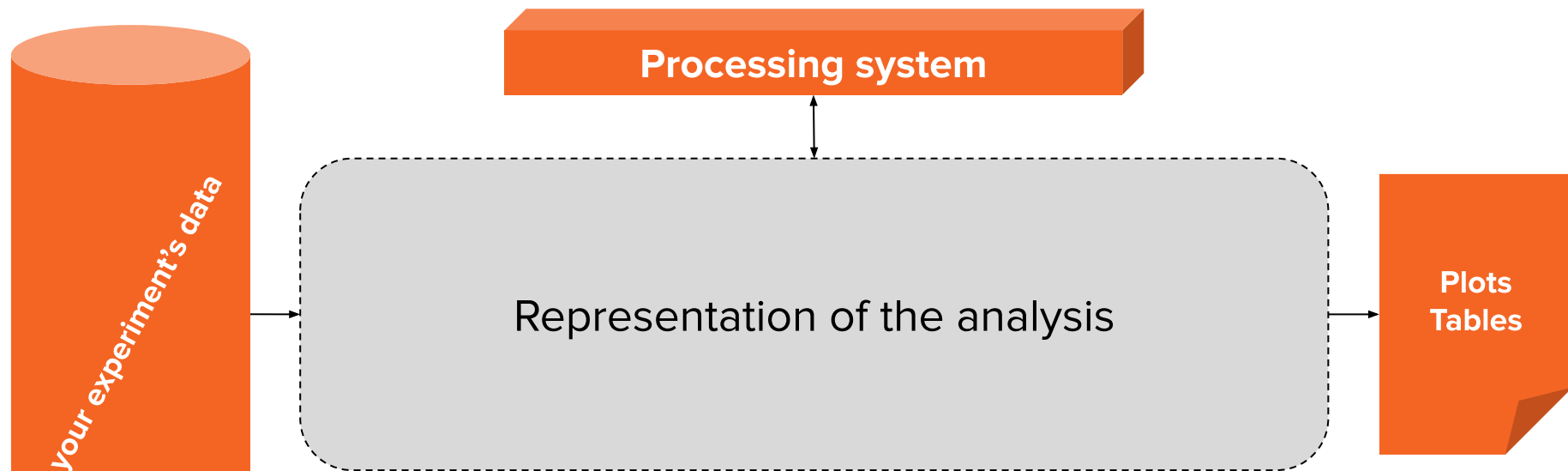
High learning curve

Contents of code
≠ publication

Analysis *versus* analysis tools



Your analysis repository *is* your analysis



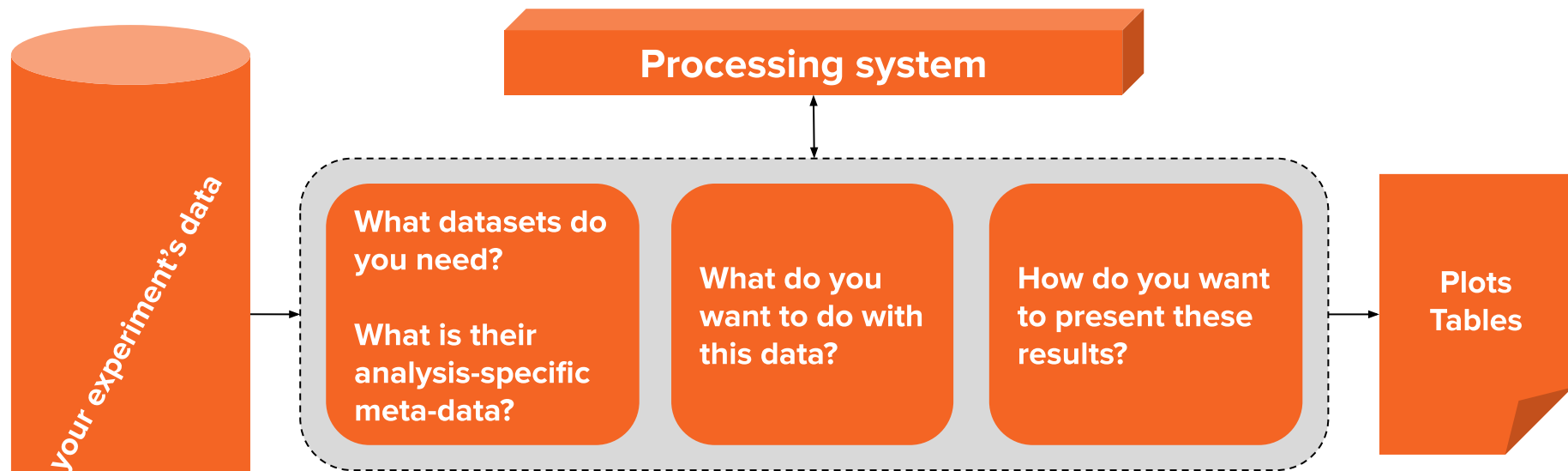
Its contents will change as:

- You design the analysis
- You get new / updated data

For free, in a repository:

- History of analysis evolution
- Continuous integration and validation

Your analysis repository *is* your analysis



Its contents will change as:

- You design the analysis
- You get new / updated data

For free, in a repository:

- History of analysis evolution
- Continuous integration and validation

How can the analysis description be:

Concise

Shareable

Flexible

Complete

Quick



Analysis description languages

A large fraction of LHC analyses involve only a few steps

Can we encapsulate these into a “Domain Specific Language”?

Several different attempts to build an ADL:

- [LINQ \(Gordon Watts et al\)](#)
- [NAIL \(Andrew Rizzi\)](#)
- FAST-HEP (this talk)
- Dedicated workshop at Fermilab May 2019:
<https://indico.cern.ch/event/769263/>

The mathematical description in your paper is what you write to do the analysis

Declarative programming

- Declarative languages the **user says WHAT**, the **interpretation decides HOW**
- User gives up flow control:
 - Cannot do: “Loop over each event, add this to that if something is true, etc”
- Allows:
 - More concise description
 - Fewer bugs
 - Easier to reproduce and share
 - Optimisation behind the scenes

The FAST implementation

For tools:
use Python



uproot
Awkward
Array

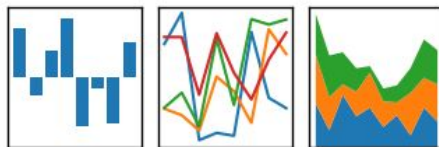
NumExpr

at (☕)

For data:
use Pandas
Demoed at CHEP 2018

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



For descriptions:
use YAML...

Describing analysis with YAML

- A superset of JSON
 - Easier to read
- Naturally declarative:
 - No “control flow” (e.g. no for loops)
- Widely used to describe pipeline configuration:
 - gitlab-CI, travis-CI, Azure CI/CD, Ansible, Kubernetes, etc
 - HEPData: YAML for reproducible Data

```
[{"martin":{"name": "Martin Devloper",  
  "job": "Developer",  
  "Skills": ["python", "perl", "pascal"]}  
, {"tabitha":{"name": "Tabitha Bitumen", "job":  
  "Developer", "Skills": ["lisp", "fortran",  
  "erlang"]}}]
```

JSON

```
- martin:  
  name: Martin Devloper  
  job: Developer  
  skills:  
    - python  
    - perl  
    - pascal  
- tabitha:  
  name: Tabitha Bitumen  
  job: Developer  
  skills:  
    - lisp  
    - fortran  
    - erlang
```

YAML

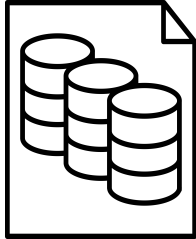
**What datasets do
you need?**

**What is their
analysis-specific
meta-data?**

**What do you
want to do with
this data?**

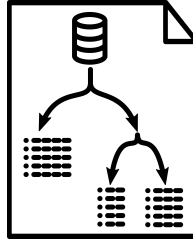
**How do you want
to present these
results?**

Step 1:
fast_curator



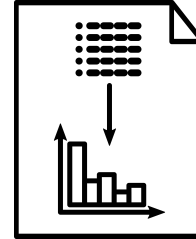
Dataset
description

Step 2:
fast_carpenter
(using *fast-flow*)



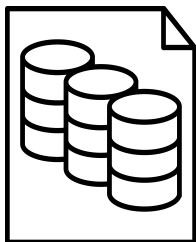
Analysis
description

Step 3:
fast_plotter
fast_datacard



Plotting and
postprocessing

Step 1:
fast_curator



**Dataset
description**

Curator: what files do you want to work on?

Dataset descriptions don't change often

- Track descriptions in repo, easy to review

Command line tool to help write YAML

- Wild-card on the command line
- Hooks ready for experiment-specific catalogues, e.g. CMS DAS
- Integrate with Rucio (?)

Dataset description

datasets:

- eventtype: **data**
Files: [**input_files/HEPTutorial/files/data.root**]
name: **data**
nevents: **469384**
- files:
 - **input_files/HEPTutorial/files/dy.root**
 - **input_files/HEPTutorial/files/dy_2.root**name: **dy**
nevents: **77729**
nfiles: **2**

defaults:

- eventtype: **mc**
- nfiles: **1**
- tree: **events**

import:

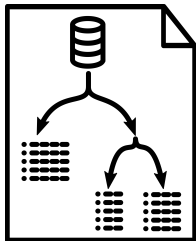
- **"{this_dir}/WW.yml"**
- **"{this_dir}/WZ.yml"**

- Each dataset has a list of files
- A unique dataset name

- Default metadata

- Can Import other dataset files
- Build complex nested dataset descriptions

Step 2:
fast_carpenter



**Analysis
description**

Take your trees and make them into tables

- Just like a carpenter

Table = Pandas DataFrame

Two main types of table for now:

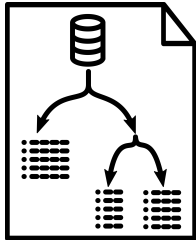
- Histogram
- Cutflow

Cover most typical particle physics analyses

- BUT: very easy to extend

Command-line switch between different
work-flow managers / batch systems

Step 2:
fast_carpenter



Analysis
description

Take your trees and make them into tables

- Just like a carpenter

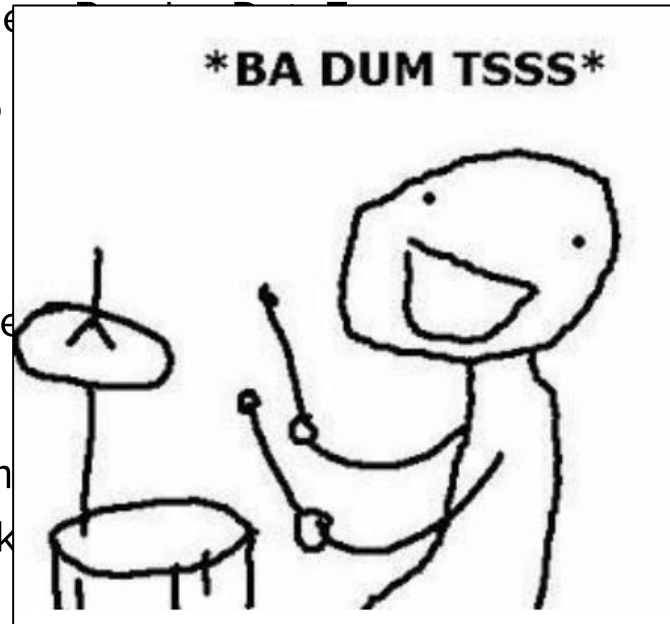
Table

Two

Cove

Com

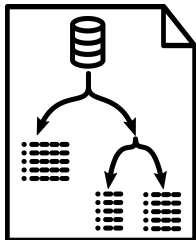
work



alyses

ht

Step 2:
fast_carpenter



**Analysis
description**

Take your trees and make them into tables

- Just like a carpenter

Table = Pandas DataFrame

Two main types of table for now:

- Histogram
- Cutflow

Cover most typical particle physics analyses

- BUT: very easy to extend

Command-line switch between different
work-flow managers / batch systems

Describe what to do with the data

What type of action to take at each step:

- Stage1 = A built-in stage of fast-carpenter
- Stage2 = A stage imported from a python module
- IMPORT = Import a list of stages and their descriptions from another YAML file

Configure each named stage above

stages:

- Stage1: `StageFromBackend`
- Stage2: `module.that.provides.some.Stage`
- IMPORT: `"{this_dir}/another_description.yaml"`

Stage1:

keyword: `value`
another_keyword: `[a, list, of, values]`

Stage2:

arg1:
takes: `["a", "dict"]`
with: `3`
different: `keys`

An example set of stages

stages:

Just defines new variables

- BasicVars: **Define**

*# A custom class to form the invariant mass of a
two-object system*

- DiMuons: **cms_hep_tutorial.DiObjectMass**

Filled a binned dataframe

- NumberMuons: **fast_carpenter.BinnedDataframe**

Select events by applying cuts

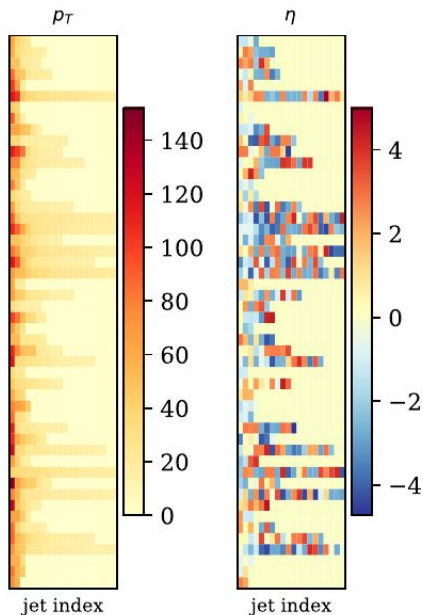
- EventSelection: **CutFlow**

Fill another binned dataframe

- DiMuonMass: **BinnedDataframe**

Define Stage:

fast_carpenter.Define



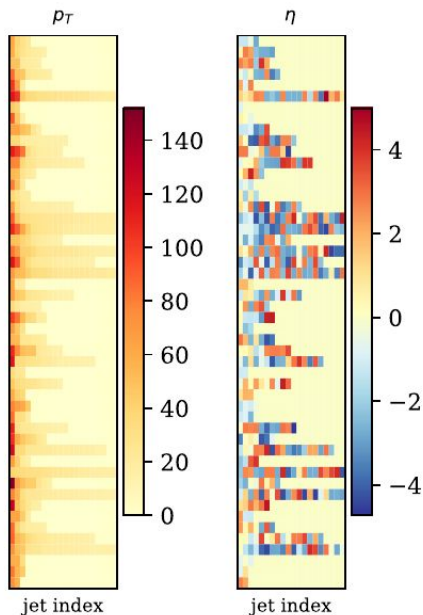
- Muon_Pt: `"sqrt(Muon_Px ** 2 + Muon_Py ** 2)"`
- IsoMuon_Idx: `(Muon_Iso / Muon_Pt) < 0.10`
- HasTwoMuons: `NIsoMuon >= 2`

- Simple operations
- Preserve the “jaggedness”

From Joosep Pata's
talk at PyHEP

Define Stage:

fast_carpenter.Define



From Joosep Pata's
talk at PyHEP

- Muon_Pt: `"sqrt(Muon_Px ** 2 + Muon_Py ** 2)"`
- IsoMuon_Idx: `(Muon_Iso / Muon_Pt) < 0.10`
- HasTwoMuons: `NIsoMuon >= 2`

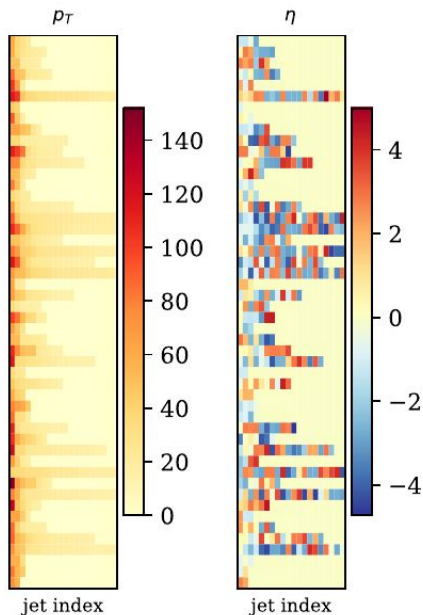
Take the Nth object
(on the deepest dimension)

- Muon_lead_Pt: `{reduce: 0, formula: Muon_Pt}`
- Muon_sublead_Pt: `{reduce: 1, formula: Muon_Pt}`

- Simple operations
- Preserve the "jaggedness"

Define Stage:

fast_carpenter.Define



From Joosep Pata's
talk at PyHEP

- Muon_Pt: `"sqrt(Muon_Px ** 2 + Muon_Py ** 2)"`
- IsoMuon_Idx: `(Muon_Iso / Muon_Pt) < 0.10`
- HasTwoMuons: `NIsoMuon >= 2`

Take the Nth object
(on the deepest dimension)

- Muon_lead_Pt: `{reduce: 0, formula: Muon_Pt}`
- Muon_sublead_Pt: `{reduce: 1, formula: Muon_Pt}`

- NIsoMuon:
 formula: `IsoMuon_Idx`
 reduce: `count_nonzero`
- IsoMuPtSum:
 formula: `Muon_Pt`
 reduce: `sum`
 mask: `IsoMuon_Idx`

- Simple operations
- Preserve the "jaggedness"

- Reduce dimensionality with a function
- Mask out objects in the event

Select events

fast_carpenter.CutFlow

```
DiMu_controlRegion:
  weights: {nominal: weight}
  selection:
    All:
      - {reduce: 0, formula: Muon_pt > 30}
      - leadJet_pt > 100
      - DiMuon_mass > 60
      - DiMuon_mass < 120
      - Any:
          - nCleanedJet == 1
          - DiJet_mass < 500
          - DiJet_deta < 2
```

Remove events from subsequent stages

Produces a cut-flow summary table

- Weighted / raw counts

Selection is specified as nested dictionaries of **All**, **Any** and a list of expressions

Individual cuts use same scheme as variable definition

Output of CutFlow stage

```
>>> import pandas as pd
>>> pd.read_csv("cuts_EventSelection-weighted.csv", header=[0, 1], index_col=[0, 1, 2])
```

			passed_incl		passed_excl		totals_excl	
			unweighted	EventWeight	unweighted	EventWeight	unweighted	EventWeight
dataset	depth	cut						
data	0	All	15995.0	15995.000000	15995.0	15995.000000	469384.0	469384.000000
	1	NIsoMuon >= 2	16208.0	16208.000000	16208.0	16208.000000	469384.0	469384.000000
		triggerIsoMu24 == 1	469384.0	469384.000000	16208.0	16208.000000	16208.0	16208.000000
dy	0	{'formula': 'Muon_Pt > 25', 'reduce': 0}	229710.0	229710.000000	15995.0	15995.000000	16208.0	16208.000000
	1	All	37263.0	16628.843750	37263.0	16628.843750	77729.0	34115.511719
		NIsoMuon >= 2	37559.0	16829.451172	37559.0	16829.451172	77729.0	34115.511719
qcd	0	triggerIsoMu24 == 1	77729.0	34115.511719	37559.0	16829.451172	37559.0	16829.451172
	1	{'formula': 'Muon_Pt > 25', 'reduce': 0}	73374.0	32168.121094	37263.0	16628.843750	37559.0	16829.451172
		All	0.0	0.000000	0.0	0.000000	142.0	79160.507812
single_top	0	NIsoMuon >= 2	0.0	0.000000	0.0	0.000000	142.0	79160.507812
	1	triggerIsoMu24 == 1	142.0	79160.507812	0.0	0.000000	0.0	0.000000
		{'formula': 'Muon_Pt > 25', 'reduce': 0}	16.0	6014.819336	0.0	0.000000	0.0	0.000000
ttbar	0	All	110.0	5.676235	110.0	5.676235	5684.0	311.622986
	1	NIsoMuon >= 2	111.0	5.748312	111.0	5.748312	5684.0	311.622986
		triggerIsoMu24 == 1	5684.0	311.622986	111.0	5.748312	111.0	5.748312
wjets	0	{'formula': 'Muon_Pt > 25', 'reduce': 0}	5278.0	290.494965	110.0	5.676235	111.0	5.748312
	1	All	206.0	47.293686	206.0	47.293686	36941.0	7929.475586
		NIsoMuon >= 2	226.0	51.629749	226.0	51.629749	36941.0	7929.475586
ww	0	triggerIsoMu24 == 1	4515.0	1001.804932	206.0	47.293686	226.0	51.629749
	1	{'formula': 'Muon_Pt > 25', 'reduce': 0}	5067.0	1109.433960	206.0	47.293686	206.0	47.293686
		All	1.0	0.311917	1.0	0.311917	109737.0	209603.531250
wZ	0	NIsoMuon >= 2	1.0	0.311917	1.0	0.311917	109737.0	209603.531250
	1	triggerIsoMu24 == 1	109737.0	209603.531250	1.0	0.311917	1.0	0.311917
		{'formula': 'Muon_Pt > 25', 'reduce': 0}	99016.0	191354.781250	1.0	0.311917	1.0	0.311917
ZZ	0	All	243.0	12.577849	243.0	12.577849	4580.0	229.949570
	1	NIsoMuon >= 2	244.0	12.639496	244.0	12.639496	4580.0	229.949570
		triggerIsoMu24 == 1	4580.0	229.949570	244.0	12.639496	244.0	12.639496
ZZ	0	{'formula': 'Muon_Pt > 25', 'reduce': 0}	4214.0	212.997131	243.0	12.577849	244.0	12.639496
	1	All	623.0	13.157759	623.0	13.157759	3367.0	69.927917
		NIsoMuon >= 2	623.0	13.157759	623.0	13.157759	3367.0	69.927917
ZZ	0	triggerIsoMu24 == 1	3367.0	69.927917	623.0	13.157759	623.0	13.157759
	1	{'formula': 'Muon_Pt > 25', 'reduce': 0}	3125.0	65.436157	623.0	13.157759	623.0	13.157759
		All	1232.0	8.985804	1232.0	8.985804	2421.0	16.922522
ZZ	0	NIsoMuon >= 2	1235.0	8.998816	1235.0	8.998816	2421.0	16.922522
	1	triggerIsoMu24 == 1	2421.0	16.922522	1235.0	8.998816	1235.0	8.998816
		{'formula': 'Muon_Pt > 25', 'reduce': 0}	2325.0	16.362473	1232.0	8.985804	1235.0	8.998816

Resulting cut-flow outputs from EventSelection config on
earlier slide

Fill a histogram

fast_carpenter.BinnedDataFrame
fast_carpenter.BuildAghast

```
NumberMuons:
  binning:
    - {in: NMuon}
    - {in: NIsoMuon}
  weights: [EventWeight, EventWeight_NLO_up]

DiMuonMass:
  binning:
    - in: DiMuon_Mass
      bins: {low: 60, high: 120, nbins: 60}
  weights: {weighted: EventWeight}
```

- Binning scheme:
 - Assume variable already discrete (eg. NumberHits)
 - Equal-width bins over a range (eg. DiMuonMass)
 - List of bin edges
- Event weights
 - Multiple weight schemes add columns
- Output written to disk:
 - Pandas to produce a dataframe in any format
 - Also (experimentally) to a Ghast

Output of BinnedDataframe stage

```
>>> import pandas as pd
>>> df = pd.read_csv('tbl_dataset.dimu_mass--weighted.csv')
>>> print(df.groupby('dataset').nth([0, 1, 2]).set_index('dimu_mass', append=True))
```

		n	weighted:sumw	weighted:sumw2
data	(-inf, 60.0]	993.0	NaN	NaN
	(60.0, 61.0]	38.0	NaN	NaN
	(61.0, 62.0]	25.0	NaN	NaN
dy	(-inf, 60.0]	821.0	655.570801	1017.549133
	(60.0, 61.0]	56.0	23.963226	12.091142
	(61.0, 62.0]	56.0	25.572840	13.094129
qcd	(-inf, 60.0]	0.0	0.000000	0.000000
	(60.0, 61.0]	0.0	0.000000	0.000000
	(61.0, 62.0]	0.0	0.000000	0.000000
single_top	(-inf, 60.0]	32.0	1.741041	0.100682
	(60.0, 61.0]	1.0	0.065288	0.004263
	(61.0, 62.0]	1.0	0.005831	0.000034
ttbar	(-inf, 60.0]	49.0	11.392980	3.072051
	(60.0, 61.0]	3.0	0.840432	0.236490
	(61.0, 62.0]	2.0	0.319709	0.075986
wjets	(-inf, 60.0]	1.0	0.311917	0.097292
	(60.0, 61.0]	0.0	0.000000	0.000000
	(61.0, 62.0]	0.0	0.000000	0.000000
ww	(-inf, 60.0]	61.0	3.600221	0.221474
	(60.0, 61.0]	1.0	0.063284	0.004005
	(61.0, 62.0]	2.0	0.102053	0.005617
wZ	(-inf, 60.0]	15.0	0.320914	0.007842
	(60.0, 61.0]	2.0	0.053328	0.001424
	(61.0, 62.0]	0.0	0.000000	0.000000
zz	(-inf, 60.0]	47.0	0.360053	0.002981
	(60.0, 61.0]	0.0	0.000000	0.000000
	(61.0, 62.0]	0.0	0.000000	0.000000

Showing only first three rows for each dataset (using groupby operation)

User-defined stages

```
stages:
  - BasicVars: fast_carpenter.Define
  - DiMuons: cms_hep_tutorial.DiObjectMass
  - Histogram: BinnedDataframe

...

DiMuons:
  mask: IsoMuon_Idx
```

- Carpenter should provide most commonly needed stages
- But if it doesn't: can define your own
 - Break out of declarative YAML to full, imperative python
- Any importable python class with the correct interface
- Keep separation of analysis decision from data-flow

User-defined stages

```
def event(self, chunk):  
    # Get the data as a pandas dataframe  
    px, py, pz, energy = chunk.tree.arrays(self.branches, outputtype=tuple)  
  
    # Rename the branches so they're easier to work with here  
    if self.mask:  
        mask = chunk.tree.array(self.mask)  
        px = px[mask]  
        py = py[mask]  
        pz = pz[mask]  
        energy = energy[mask]  
  
    # Find the second object in the event (which are sorted by Pt)  
    has_two_obj = px.counts > 1  
  
    # Calculate the invariant mass  
    p4_0 = TLorentzVectorArray(px[has_two_obj, 0], py[has_two_obj, 0],  
                               pz[has_two_obj, 0], energy[has_two_obj, 0])  
    p4_1 = TLorentzVectorArray(px[has_two_obj, 1], py[has_two_obj, 1],  
                               pz[has_two_obj, 1], energy[has_two_obj, 1])  
    di_object = p4_0 + p4_1  
  
    # insert nans for events that have fewer than 2 objects  
    masses = np.full(len(chunk.tree), np.nan)  
    masses[has_two_obj] = di_object.mass  
  
    # Add this variable to the tree  
    chunk.tree.new_variable(self.out_var, masses)  
    return True
```

Different processing backends

Command-line option to switch processing engine:

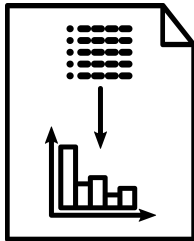
- Local multi-core processing
- htcondor
- Sun grid engine (i.e. qsub)
- Experimental (via Coffea): DASK, Parsl

```
fast_carpenter [--mode local] --ncores 4 datasets.yml \  
  processing.yml
```

```
fast_carpenter --mode htcondor datasets.yml \  
  processing.yml
```

```
fast_carpenter --mode coffea:dask \  
  --execution-cfg exec.yml datasets.yml processing.yml
```

Step 3:
fast_plotter
fast_datacard



Plotting and
postprocessing

fast-plotter:

- Easy to produce basic plots, tools to support final publication-quality
- Command-line tool with reasonable defaults and simple configuration

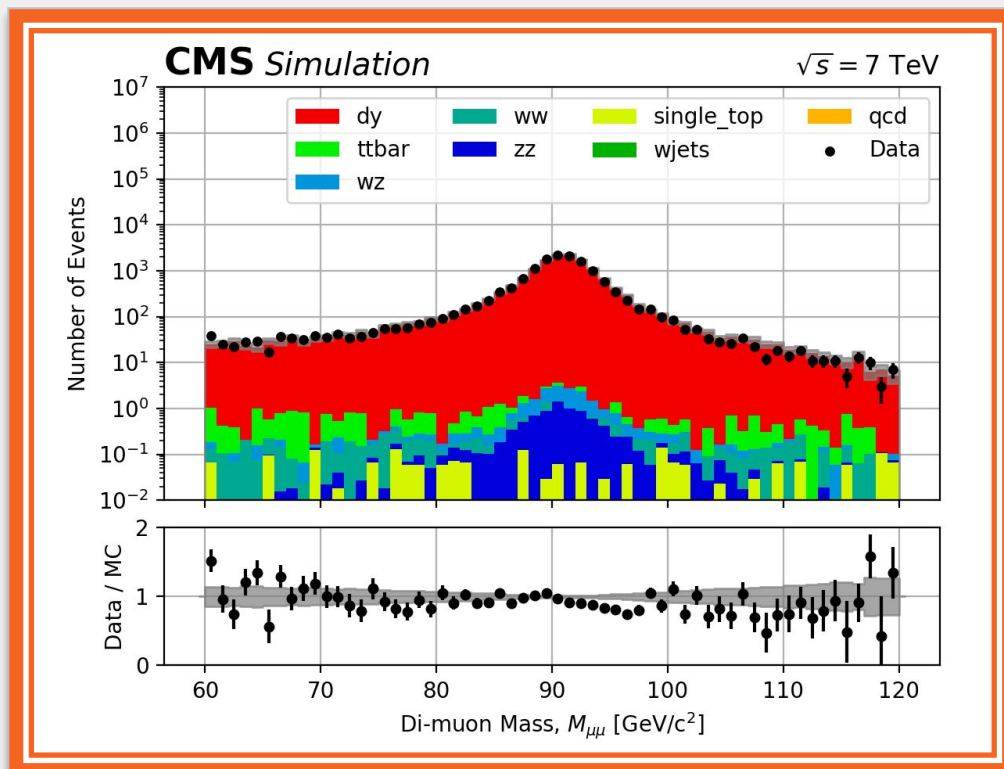
fast-datacard:

- Bring resulting DataFrames into CMS' Combine fitting procedures

BinnedDataframes into plots

- Plot on the right with:

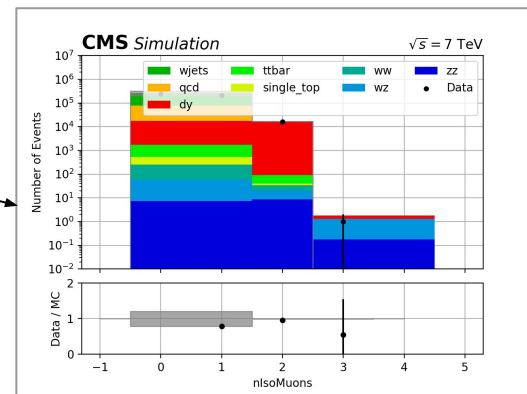
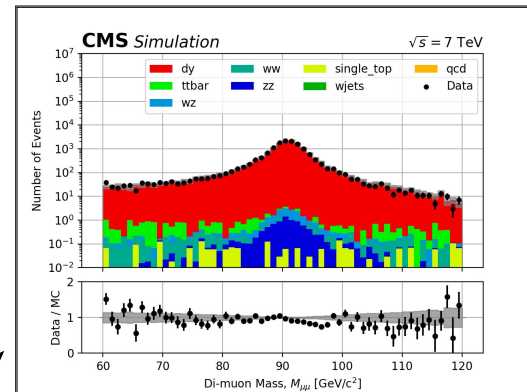
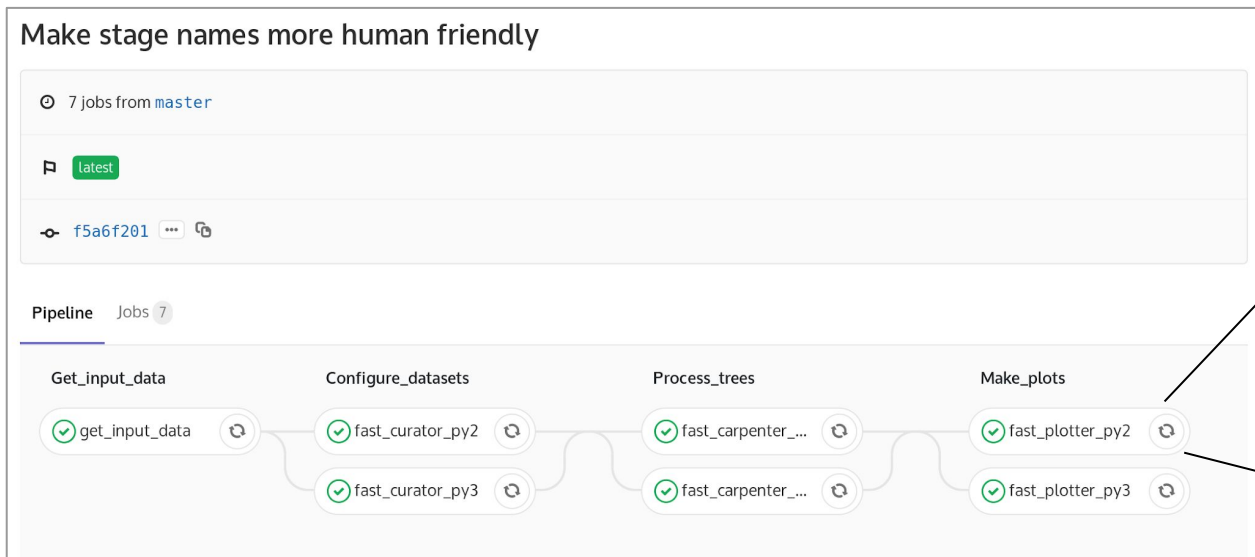
```
fast_plotter -y log \  
-c plot_config.yml \  
-o tbl_*.csv
```
- YAML config:
 - Colour scheme, axis labels
 - Dataset definition
 - Annotations
 - Legend



Plot of DiMuonMass using binned dataframe from fast-carpenter stage

"Analysis in a CI pipeline"

Make stage names more human friendly



- To run this:
 - [Demo analysis in a pipeline](#)
 - [The gitlab-ci config](#)
 - [Script tying the commands together](#)
- Feasibility for huge datasets unclear, but can happily manage subsets of data for testing

Current FAST-HEP codebase

Demonstrate the previous principles

- A Minimal Viable Product where we're continually adding features
- Hope to cover most analyses using just YAML
- Easy to add user features when FAST-HEP doesn't include

Being used for **CMS analyses**, **LUX-ZEPLIN** getting going, design studies for **DUNE**, **FCC** experiments

- New features being fed back to core packages from analysis-specific repositories
- Contributions growing from various activities

Keep our packages “slim”

Major updates during 2020

Core team development:

- New post-processing module (within fast-plotter)
- Built-in book-keeping (stores versions, configs, etc)
- Integrating against Coffea
- Still to come:
 - Multi-tree support
 - Formulae on structured objects
 - Better support for bin-migration systematics

New features being fed back to core packages from analysis-specific repositories

- Direct use in Jupyter notebooks
- Writing skimmed / slimmed outputs
- Persistency outside of CSV formats
- Docker container for running at NERSC, etc

Where to find the code

- All public on github:
 - github.com/fast-hep/
 - Main package:
github.com/fast-hep/fast-carpenter
- On PyPI, e.g. [fast-carpenter](https://pypi.org/project/fast-carpenter/)
- Docker image with all tools:
[fasthep/fast-hep-docker](https://fasthep.github.io/fast-hep-docker)
- Docs: fast-carpenter.readthedocs.io/
- Clonable demo analysis repository:
 - [gitlab.cern.ch/fast-hep/public/fast cms public tutorial](https://gitlab.cern.ch/fast-hep/public/fast_cms_public_tutorial)
- Chat: gitter.im/FAST-HEP

fast-carpenter
BKS, pin, ataga

Search docs

CONTENTS:

- Installing
- Key Concepts
- Command-line Usage
- The Processing Config
- Example repositories
- Glossary

CODE REFERENCE

- fast_carpenter package
- fast_carpenter.backends package
- fast_carpenter.backends.alphatwirl module
- fast_carpenter.backends.coffea module
- fast_carpenter.define package
- fast_carpenter.define.reductions module
- fast_carpenter.define.systematics module
- fast_carpenter.define.variables module
- fast_carpenter.event_builder module
- fast_carpenter.expressions module
- fast_carpenter.help module
- fast_carpenter.masked_tree module
- fast_carpenter.selection package
- fast_carpenter.selection.filters module
- fast_carpenter.selection.stage module
- fast_carpenter.summary package
- fast_carpenter.summary.ghast module
- fast_carpenter.summary.binned_dataframe module
- fast_carpenter.summary.binning_config module

Docs » Indices and tables [Edit on GitHub](#)

pyth v0.15.1 build passing codecov 100% docs failing chat on gitter DOI 10.5201/zenodo.3520144

FAST carpenter

Turns your trees into tables (i.e. reads ROOT TFiles, writes summary Pandas DataFrames)

fast-carpenter can:

- Be controlled using YAML-based config files
- Define new variables
- Cut out events or define phase-space "regions"
- Produce histograms stored as CSV files using multiple weighting schemes
- Make use of user-defined stages to manipulate the data

Powered by:

- AlphaTwirl (presently): to run the dataset splitting
- Atuproot: to adapt AlphaTwirl to use uproot
- uproot: to load ROOT TFiles into memory as numpy arrays
- fast-flow: to manage the processing config files
- fast-curator: to orchestrate the lists of datasets to be processed
- Espresso: to keep the developer(s) writing code

A tool from the Faster Analysis Software Taskforce: <http://fast-hep.web.cern.ch/>

Contents:

- Installing
 - From PyPI
 - From Source
- Key Concepts
 - Goals of fast-carpenter
 - From the user's perspective
 - From the code and development perspective
 - Overall approach for data-processing
 - Step 1: Create dataset configs
 - Step 2: Write a processing config
 - Step 3: Run fast_carpenter
 - Step 4: Produce plots
- Command-line Usage

Summary

High-level analysis faces major many challenges

- Lots more data coming but fewer relative resources
- Steep learning curve for newcomers
- Brittle code and environments
- *Hard to reproduce an analysis from just the paper*

The FAST-HEP toolkit tries to tackle this

- Builds off rapidly maturing Python for HEP ecosystem
- More: “what” an analysis does, less: “how” is it implemented
- Being used for analysis on several experiments
- Keen to expand contributors!

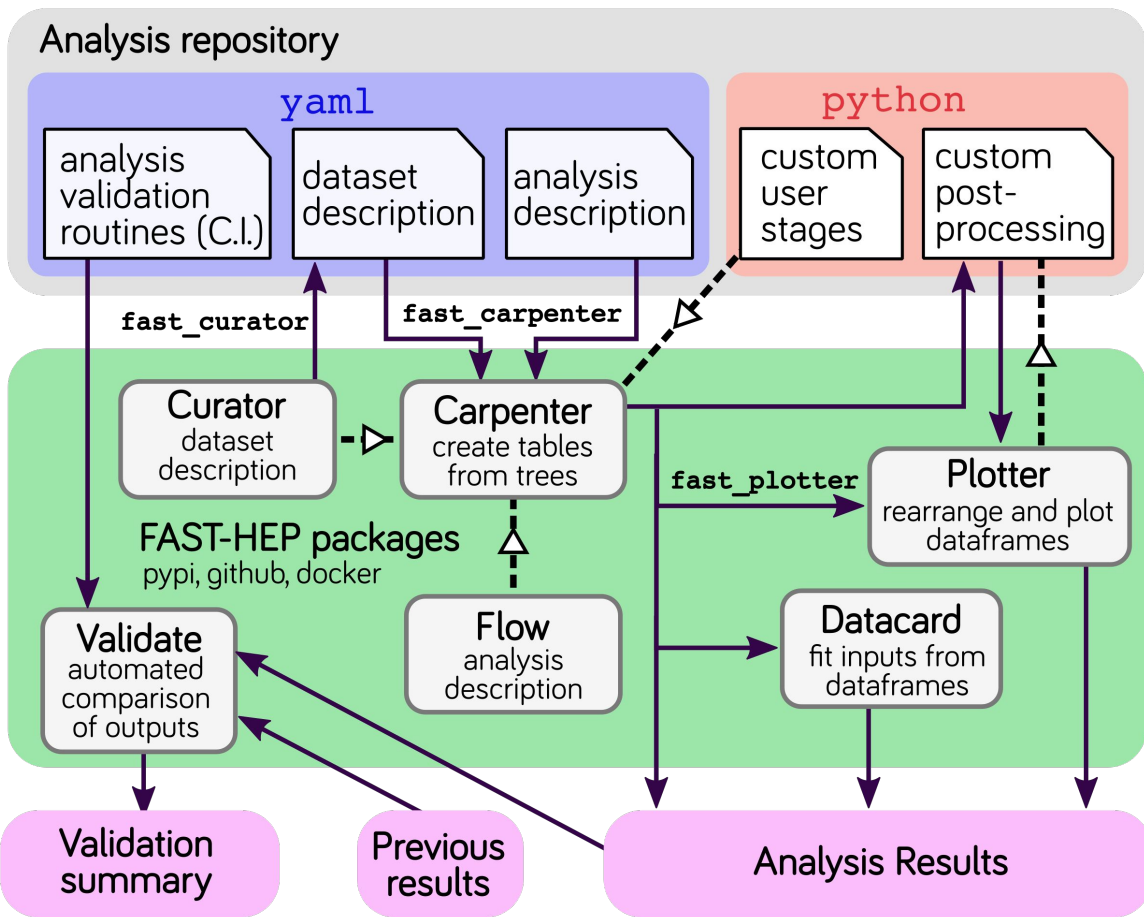


Thank You

✉ *b.krikler@cern.ch*

🐦 *@benkrikler*

Interplay in a typical user's analysis repo



Jupyter Notebook?

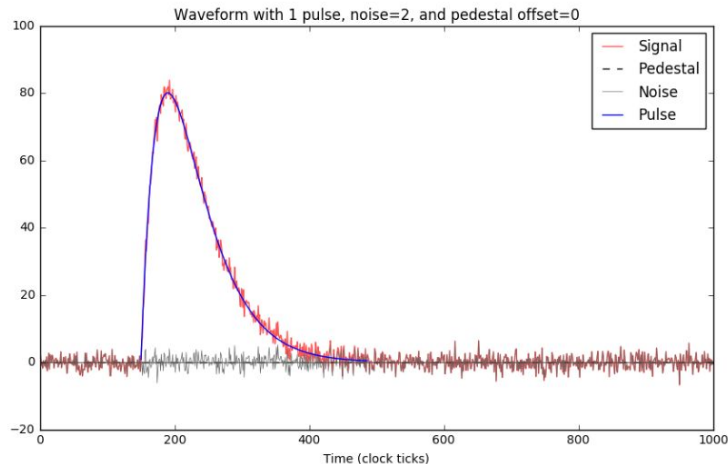
Waveforms will contain multiple components:

- Noise
- Pedestal
- One or more actual signal pulses

Here we assume that the shape of a signal pulse is given by the expression: $f(x; \tau) = x e^{1-x/\tau}$

```
In [3]: wave=Waveform([[150,80]],noise=2,pedestal=0)
        wave.plot_all(show_noise=True)
        plt.legend()
```

```
Out[3]: <matplotlib.legend.Legend at 0x7fb5b6ff8860>
```



Template pulse

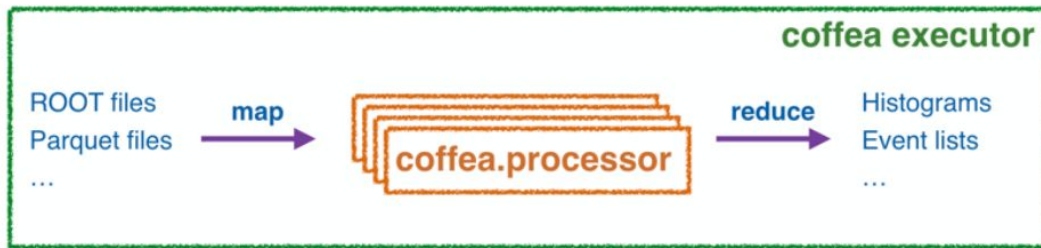
Now we set up our template pulse. We cheat here and use the analytic expression that we know is being used to generate the pulses, but in a real situation this would be a sizeable task, involving pulse registration and averaging.

We also fix all pulse shaping times from here on, to 50 ticks.

To fix: shaping time=50

- Great:
 - Mixing code, documentation, and results
- Bad:
 - Code can still be dense
 - Scaling to full analysis?
 - Connecting to batch system tricky
 - Version control
- Carpenter can be used via Python API:
provide python dicts instead of YAML
 - Addresses some of bad points above

Coffea - Column Object Framework for Effective Analysis



Fermilab project to build an analysis framework on top of awkward array and uproot

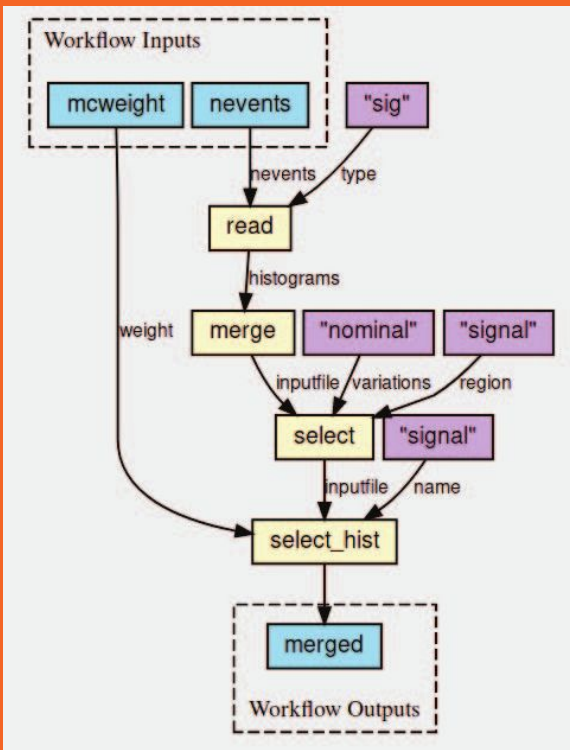
Separation of “user code” and “executors”

- User writes a Processor to do the analysis
- Executor runs this on different distributed job systems, e.g.:
 - Local multiprocessing, Parsl or Dask (batch systems), **Spark** cluster

Coffea **achieved 1 to 3 MHz** event processing rates

- Using Spark cluster on same site as data at Fermilab

Workflow managers



Let another system take care of your workflow

Many tools:

- SnakeMake, Parsl, Airflow
- More integrated: Spark, DASK

Define your analysis as a Directed Acyclic Graph (DAG)

- Each task is a node
- Data flow is an arrow connecting two nodes

For free:

- Caching of repeated tasks at each node
- Can optimise the DAG: “elide” (remove) nodes if result is never used

What is Pandas?

- Programmatic tables, built on numpy
- A staple of data science
- <https://pandas.pydata.org/>

```
A = ['foo', 'bar', 'foo', 'bar']
B = ['one', 'one', 'two', 'three']
C = np.random.randn(4)
D = np.random.randn(4)

df = pd.DataFrame({"A": A, "B": B,
                   "C": C, "D": D})
```

df

	A	B	C	D
0	foo	one	-0.678386	0.072926
1	bar	one	-0.338564	-1.038362
2	foo	two	0.527912	-0.478806
3	bar	three	-0.237991	-1.296666

df.set_index(["A", "B"])

		C	D
A	B		
foo	one	-0.678386	0.072926
bar	one	-0.338564	-1.038362
foo	two	0.527912	-0.478806
bar	three	-0.237991	-1.296666

All built-in stages

- Full list of stages can be found with:

```
$ fast_carpenter  
--help-stages
```

- Can get full help for specific stage e.g.:

```
$ fast_carpenter  
--help-stages-full  
CutFlow
```

- **Define:** Create new variables
- **SystematicWeights:** Create event weights with systematic variations from multiple sources
- **CutFlow:** Remove events failing cuts and summarize # of events passing each cut
- **SelectPhaseSpace:** Like CutFlow but creates mask without applying it
- **BinnedDataframe:** Creates a binned pandas dataframe that can be fed into fast-plotter
- **BuildAghast:** Like BinnedDataframe but result is a Ghast

How we have worked

Design principles:

- Write as little code as possible: act as glue
- Contribute first to other projects
- Value modularity

Goals:

- a. Reproducibility
- b. Simplicity
- c. Speed
- d. Documentation
- e. Automation

User-defined stages

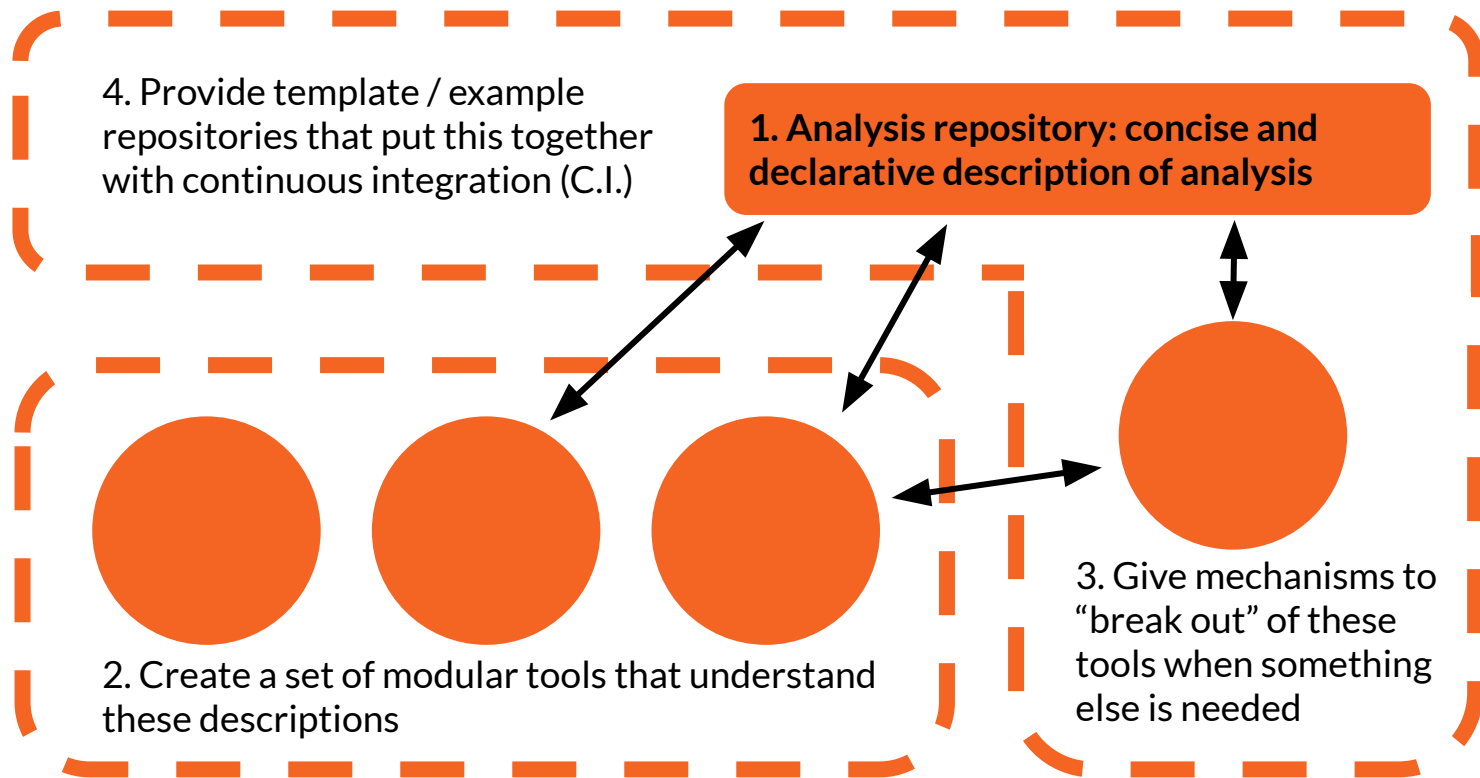
Parameters
controlled
from analysis
description

```
from uproot_methods import TLorentzVectorArray
import numpy as np
```

```
class DiObjectMass():
    def __init__(self, name, out_dir, collection="Muon", mask=None, out_var=None):
        self.name = name
        self.out_dir = out_dir
        self.mask = mask
        self.collection = collection

        self.branches = [self.collection + "_" + var for var in ["Px", "Py", "Pz", "E"]]
        if out_var:
            self.out_var = out_var
        else:
            self.out_var = "Di{}_Mass".format(collection)
```

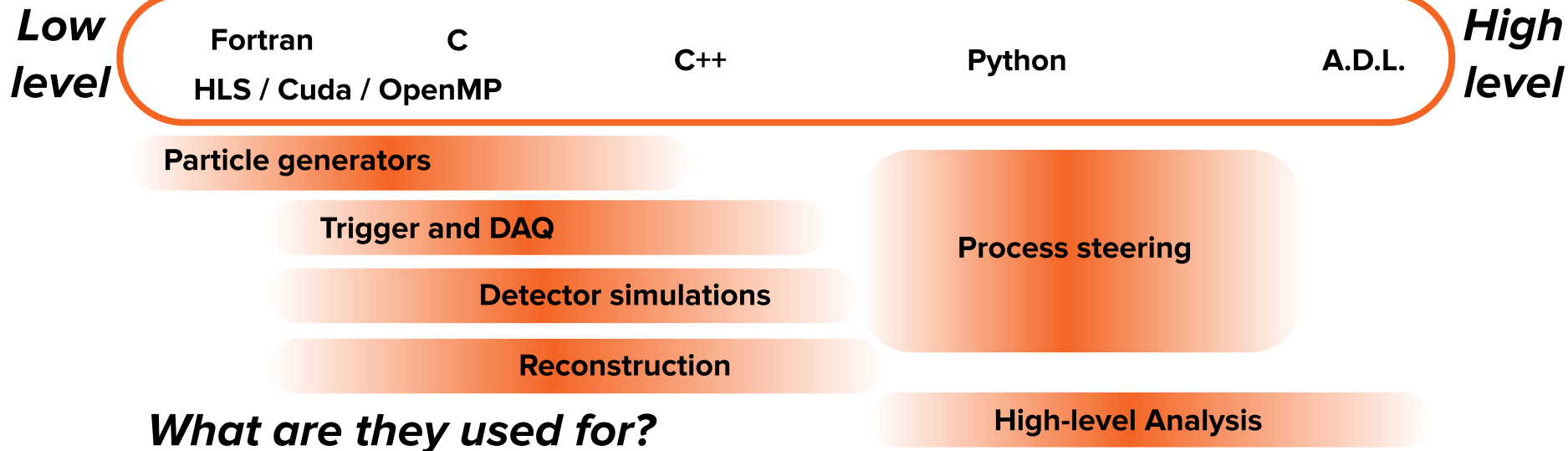
Your analysis repository is your analysis



The future HEP code landscape (?)

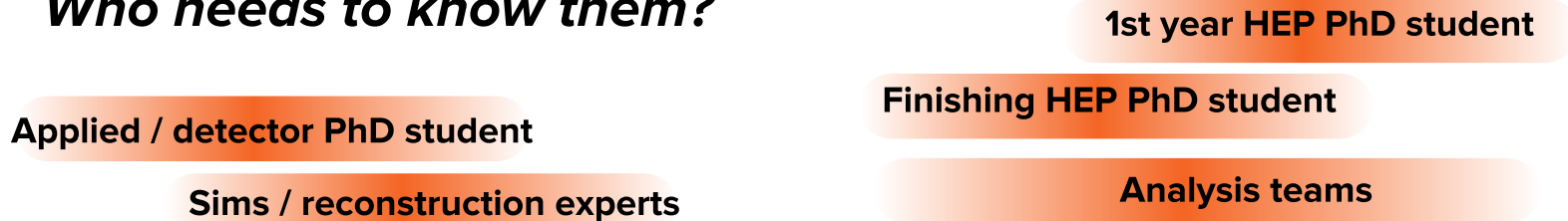


The future HEP code landscape (?)



The future HEP code landscape (?)

Who needs to know them?



Particle generators

Trigger and DAQ

Detector simulations

Reconstruction

Process steering

High-level Analysis

What are they used for?