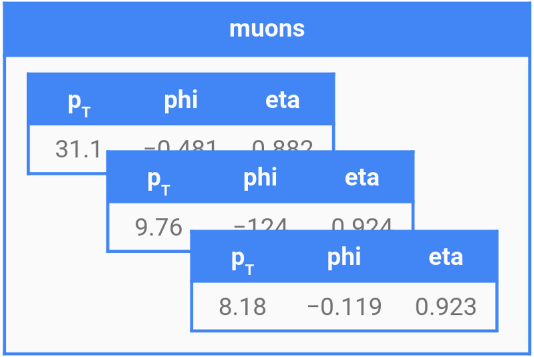
Nested data structures in array and SIMD frameworks

Jim Pivarski

Princeton University – DIANA-HEP, IRIS-HEP

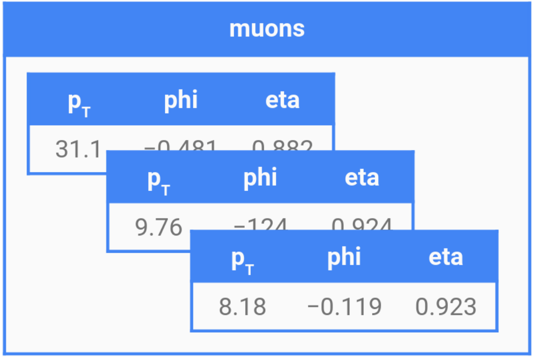
March 14, 2019

Nested, variable-sized data structures are crucial in HEP



mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

Nested, variable-sized data structures are crucial in HEP



mu1 p_T	mu1 phi	mu1 eta	mu2 p_T	mu2 phi	mu2 eta
31.1	−0.481	0.882	9.76	−0.124	0.924
5.27	1.246	−0.991	n/a	n/a	n/a
4.72	−0.207	0.953	n/a	n/a	n/a
8.59	−1.754	−0.264	8.714	0.185	0.629

Analysis datasets are big lists of variable-length lists of structs/objects/records.

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)],
 ...
```

Columnar representation

But they don't have to be "structs," pointers to contiguous p_T, η, ϕ triples.

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)],
 ...
```

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

counts	3,			1,	1,	2	
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

Columnar representation

But they don't have to be "structs," pointers to contiguous p_T, η, ϕ triples.

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)],
 ...
```

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

offsets	0,			3,	4,	5,	7
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

Columnar representation

But they don't have to be "structs," pointers to contiguous p_T, η, ϕ triples.

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)],
 ...
```

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

starts	0,			3,	4,	5	
stops	3,			4,	5,	7	
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

Columnar representation

But they don't have to be "structs," pointers to contiguous p_T, η, ϕ triples.

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)],
 ...
```

They can be contiguous by field with counts or offsets or starts/stops or parents arrays.

parents	0,	0,	0,	1,	2,	3,	3
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

This allows for efficient ways of *manipulating* data

"Remove the first muon from each event."

```
[[Muon(31.1, -0.481, 0.882), Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [Muon(5.27, 1.246, -0.991)],
 [Muon(4.72, -0.207, 0.953)],
 [Muon(8.59, -1.754, -0.264), Muon(8.714, 0.185, 0.629)],
 ...
```

"Remove the first muon from each event."

starts	0,			3,	4,	5	
stops	3,			4,	5,	7	
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

This allows for efficient ways of *manipulating* data

"Remove the first muon from each event." $\longrightarrow$ rewrite all inner lists.

```
[[                          Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [                         ],
 [                         ],
 [                            Muon(8.714, 0.185, 0.629)],
 ...
```

"Remove the first muon from each event." $\longrightarrow$ increase all starts by 1.

starts	1,			4,	5,	6	
stops	3,			4,	5,	7	
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

This allows for efficient ways of *manipulating* data

"Remove the first muon from each event." ⟶ rewrite all inner lists.

```
[[                           Muon(9.76, -0.124, 0.924), Muon(8.18, -0.119, 0.923)],
 [                         ],
 [                         ],
 [                            Muon(8.714, 0.185, 0.629)],
 ...
```

"Remove the first muon from each event." ⟶ increase all starts by 1.

starts	1,			4,	5,	6	
stops	3,			4,	5,	7	
p_T	31.1,	9.76,	8.18,	5.27,	4.72,	8.59,	8.714
phi	-0.481,	-0.123,	-0.119,	1.246,	-0.207,	-1.754,	0.185
eta	0.882,	0.924,	0.923,	-0.991,	0.953,	-0.264,	0.629

We didn't need to touch any contents (read them from disk, decompress them. . .).

Library for manipulating non-standard array types

Awkward Array

https://github.com/scikit-hep/awkward-array

- variable-length subarrays: "jagged arrays"
- struct-of-arrays viewed as array-of-structs
- nullable types
- heterogeneous types (tagged unions)
- cross-references or even cyclic references
- sparse, non-contiguous, lazy

Library for manipulating non-standard array types

Awkward Array

https://github.com/scikit-hep/awkward-array

- variable-length subarrays: "jagged arrays"
- struct-of-arrays viewed as array-of-structs
- nullable types
- heterogeneous types (tagged unions)
- cross-references or even cyclic references
- sparse, non-contiguous, lazy

Fully composable: any awkward array can be placed within any other awkward array.

Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records...

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])
```

Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records...

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])
>>> print(array)                # internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]
```

Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records. . .

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])
>>> print(array)              # internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]
>>> print(array[:, -2:])      # all of outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [None <Row 1>]]
```

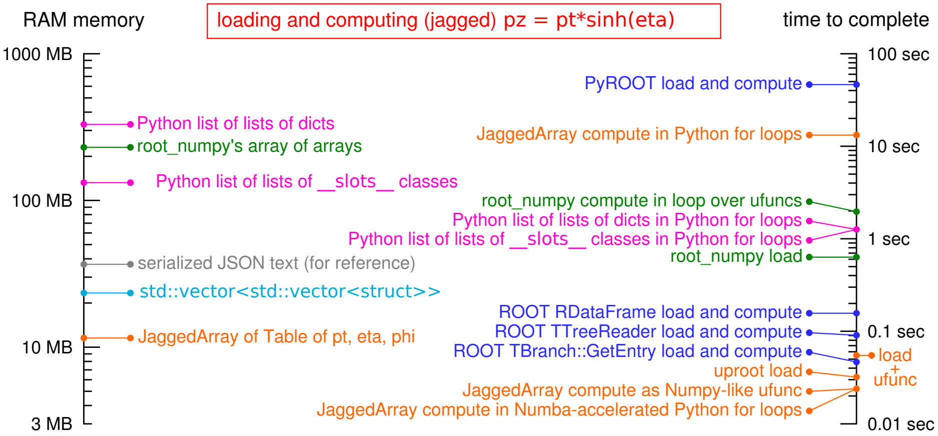
Not lacking for data types

Nullable, heterogeneous, multiple levels of depth, nested records. . .

```
>>> import awkward
>>> array = awkward.fromiter(
...     [[1.1, 2.2, None, 3.3, None],
...      [4.4, [5.5]],
...      [{"x": 6, "y": {"z": 7}}, None, {"x": 8, "y": {"z": 9}}]])

>>> print(array)                # internally, these are all arrays
[[1.1 2.2 None 3.3 None] [4.4 [5.5]] [<Row 0> None <Row 1>]]

>>> print(array[:, -2:])        # all of outer list, last two of inner
[[3.3 None] [4.4 [5.5]] [None <Row 1>]]

>>> (array + 100).tolist()      # element-wise function applied to arrays
[[101.1, 102.2, None, 103.3, None],
 [104.4, [105.5]],
 [{'x': 106, 'y': {'z': 107}}, None, {'x': 108, 'y': {'z': 109}}]]
```

Columnar data structures minimize memory use and time

Example of one operation, deriving p_z of a variable number of p_T and η per event, using awkward-array, ROOT, pure Python, and root_numpy.



a single operation $\neq$ a physics analysis

Beyond toy studies

Coffea

Columnar **O**bject **F**ramework **F**or **E**fficient **A**nalysis

Matteo Cremonesi, Lindsey Gray, Oliver Gutsche, Allison Hall, Bo Jayatilaka, Igor Mandrichenko, Kevin Pedro, Nick Smith [FNAL], and me [Princeton] `https://github.com/CoffeaTeam`

Performing two complete CMS analyses with columnar tools:

- Dark Higgs search
- Boosted SM $H \to b\bar{b}$

Also developing `fnal-column-analysis-tools`, a HEP layer on awkward-array, and a distributed query processing system with Ben Galewsky, Mark Neubauer [Illinois], and Andrew Melo [Vanderbilt].

First finished analysis: Z peak

Z peak is the "hello world" of analysis frameworks.

This implementation is realistic: run-lumi mask, pile-up correction, ID scale factors, and $ee/\mu\mu/e\mu$ channels. 350 lines in a Jupyter notebook, accessing 25 columns.

Z peak is the "hello world" of analysis frameworks.

This implementation is realistic: run-lumi mask, pile-up correction, ID scale factors, and $ee/\mu\mu/e\mu$ channels. 350 lines in a Jupyter notebook, accessing 25 columns.

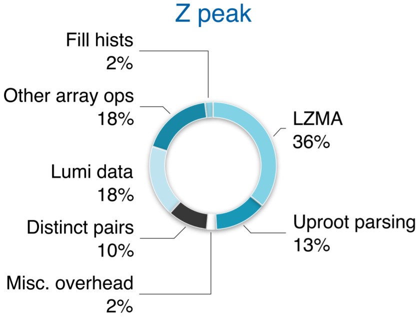
columnar analysis:

6 μs/event/thread (165 kHz)

ROOT C++:

4 μs/event/thread (250 kHz)

Columnar analysis is about 50% slower than its C++ equivalent.



Second near-complete analysis

Prototype of boosted $H \rightarrow b\bar{b}$ has

- recursive gen parent-finding
- gen-reco matching
- binned corrections
- parametric corrections
- systematics

70 μs/event/thread (14 kHz)

Uses about 100 columns.

VBF signal region definition:

```
AK4jet_AK8jet_matches = ak4_goodjets.fastmatch(leadingak8jet)
unmatched_ak4      = ak4_goodjets[~AK4jet_AK8jet_matches]
vbf_ak4_pairs      = unmatched_ak4.p4.distincts(nested=True)
vbf_ak4_detas      = np.abs(vbf_ak4_pairs.i0.eta - vbf_ak4_pairs.i1.eta).flatten()
vbf_ak4_maxdetas = vbf_ak4_detas.argmax()
vbf_ak4_masses     = (vbf_ak4_pairs.i0 + vbf_ak4_pairs.i1).mass.flatten()[vbf_ak4_maxdetas]
vbf_ak4_pass       = (vbf_ak4_detas[vbf_ak4_maxdetas] > 3.25) & (vbf_ak4_masses > 975.0)
```

Leading fat-jet selection with vetos:

```
k8puppijet_pt200 = ak8puppijet[
                       passLooseJetSel(ak8puppijet) &
                       (ak8puppijet.pt > 200) &
                       (np.abs(ak8puppijet.eta) < 2.5)]
ak8veto = ~(
    ak8puppijet_pt200.fastmatch(vetoMuons,deltaRCut=0.4) |
    ak8puppijet_pt200.fastmatch(vetoElectrons,deltaRCut=0.4) |
    ak8puppijet_pt200.fastmatch(vetoPhotons,deltaRCut=0.4) )
ak8jets_veto = ak8puppijet_pt200[ak8veto]
leadingak8jet = ak8jets_veto[ak8jets_veto.pt.argmax()]
```

Syntax is an extension of Numpy

Numpy arrays must all be rectangular: vectors, matrices, and tensors. Awkward arrays reproduce this behavior in rectangular cases and generalize in jagged cases.

- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`
- Masking (list compaction): `data[trigger & (pt > 40)]`
- Fancy indexing (gather/scatter): `all_eta[argsort(all_pt)]`
- Row/column commutativity (hides AoS $\leftrightarrow$ SoA): `table["column"][7]` (row 7 of column array)
  `table[7]["column"]` (field of row tuple 7)
- Array reduction: `array.sum()` $\rightarrow$ scalar

Syntax is an extension of Numpy

Numpy arrays must all be rectangular: vectors, matrices, and tensors. Awkward arrays reproduce this behavior in rectangular cases and generalize in jagged cases.

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep jagged structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation
- Masking (list compaction): `data[trigger]` → drop whole events
  `data[jetpt > 40]` → drop jets from events
- Fancy indexing (gather/scatter): `a = argmax(jetpt)` → `[[2], [], [1], [4]]`
  `jeteta[a]` → `[[3.6], [], [-1.2], [0.4]]`
- Row/column commutativity (hides AoS ↔ SoA): `events["jets"]["pt"][7, 1]`,
  `events["jets"][7]["pt"][1]`,
  `events[7]["jets"]["pt"][1]`, ...
- Jagged array reduction: `jetpt.max()` → array of max jet p_T per event

Structure for most physics data: multiple candidates per event

```
JaggedArray(ObjectArray(Table(px, py, pz, E), LorentzVector))
```

Structure for most physics data: multiple candidates per event

```
JaggedArray(ObjectArray(Table(px, py, pz, E), LorentzVector))
```

- `px`, `py`, `pz`, `E` are flat Numpy arrays (all particles, all events).
- `Table` to present contiguous columns as an array of rows
- `ObjectArray` to interpret rows of the `Table` as `LorentzVector` objects
- `JaggedArray` because there's a variable number of `LorentzVectors` per event

Structure for most physics data: multiple candidates per event

```
JaggedArray(ObjectArray(Table(px, py, pz, E), LorentzVector))
```

- `px`, `py`, `pz`, `E` are flat Numpy arrays (all particles, all events).
- `Table` to present contiguous columns as an array of rows
- `ObjectArray` to interpret rows of the `Table` as `LorentzVector` objects
- `JaggedArray` because there's a variable number of `LorentzVectors` per event

Individual `LorentzVector` objects have kinematic methods—`pt`, `eta`, `mass`, etc.— but so do the `ObjectArray` and `JaggedArray`. Whole-array methods are vectorized.

To compute the mass of all particles in all events are pack it into per-event sublists, you say

```
>>> particles.mass
```

Putting it all together: a simple Z peak

```
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")
```

Putting it all together: a simple Z peak

```
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array                                         # muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
               TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>
```

Putting it all together: a simple Z peak

```
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array                                   # muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
               TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>

>>> array[0, 1]                             # second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)
```

Putting it all together: a simple Z peak

```
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array                                        # muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
               TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>

>>> array[0, 1]                                  # second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)

>>> hastwo = (array.counts >= 2)                 # to select at least two muons
>>> leading = array[hastwo, 0]                   # mask and select first
>>> subleading = array[hastwo, 1]                # mask and select second
```

Putting it all together: a simple Z peak

```
>>> import uproot
>>> dataset = uproot.open("HZZ-objects.root")["events"]
>>> array = dataset.array("muonp4")

>>> array                                      # muons for all events
<JaggedArray [[TLorentzVector(-52.899, -11.655, -8.1608, 54.779)
               TLorentzVector(37.738, 0.69347, -11.308, 39.402)] ...]>

>>> array[0, 1]                                # second muon in first event
TLorentzVector(37.738, 0.69347, -11.308, 39.402)

>>> hastwo = (array.counts >= 2)               # to select at least two muons
>>> leading = array[hastwo, 0]                 # mask and select first
>>> subleading = array[hastwo, 1]              # mask and select second

>>> candidates = leading + subleading          # Lorentz vector sum across all
>>> candidates.mass                            # compute mass for all
array([90.22779777, 74.74654928, ..., 85.44384208, 75.96066262])
```

Feedback from physicists

- Many bug-fixes, of course.

Feedback from physicists

- Many bug-fixes, of course.
- `argmin()` was an analysis bottleneck. Reimplemented to be $125\times$ faster.

Feedback from physicists

- Many bug-fixes, of course.
- `argmin()` was an analysis bottleneck. Reimplemented to be 125× faster.
- Validity checks (e.g. to ensure that all `starts`/`stops` are within `content`) are expensive. Need to avoid redundant checks without removing safety.

Feedback from physicists

- Many bug-fixes, of course.
- `argmin()` was an analysis bottleneck. Reimplemented to be $125\times$ faster.
- Validity checks (e.g. to ensure that all `starts`/`stops` are within `content`) are expensive. Need to avoid redundant checks without removing safety.
- Discovering which operations are frequently used in analysis, which aren't.

Feedback from physicists

- Many bug-fixes, of course.
- `argmin()` was an analysis bottleneck. Reimplemented to be 125× faster.
- Validity checks (e.g. to ensure that all `starts`/`stops` are within `content`) are expensive. Need to avoid redundant checks without removing safety.
- Discovering which operations are frequently used in analysis, which aren't.
- Interesting mistake: analysts must unlearn order-dependent coding habits!

```
(nMuons > 0) & (Muons_pt[:, 0] > 30) # intersection of masks
```

The latter might try to access the first of zero muons.

Feedback from physicists

- Many bug-fixes, of course.
- `argmin()` was an analysis bottleneck. Reimplemented to be 125× faster.
- Validity checks (e.g. to ensure that all `starts`/`stops` are within `content`) are expensive. Need to avoid redundant checks without removing safety.
- Discovering which operations are frequently used in analysis, which aren't.
- Interesting mistake: analysts must unlearn order-dependent coding habits!

  ```
  (nMuons > 0) & (Muons_pt[:, 0] > 30)  # intersection of masks
  ```

  The latter might try to access the first of zero muons.

  Instead,

  ```
  Muons_pt[(nMuons > 0), 0] > 30         # mask first dim, pick 0
  ```

Is this making analysis easier?

If yes, great! Continue developing array operations, thinking about their "ergonomics," and optimize their implementations.

If no, it's still a useful abstraction layer, but we'll need a more user-friendly interface on top of it, such as a functional or declarative language.

Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

Some motivated by execution speed, some by ease of use.

- "Thirty minutes is too long to wait for a plot."
- "Will be run order-of-a-hundred times over the course of the year; this is a big investment." but "For something that could be two times faster, I wouldn't do these optimizations."
- "Ease of use is paramount; I've always struggled with poorly written code." and "Making it fast to run it again and again is *going around* ease of use."
- "Ease of use is most important, even if execution speed decreases."

Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

Some found it easier, some more difficult.

- "Way, way much easier than applying cuts with for loops."
- "Surprised by how conceptually different you have to think about selections, combining objects." but "Not good or bad, just surprising that it has a learning curve."
- "Individual problems have been much more difficult than expected." and
  "Translating 'if' statements is where I get hung up." but
  "Not inherently harder; just harder now for those of us used to the 'for' loop version."

Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

One point came up multiple times: easier to read than write.

- “The good thing is, once you figure it out, it’s clear why it works. It’s not magic, you just have to get the mapping right.”
- “If I ask a student to read my code, he’ll be able to read it. But five minutes later, he’ll try something similar and it won’t work.”

Half-hour interviews with physicists about array syntax

1 grad student, 2 postdocs (beginning & advanced), and 1 advanced researcher

Everyone had most experience in C++ (5 years to decades), less in Python, which was primarily PyROOT (6 months to 3–4 years), very little in Numpy (2 to 5 months).

One point came up multiple times: easier to read than write.

- "The good thing is, once you figure it out, it's clear why it works. It's not magic, you just have to get the mapping right."
- "If I ask a student to read my code, he'll be able to read it. But five minutes later, he'll try something similar and it won't work."

~~Write-only~~ Read-only code???

Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are "statically typed at runtime," so I'm extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), imperative (for-loop style) calculations in Python.

Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are "statically typed at runtime," so I'm extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), imperative (for-loop style) calculations in Python.

- Possible Google Summer of Code project to add precompiled and/or CUDA implementations, depending on the abilities and interests of the student.

Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are "statically typed at runtime," so I'm extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), imperative (for-loop style) calculations in Python.

- Possible Google Summer of Code project to add precompiled and/or CUDA implementations, depending on the abilities and interests of the student.

- Michael Hedges (Purdue) is developing Pandas extensions, so that DataFrame columns can contain and operate on jagged data.

Further developments

- Numba is a JIT-compiler for Python, but only for statically typed data. Awkward-array types are "statically typed at runtime," so I'm extending Numba to recognize and JIT-compile them.

  This will permit fast (compiled), imperative (for-loop style) calculations in Python.

- Possible Google Summer of Code project to add precompiled and/or CUDA implementations, depending on the abilities and interests of the student.

- Michael Hedges (Purdue) is developing Pandas extensions, so that DataFrame columns can contain and operate on jagged data.

- Giuseppe Cerati (Fermilab) is investigating the use of jagged arrays in C++, to write reconstruction algorithms that are equally efficient on CPUs and GPUs.

  Status: implemented track-propagation in Python and switched from CPU to GPU with from "`import numpy as np`" → "`import cupy as np`".

  Reimplemented in C++ using xtensor (Numpy clone for C++).

Conclusions

Awkward-array is a library for complex data, presented as arrays. Jagged arrays are the most important for HEP. (Perhaps the only type *necessary* for HEP?)

We're beyond single-operation tests; we're implementing complete analyses. Performance is within a factor of two of C++, and there's low-hanging fruit for improvements.

Physicists find it hard to write, but easy to read.

This is an open area of development with many paths to follow!