fastjet

VECTORIZED JET-FINDING IN PYTHON

Aryan Roy
IRIS-HEP Fellow
Manipal Institute of Technology

Jim Pivarski
Mentor
Princeton University

Fastjet: A Tool for Jet Finding

- The **C++ library** is the standard for performing Jet finding in HEP.
- Various interfaces in Python exist, including official bindings provided by the Fastjet authors.
- The official bindings require loops over Python objects:

```
>>> data = []
>>> for elem in raw_data:
...   data.append(fastjet.PseudoJet(elem[0],elem[1],elem[2],elem[3]))
```

This is acceptable in C++, however, doing the same thing in Python is a performance bottleneck that needs to be eliminated.

Fastjet: A Tool for Jet Finding

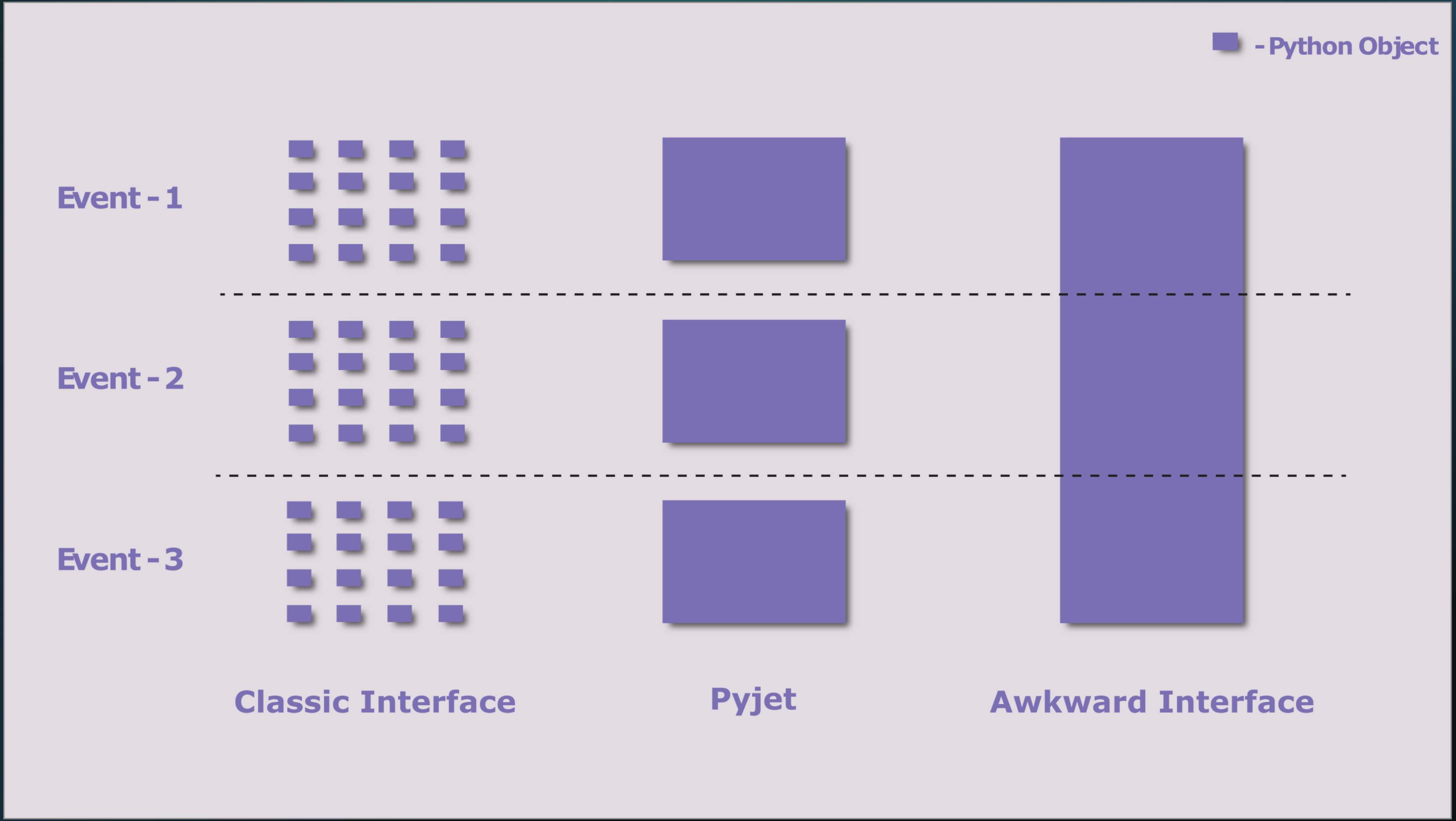
- Some wrappers were made that partly solved the problem, one such example is pyjet.
- It provides event level vectorization, which means all the PseudoJets for an event are cleanly packaged into a numpy array which enables the user to use vectorization.
- This only solves the problem partially because often in Jet finding, users work with multiple events. If we want to vectorize multi-event data, we need something that can handle deeply nested, variable length lists, like Awkward Array!
- Using Awkward Arrays also saves us from frequent conversions between data types to use the modern Scikit-HEP tools.

Awkward Array

The New Package

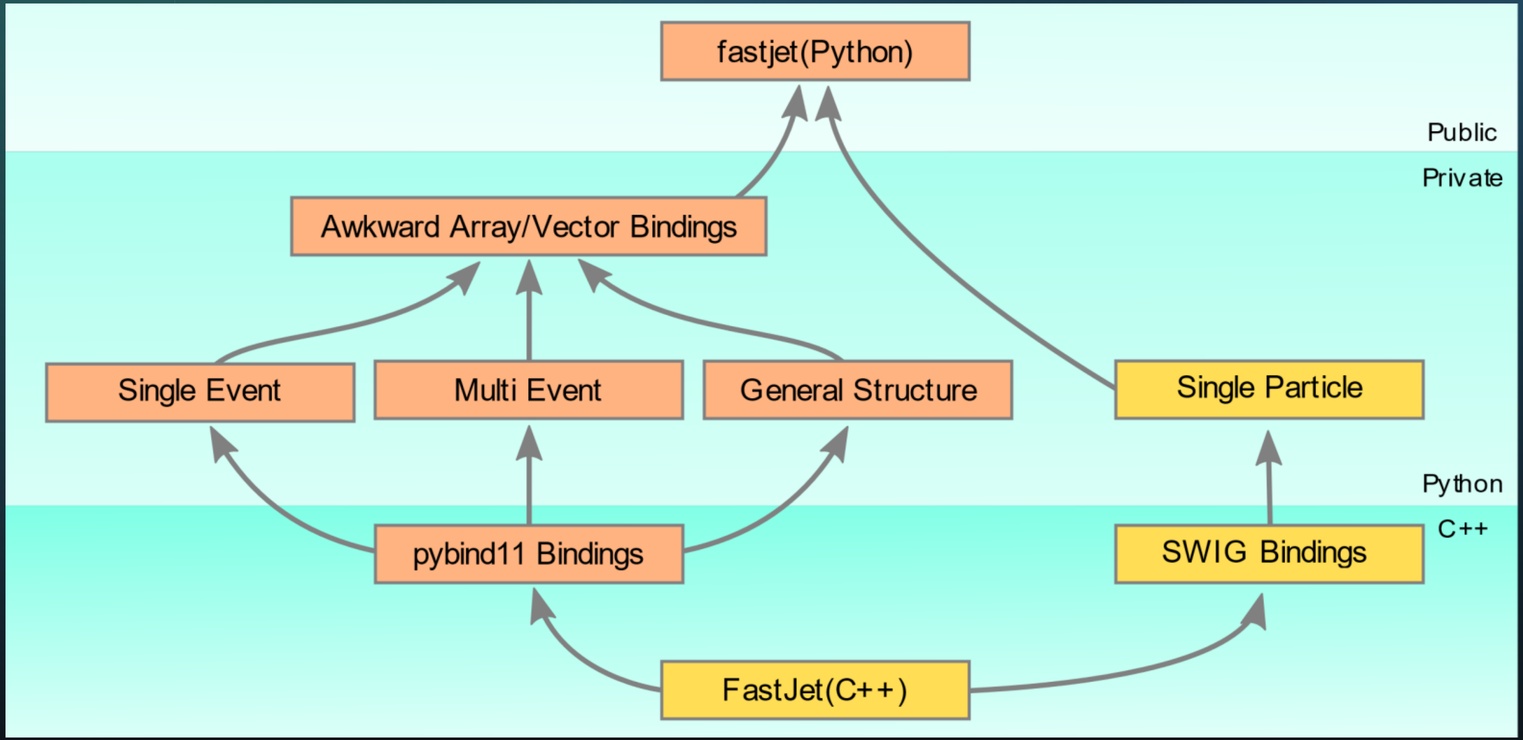
- pyjet is based on a FastJet subset that can give different answers than the real FastJet.
- The pyjet interface diverges considerably from the C++ FastJet, which can be a problem.
- Given this, we decided to create **fastjet**, a new package that aims to consolidate and address all the issues in the currently available alternatives.
- fastjet is a new python package that contains the official bindings along with an Awkward Array interface.
- This is the first time that the official bindings for FastJet will be available through pypi.

Vectorizing Particle Data Handling



The Structure of the interface

- The new package contains two interfaces within: The Awkward Array interface and the Classic interface.
- The Classic interface is generated when the C++ library is compiled (using SWIG), which we pass through as is.
- The Awkward Array interface is the new interface that can handle multi-event data.



Now, you can get all of this through a single **pip install**!

The Classic Interface

```
>>> import fastjet
>>> data = [
...     fastjet.PseudoJet(1.2,3.2,5.4,2.5),
...     fastjet.PseudoJet(32.2,64.21,543.34,24.12),
...     fastjet.PseudoJet(32.45,64.21,543.14,24.56),
... ]
>>> jetdef = fastjet.JetDefinition(fastjet.antikt_algorithm, 0.6)
>>> classic_clusters = fastjet.ClusterSequence(data,jetdef)
```

It takes a list of PseudoJets,
Which is a Python equivalent
of the C++ Fastjet that
takes a vector of PseudoJets.

```
>>> output = classic_clusters.inclusive_jets()
>>> for jet in output:
...   print(jet.px())

1.2
64.65
```

Information is returned in the form of list of PseudoJets as well. (Only where the returned Information has to be a list of particles.)

The Single Event Case

```
>>> import awkward as ak
>>> array = ak.Array(
... [
...     {"px": 1.2, "py": 3.2, "pz": 5.4, "E": 2.5, "charge": 1},
...     {"px": 32.2, "py": 64.21, "pz": 543.34, "E": 24.12, "charge": -1},
...     {"px": 32.45, "py": 63.21, "pz": 543.14, "E": 24.56, "charge": 1},
... ],
... )
>>> jetdef = fastjet.JetDefinition(fastjet.antikt_algorithm, 0.6)
>>> vectorized_clusters = fastjet.ClusterSequence(array,jetdef)
```

The Single event case takes the input as a simple Record Array, with possibly extra fields.

From Object Oriented to Vectorized

```
>>> print(vetorized_clusters.inclusive_jets().px)
[1.2, 64.7]
```

Extracting information from the Awkward Interface is the same as the classic interface syntactically. However, the outputs of lists of particles are Awkward Arrays, not list of PseudoJets.

The Object oriented interface not only requires looping over Python Objects, but also extra lines of code to properly handle the output.

```
>>> output = classic_clusters.inclusive_jets()
>>> for jet in output:
...   print(jet.px())

1.2
64.65
```

It’s much cleaner when it’s vectorized!

The Multi Event Case

Using Awkward Array, we can define a multi event case like this:

```
>>> array = ak.Array(
...     [
...         [
...             {"px": 1.2, "py": 3.2, "pz":5.4, "E": 2.5, "charge": 1},
...             {"px": 32.2, "py": 64.21, "pz": 543.34, "E": 24.12, "charge": -1},
...             {"px": 32.45, "py": 63.21, "pz": 543.14, "E": 24.56, "charge": 1},
...         ],
...         [], # empty event
...         [
...             {"px": 2.95, "py": -0.35, "pz": 0.62, "E": 2.86, "charge": 1},
...             {"px": 4.33, "py": 0.53, "pz": 1.47, "E": 3.95, "charge": -1},
...             {"px": 0.32, "py": 0.06, "pz": 0.12, "E": 0.21, "charge": 1},
...             {"px": 0.32, "py": 0.01579, "pz": 0.01, "E": 0.32, "charge": 1},
...         ],
...         [
...             {"px": 9.74, "py": -0.01, "pz": 0.23, "E": 9.73, "charge": -1}
...         ],
...     ],
... )
```

The Multi Event Case

Now the Awkward Array can be given as argument to the constructor of the ClusterSequence Class, much like the last two cases:

```
>>> jetdef = fastjet.JetDefinition(fastjet.antikt_algorithm, 0.6)
>>> multievent_clusters = fastjet.ClusterSequence(array,jetdef)
>>> print(multievent_clusters.inclusive_jets().px)
<Array [[1.2, 64.7], [], [7.94], [9.74]] type='4 * var * float64'>
```

In this case, the constituents of each inclusive jet also contains the “charge” field that was present in the input. We can use this to calculate the sum of charges of each inclusive jet!

```
>>> print(ak.sum(multievent_clusters.constituents().charge, axis = -1))
[[1, 0], [], [2], [-1]]
```

This is possible because the “constituents” are just a re-ordering of the original input.

Replacing The PseudoJet Class

Since the PseudoJet class has been completely taken out, something needs to take it’s place. This job is performed by the Vector library in Awkward interface.

```
>>> import vector
>>> vector.register_awkward()
>>> input_data = ak.Array(
...     [[
...             {"pt": 1.2, "eta": 3.2, "phi": 2.14, "mass":  0.13957},
...             {"pt": 32.2, "eta": -3.1, "phi": 1.67, "mass":  0.13957},
...             {"pt": 32.45, "eta": -3.14, "phi": 1.66, "mass":  0.13957},
...     ],
...     [
...             {"pt": 1.2, "eta": -4.2, "phi": -2.89, "mass":  0.13957},
...             {"pt": 32.2, "eta": -4.21, "phi": -2.891, "mass":  0.13957},
...             {"pt": 32.45, "eta": 1.25, "phi": 1.68, "mass":  0.13957},
...     ]],
...     with_name = "Momentum4D", # This line specifies that this is a lorentz vector
... )
```

All you need to do is register your Awkward Arrays!

Replacing The PseudoJet Class

The Awkward Array we just constructed can be input into the ClusterSequence class directly.

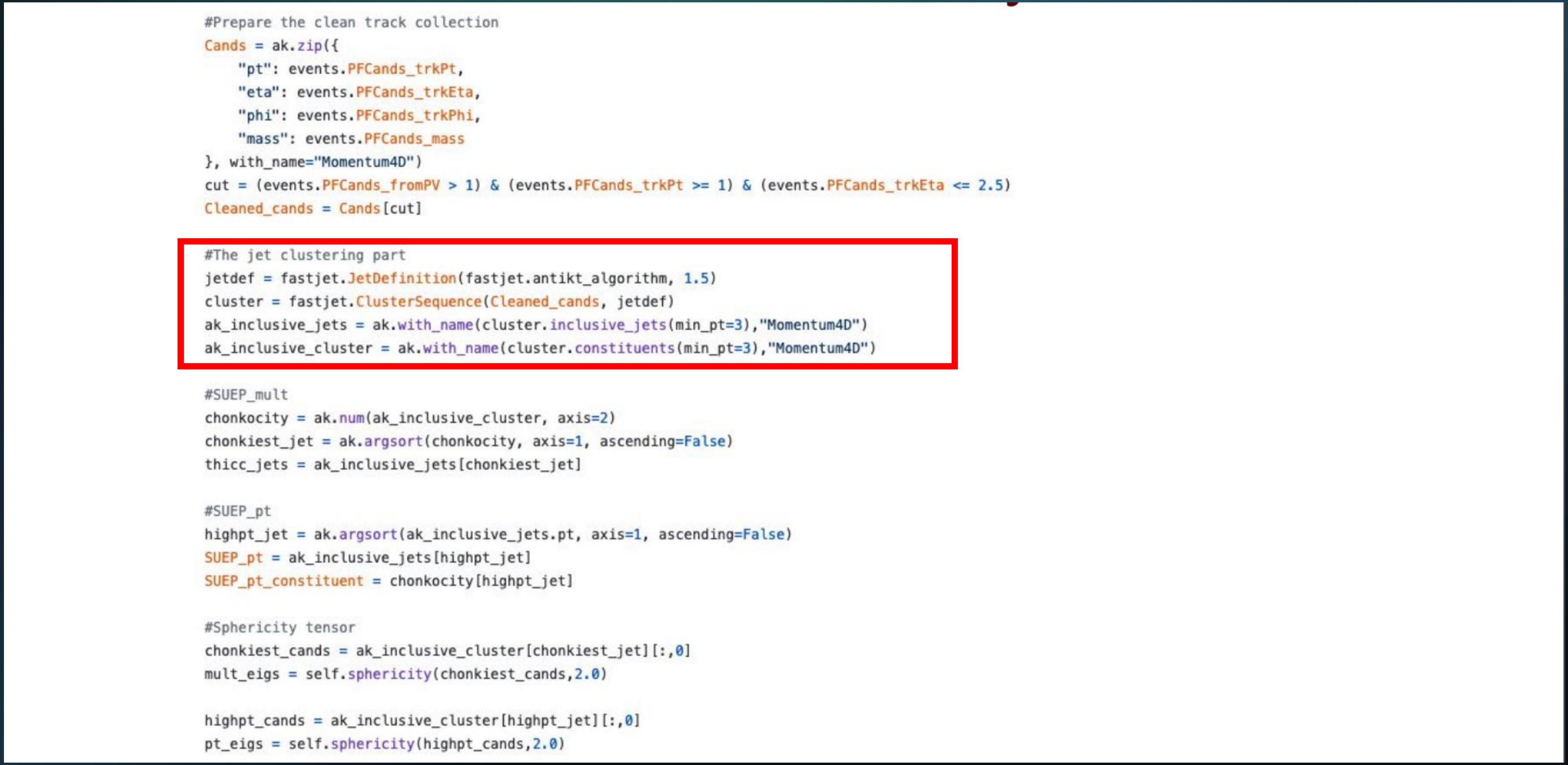
```
>>> cluster = fastjet.ClusterSequence(input_data, jetdef)
>>> cluster.constituents()
<MomentumArray4D [[[{pt: 1.2, ... mass: 0.241}]]] type='2 * var * var * Momentum...'>
```

The constituents here are also just a reordering of the original input, only this time, the input was in the pt-eta-phi-mass coordinate system.

Any particle data that is generated by the library will be cartesian, however, registering the Awkward Arrays allows the user to freely convert between different coordinate systems.

The Scikit-HEP Ecosystem

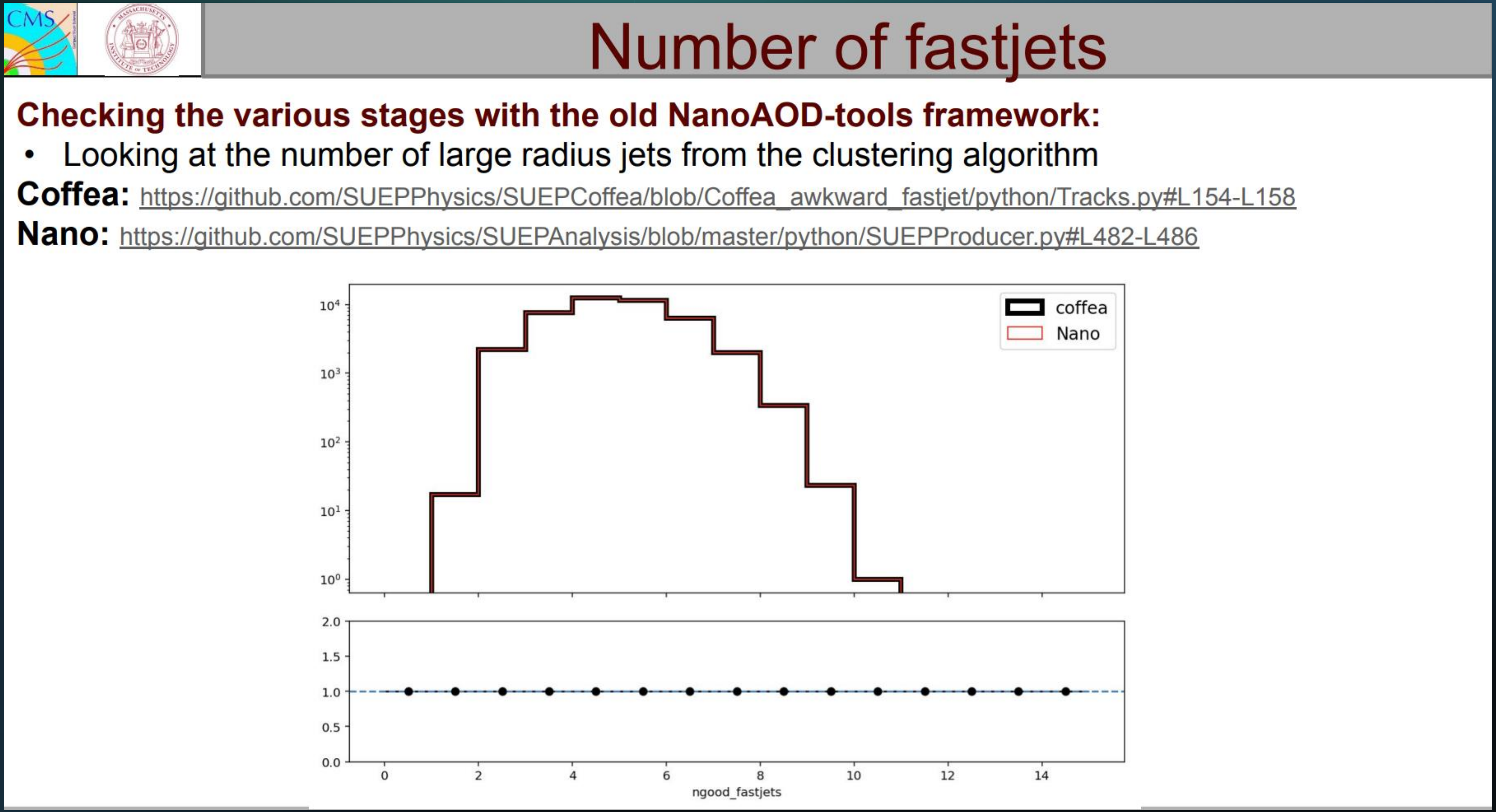
There's already analysis being developed based on fastjet!



```
#Prepare the clean track collection
Cands = ak.zip({
    "pt": events.PFCands_trkPt,
    "eta": events.PFCands_trkEta,
    "phi": events.PFCands_trkPhi,
    "mass": events.PFCands_mass
}, with_name="Momentum4D")
cut = (events.PFCands_fromPV > 1) & (events.PFCands_trkPt >= 1) & (events.PFCands_trkEta <= 2.5)
Cleaned_cands = Cands[cut]

#The jet clustering part
jetdef = fastjet.JetDefinition(fastjet.antikt_algorithm, 1.5)
cluster = fastjet.ClusterSequence(Cleaned_cands, jetdef)
ak_inclusive_jets = ak.with_name(cluster.inclusive_jets(min_pt=3),"Momentum4D")
ak_inclusive_cluster = ak.with_name(cluster.constituents(min_pt=3),"Momentum4D")

#SUEP_mult
chonkocity = ak.num(ak_inclusive_cluster, axis=2)
chonkiest_jet = ak.argsort(chonkocity, axis=1, ascending=False)
thicc_jets = ak_inclusive_jets[chonkiest_jet]

#SUEP_pt
highpt_jet = ak.argsort(ak_inclusive_jets.pt, axis=1, ascending=False)
SUEP_pt = ak_inclusive_jets[highpt_jet]
SUEP_pt_constituent = chonkocity[highpt_jet]

#Sphericity tensor
chonkiest_cands = ak_inclusive_cluster[chonkiest_jet][:,0]
mult_eigs = self.sphericity(chonkiest_cands,2.0)

highpt_cands = ak_inclusive_cluster[highpt_jet][:,0]
pt_eigs = self.sphericity(highpt_cands,2.0)
```

Credits: Chad Freer

The Scikit-HEP Ecosystem



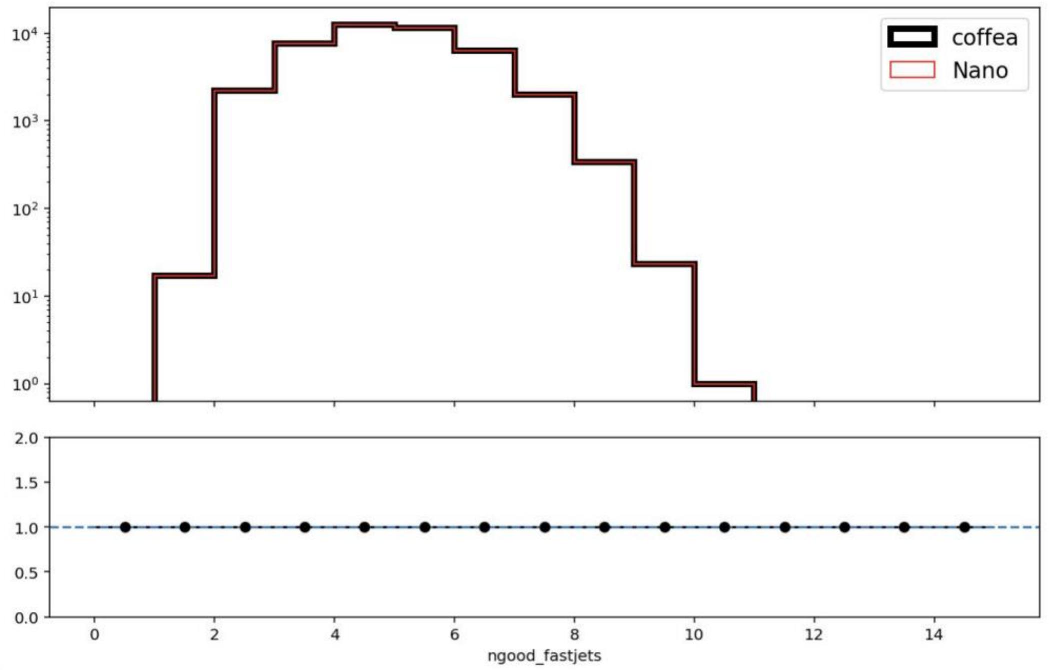
Number of fastjets

Checking the various stages with the old NanoAOD-tools framework:

- Looking at the number of large radius jets from the clustering algorithm

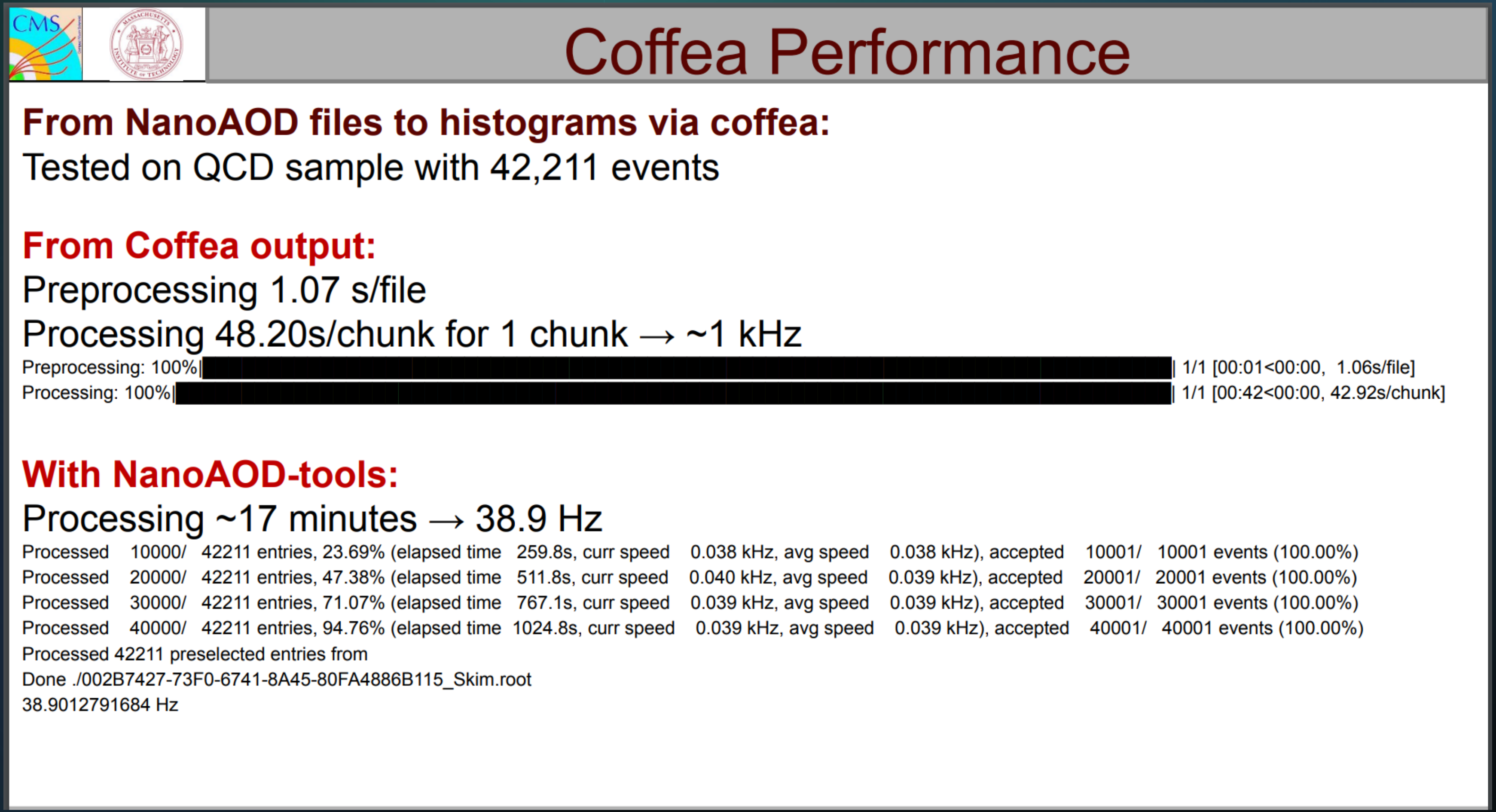
Coffea: https://github.com/SUEPPhysics/SUEPCoffea/blob/Coffea_awkward_fastjet/python/Tracks.py#L154-L158

Nano: https://github.com/SUEPPhysics/SUEPAnalysis/blob/master/python/SUEPProducer.py#L482-L486



Credits: Chad Freer

The Scikit-HEP Ecosystem



Coffea Performance

From NanoAOD files to histograms via coffea:
Tested on QCD sample with 42,211 events

From Coffea output:
Preprocessing 1.07 s/file
Processing 48.20s/chunk for 1 chunk → ~1 kHz

```
Preprocessing: 100%|██████████████████████████████████████████████████| 1/1 [00:01<00:00,  1.06s/file]
Processing: 100%|███████████████████████████████████████████████████| 1/1 [00:42<00:00, 42.92s/chunk]
```

With NanoAOD-tools:
Processing ~17 minutes → 38.9 Hz

```
Processed   10000/  42211 entries, 23.69% (elapsed time  259.8s, curr speed  0.038 kHz, avg speed  0.038 kHz), accepted  10001/  10001 events (100.00%)
Processed   20000/  42211 entries, 47.38% (elapsed time  511.8s, curr speed  0.040 kHz, avg speed  0.039 kHz), accepted  20001/  20001 events (100.00%)
Processed   30000/  42211 entries, 71.07% (elapsed time  767.1s, curr speed  0.039 kHz, avg speed  0.039 kHz), accepted  30001/  30001 events (100.00%)
Processed   40000/  42211 entries, 94.76% (elapsed time 1024.8s, curr speed  0.039 kHz, avg speed  0.039 kHz), accepted  40001/  40001 events (100.00%)
Processed 42211 preselected entries from
Done ./002B7427-73F0-6741-8A45-80FA4886B115_Skim.root
38.9012791684 Hz
```

A 20x decrease in wall time taken.

A 25x increase in throughput.

The analysis is on github: https://github.com/SUEPPhysics/SUEPCoffea_dask

Credits: Chad Freer

Conclusion

- You can **pip install fastjet** now!
- It includes the classic interface, the official Python bindings.
- The vectorized interface overloads the classic interface with event-at-a-time and multievent-at-a-time functions.
- Uses Awkward Array for arrays and Vector for coordinate systems.
- Allows interoperability with the rest of the Scikit-HEP ecosystem.
- It’s already being used in analysis.