`hep_tables`: Heterogeneous Array Programming for HEP[*]

Gordon Watts[1,**]

[1]University of Washington, Seattle

Abstract. Array operations are one of the most concise ways of expressing common filtering and simple aggregation operations that are the hallmark of a particle physics analysis: selection, filtering, basic vector operations, and filling histograms. The High Luminosity run of the Large Hadron Collider (HL-LHC), scheduled to start in 2026, will require physicists to regularly skim datasets that are over a PB in size, and repeatedly run over datasets that are 100's of TB's – too big to fit in memory. Declarative programming techniques are a way of separating the intent of the physicist from the mechanics of finding the data and using distributed computing to process and make histograms. This paper describes a library that implements a declarative distributed framework based on array programming. This prototype library provides a framework for different sub-systems to cooperate in producing plots via plug-in's. This prototype has a `ServiceX` data-delivery sub-system and an `awkward` array sub-system cooperating to generate requested data or plots. The `ServiceX` system runs against ATLAS `xAOD` data and flat `ROOT TTree`'s and `awkward` on the columnar data produced by `ServiceX`.

1 Introduction

A particle physicist uses a number of heterogeneous systems to make a plot [1]. A typical workflow might start with reconstructed data - data that contains physics objects - located in an experiment's production storage system. The output of the experiment's production system is often located on the GRID, a loosely federated collection of CPU and disk facilities. The physicist must submit a job to run on the GRID to access that data. This job extracts the data, applies corrections, and writes out the data in a simplified format, mostly likely ROOT `TTree`'s. The physicist then downloads the data locally, and uses local tools to run over that data to extract plots. Time scales can be long: while the jobs on the GRID will normally run quickly, GRID site down times, data delivery glitches, etc., often mean there is a long trail when the user runs on 100's of files. For many analyses this phase takes more than a week. Once the data is stored locally, it can take approximately a day to produce a plot. However, if the physicist decides that they need a new quantity from the original production data, the complete cycle must be repeated. For this reason, physicists tend to extract a large amount of data, making the local data sets 10's to 100's of TB.

A second aspect makes this harder than, perhaps, it needs to be: the physicist must know multiple programming languages and tools: C++ and Python for the GRID jobs, along with command line tools and scripts to submit and babysit the jobs. For making the plots one has

[*]This work was supported by the National Science Foundation under Cooperative Agreement OAC-1836650.
[**]e-mail: gwatts@uw.edu

to use either C++ and ROOT or Python and the SciKit-HEP set of libraries. Further, one has to have more than a passing familiarity with the complexity of distributed computing systems, along with how well the weakest GRID site is working so that it can be avoided.

The field of database design and management offers two concepts that apply for HEP data analysis: data query languages and data independence [2]. SQL, a common database query language, is declarative: it only specifies the relationships between data fields. Data independence is the idea that how the data is stored should not impact how one queries the data: the same SQL can be used to query many different databases with different storage schema.

The Python ecosystem has developed a compelling tool-set that uses array-processing semantics to quickly and intuitively analyze data that can be loaded as in-memory arrays. In particle physics, the `awkward` array library [3] is the most popular of these libraries, and it borrows and extends semantics from the famous rectangular `numpy` [4] array processing library to work with non-uniform multi-dimensional arrays (e.g. `JaggedArray`'s). This project explores using an array-like interface to take over not only the last step in the plot production process, but also the part that uses the GRID. As an example, the following code will make a plot of electron p_T starting from a set of GRID files that are the output of the ATLAS Monte Carlo production:

```
 1  from func_adl import EventDataset
 2  from hep_tables import xaod_table, make_local
 3
 4  dataset = EventDataset('localds://mc15_13TeV:mc15_13TeV.361106.
      ↪ PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.DAOD_STDM3.
      ↪ e3601_s2576_s2132_r6630_r6264_p2363_tid05630052_00')
 5  df = xaod_table(dataset)
 6
 7  eles = df.Electrons("Electrons")
 8  good_eles = eles[(eles.pt > 50000.0) & (abs(eles.eta) < 1.5)]
 9
10  np_pts = make_local(good_eles.pt/1000.0)
11
12  plt.hist(np_pts.flatten(), range=(0, 100), bins=50)
13  plt.xlabel('Electron $p_T$ [GeV]')
14  plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
```

Listing 1: A complete `hep_tables` code snippet that extracts the electron p_T from an ATLAS `xAOD` Monte Carlo sample, brings it to the local machine, and uses `matplotlib` to plot it as a histogram.

Line 4 declares the GRID dataset to be used, lines 7-8 define a good electron as being central and $p_T > 50000$ MeV (the units in an ATLAS `xAOD` file). Line 10 triggers the rendering of the data, which involves calling out to the `ServiceX` [5] backend to fetch the data. It is returned as an awkward array, which is plotted in lines 12-14. An interesting subtlety, which is not much further discussed in this paper, can be found in line 12: note the call to `np_pts.flatten()`. The `awkward` data returned this `hep_tables` query is jagged - an array of arrays. The first array has an entry per event, and the inner array is of all electron p_T's in the event. `hep_tables` does its best to allow you to work on a per-event basis, though most operations are done on large blocks of events at once via array programming. Both a query language and data independence are present here: the array-programming interface is the query language, and as the plug-in's translate the query into the proper instructions, the query language is independent of the way the data is stored.

2 Library Structure

This library provides a user-interface that looks much like `numpy` or `awkward`. Instead of immediately executing the operations, however, it records them. When the physicist requests the actual plot or other result, that triggers the library to resolve the query into a set of actions via plug-ins. Each plug-in uses a sub-system to execute the requested actions. The end-result is a number, or a column of data (rendering of plots is planned).

The library is built in two layers. The first layer is responsible for recording the actions in the code. The second layer is responsible for executing them. The most popular libraries, like `numpy` and `awkward`, do both these operations at execution time: an expression like $x[0] + y[0]$ is executed as it is evaluated. Two advantages are gained by splitting this operation into recording and execution. First, the operations can be fused - so the addition and the array slicing can take place at the same time. This avoids the creation of temporaries. This is done by analyzing the recording and looking for operations that can be fused. Second, the recording can easily be efficiently shipped to a remote system where it is executed and only the results transmitted back. Or the execution could be split across multiple systems, without the physicist having to track what is going on.

2.1 The Interface

There are standards for array programming in the Python eco-system. Two in particular. For rectangular arrays, `numpy` is the standard, and for jagged arrays, `awkward` is the standard. A huge amount of effort has gone into both interfaces and it would be counter productive to alter them. It might be worth extending them: there are certain types of queries that are difficult to encode, for example, object-matching using these interfaces.

numpy

The original array programming interface. Python has standardized some of the low-level interface work in a PEP [6] (Python language extension). Defines basic operators, slicing, and accumulation functions. Designed specifically for rectangular data.

`numpy` operations, of course, are expected to be performed on `numpy` arrays. In the case of a project like `hep_tables`, one really wants to record the operations for execution later. `numpy` provides an interface, `__array_function__`, which is designed for this [7]. As a result, a call to `numpy.abs(a)` is turned into a call to `a. __array_function__`. The object `a` can then record the fact that `numpy.abs` was called on it.

awkward

HEP data is not rectangular, of course, and `awkward` was written to extend `numpy` to support more complex arrays, called `JaggedArray`'s. Besides defining slicing across non-uniform array runs, it also defines operations that work across array axes, for example, counting the number of jets in an event to yield a number-of-jets-per-event column. `awkward` also provides other conveniences like behaviors, which allow an end user to imbue an array with additional behaviors (a 4-vector can be given Lorentz-like semantics).

Improving On These Interfaces

While almost any operation desired is possible with these interfaces, there are places where it is quite difficult to encode them - or that they are not at all straight forward. Consider the following code snippet that matches a Monte Carlo particle to a jet using $\eta - \phi$ matching:

```python
result = []
for e in electron:
  min_value = 0.1
  particle = None
  for t in truth:
    if dr(e,t) < min_value:
      min_value = dr(e,t)
      particle = t
  result.append(particle)
```

A particle physicist will look at this and know almost immediately what is being done: find the closest MC particle to the electron. Array programming, however, is not nearly as straight forward one must operate between two arrays and the operation contains a reduction (from the list of truth particles to a single truth particle).

2.2 Dataframe Expressions

There is no Python library that will tape-record a set of array operations. There are many libraries that implement `numpy` semantics and are drop-in replacements [8–10]. However, their re-implementation of the `numpy` semantics is tightly tied to their implementation.

`dataframe_expressions` is built to be a standalone library that simply records the operations made to `DataFrame`. The library supplies an abstract `DataFrame` type; and then records all operations done to it. It makes as few assumptions about the underlying data model as possible. Operations occur on the `DataFrame` type, and include:

- The basic unary and binary math operations (e.g. $-a$, $a + b$, a/b, etc.)
- A number of Python built-in functions (e.g. $abs(a)$)
- Array leaf references and functions (e.g. $a.jets$ and $a.jets.count()$)
- numpy type array references (e.g. $numpy.sin(a)$)
- Filtering, or array slicing (e.g. `a.jets[a.jets.pt > 30]` or `a.jets[abs(a.jets.eta)<2.4)`

The recording is done with a combination of a linked-tree structure and a Python abstract syntax trees (AST). The Python AST is rich - the complete Python language can be expressed in it (by design). Each `dataframe_expressions` `DataFrame` object holds a link to a Python AST. If a and b are `DataFrame` objects, then `a+b` becomes a new `DataFrame` object, which contains an AST. The AST is the binary `+` operator, with reference to the `a` and `b` `DataFrame`'s. All expressions are encoded in this way, building a tree of operations.

To use the `dataframe_expressions` `DataFrame`, a library creates the base `DataFrame`. Usually this means defining the source data (a file, etc.). The user then manipulates the `DataFrame`'s, finishing with some sort of a call to render the results of the expression, like the `make_local` call in the first listing above. At that point, the library takes the `DataFrame` that represents the final result and can unwind the user's intent, executing it as required.

One advantage of separating the recording of user intent and the rendering of the expression is `dataframe_expressions` `DataFrame` can provide some short cuts. These short cuts then do not need to be directly handled by the back-end library. Filter functions are one example:

```python
def good_jet(j):
    return (j.pt > 35) & (abs(j.eta) < 2.5) & j.isGood

good_jets = a[good_jet]
```

Functions like this are helpful when one needs to define a `good_jet` in multiple places. Behind the scenes, `dataframe_expressions` just calls the lambda with a `DataFrame` that represents the jet, and records the operations performed on it.

Python `lambda` functions are also supported. For example, `a.jets.pt/1000.0` and `a.jets.map(lambda j: j.pt/1000.0)` resolve to identical array expressions (given the interpretation library applies `map` as you'd expect here). Python `lambda` functions can be used in most places in the code, but their power comes from lambda capture - where a variable declared in the outer lambda is captured by an inner lambda. For example, calculating the ΔR between a jet and electrons: `a.jets.map(lambda j: a.electrons.map(lambda e: j.DeltaR(e)))`. This creates a 2D `JaggedArray` for each event. The first axis is the jet, and the second axis is the ΔR between that row's jet and each electron in the event. It is now possible to find the electron closest to each jet using sorting, or arg-sorting, or indexing.

It is also possible to add new expressions anywhere in the data model. If an analysis group wants to define `good_jet`'s for everyone, in a separate imported Python module file it would be possible to write:

```
def define_shortcuts(df):
    df['good_jets'] = df.jets[good_jet]

define_shortcuts(a)
good_pts = make_local(a.good_jets.pt)
```

This provides some level of composability - as you can easily chain complex expressions, and still conveniently define them inside a function without having to return lots of different values. For example, one could define a new property of a jet, which was the closest Monte Carlo parton, using $\eta - \phi$ matching.

One particular place this proves useful is defining a macro at one level in the tree, but using it at another. For example:

```
a.jets['ptgev'] = a.jets.pt / 1000.0

jetpt_in_gev = a.jets[a.jets.ptgev > 30].ptgev
```

Note that the `ptgev` is defined before the filter. But used after the filter. It is intuitive that this should work, of course, but technically it turned out to be non-trivial. Any value, like `pt`, can also be referred to as a string lookup - which helps a great deal in simplifying automated histogram and plot generation: `a.jets['pt']` does exactly what you'd expect it to do. `dataframe_expressions` translates both this lookup and `a.jets.pt` into the same AST so the backend library does not need to handle complex features like this.

2.3 HEP Tables

While `dataframe_expressions` provides the user interface for the package, `hep_tables` is the implementation. Its job is to take a `DataFrame` and to render it with the data producing a histogram or other final product.

All `dataframe_expressions` start with an initial `DataFrame` object. `hep_tables` defines a single root `DataFrame` which currently refers to an ATLAS `xAOD` dataset (`xaod_table`). These datasets are registered on the GRID; despite being stored around the world, the dataset names can be thought of a unified namespace. Lines 4-5 in Listing 1 do this. The fact that this is restricted to `func_adl`'s `EventDataset` is an artifact of the current implementation (see Section 4). The `xaod_table` is directly derived from a `dataframe_expressions` `DataFrame`.

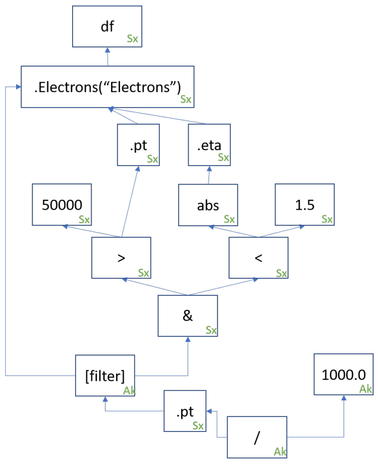

Figure 1. The `DataFrame` tree recorded by the calculation of the good electron p_T. The text in each node's lower right indicates which backend plug-in will handle that node's calculation.

Until the `make_local` function call in line 10, everything is handled by the `dataframe_expressions` package: no actual execution is scheduled. The `make_local` triggers the execution. The `hep_tables` package uses utilities in the `dataframe_expressions` package to build the full AST that summarizes the calculation. It then uses a plug-in system to create a plan for execution.

Currently a `ServiceX` and an `awkward` plug-in exist. Each plug-in can look at a single level in the AST and decide if it can execute it (given it can execute all parents). `hep_tables` starts at the deepest parts of the tree and finds a backend that can execute each node of the AST. It does its best to use the executor that could handle all the parents. When the executor switches, `hep_tables` creates a Python `async` task to perform the execution via the selected plug-in. The plug-in's are expected to render `awkward` arrays, and the different plug-in's use these to move data between them.

Figure 1 shows the AST that is produced by the `dataframe_expressions` starting from the `xaod_table` defined by the `hep_tables` package (see Listing 1). Each operation or element is a new `DataFrame`. `hep_tables` has walked the tree, labeling each node with a plug-in. When `hep_tables` gets to the bottom of the tree (the filter and division), it will submit to `ServiceX` a request to render the data from the & and `.pt` nodes, using the `awkward` backend to combine them.

`ServiceX`

The `ServiceX` plug-in works by translating the AST tree to `func_adl`, and submitting the query via the `ServiceX` web API. The result is returned as `awkward` arrays.

Translation to `func_adl` for an expression tree like above requires some finesse. The tree contains multiple references to the electron branch, and those that are separated by a distance, the `.pt` and the `[filter]` nodes, require an extra carry-along in the `func_adl` expression.

awkward

The awkward plug-in is an immediate translator: it executes each AST node, one at a time. It does not try to take advantage of operation fusing. In that sense, this is a direct translation of awkward operations (which also made it very easy to write).

3 Capabilities and an Example

The capabilities of hep_tables are, to first order, defined by the backend plug-in's. This section notes a few capabilities that are implemented:

- map: The map function is implemented (see Section 2.2). It takes whatever is to the left as a collection and loops over it. The exact operation depends on what is to the left and the **lambda** function, of course.
- Backend-Functions: There are often utility functions declared in the backend. hep_tables needs to know about the existence of these, so they must be declared up front. The DeltaR function is an example of this. Any function can be declared, and there is the possibility of shipping arbitrary C++ down to run on the xAOD/ServiceX backend as well, though that is not currently implemented.
- First and Count and similar functions: a.First() will return the first element of an array, and a.Count() returns number of items in an array. Both of these turn an array into a scalar. Various other *aggregation* functions are also supported.
- Types are tracked through the expression, though heuristics are used at the start. For example, collections like Electrons are hard-coded into an xAOD data model. But once the collection is accessed, its type is tracked. The leaves, like .pt, are assumed to be a double.

The following more complex example demonstrates some of these capabilities, and some of the short-comings of the current design. This example looks at reconstructed electrons from an xAOD file, and plots the matching Monte Carlo electron. The match is defined as an electron that is within $\Delta R < 0.1$.

```
from hep_tables import xaod_table, make_local, curry
from dataframe_expressions import user_func, define_alias
from func_adl import EventDataset
import matplotlib.pyplot as plt
import numpy as np

@user_func
def DeltaR(p1_eta: float, p1_phi: float, p2_eta: float, p2_phi: float) ->
↪ float:
    # Calculate DeltaR between two eta/phi combinations. Implemented in the
    ↪ backend.
    assert False, 'This should never be called'

dataset = EventDataset('localds://mc15_13TeV:mc15_13TeV.361106.⌋
↪ PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.DAOD_STDM3.⌋
↪ e3601_s2576_s2132_r6630_r6264_p2363_tid05630052_00')
df = xaod_table(dataset)

mc_part = df.TruthParticles('TruthParticles')
mc_ele = mc_part[(mc_part.pdgId == 11) | (mc_part.pdgId == -11)]

eles = df.Electrons('Electrons')

def good_e(e):
```

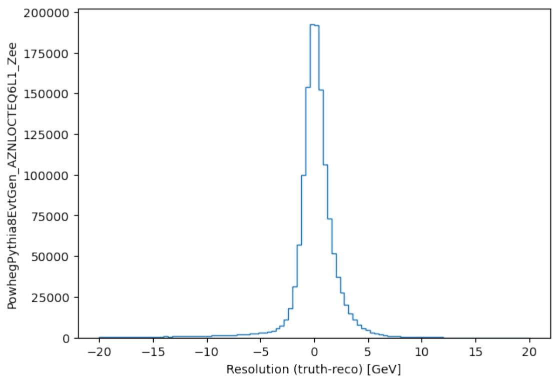

Figure 2. The difference in energy between the reconstructed electron and the MC electron in Monte Carlo

```
21        'Good electron particle'
22        return (e.ptgev > 20) & (abs(e.eta) < 1.4)
23
24    good_eles = eles[good_e]
25    good_mc_ele = mc_ele[good_e]
26
27    def associate_particles(source, pick_from):
28        def dr(p1, p2):
29            'short hand for calculating DR between two particles.'
30            return DeltaR(p1.eta(), p1.phi(), p2.eta(), p2.phi())
31
32        def very_near(picks, p):
33            'Return all particles in picks that are DR less than 0.1 from p'
34            return picks[lambda ps: dr(ps, p) < 0.1]
35
36        source[f'all'] = lambda source_p: very_near(pick_from, source_p)
37
38        source[f'has_match'] = lambda e: e.all.Count() > 0
39        with_assoc = source[source.has_match]
40        with_assoc['mc'] = lambda e: e.all.First()
41
42        return with_assoc
43
44    matched = associate_particles(good_eles, good_mc_ele)
45
46    pt_matched_mc = make_local(matched.mc.ptgev)
47    pt_matched_reco = make_local(matched.ptgev)
48
49    plt.hist((pt_matched_mc-pt_matched_reco).flatten(), bins=100, range=(-20, 20),
       ↪  histtype='step')
50    plt.ylabel('PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee')
51    plt.xlabel("Resolution (truth-reco) [GeV]")
```

The result of the plot in line 49 is shown in Figure 2. Some of the code is familiar from Listing 1. Lines 7-10 declare a backend function. In this particular case, that will be executed on `ServiceX`. It is used in the function `dr` defined at line 28. Lines 24 and 25 define good MC and Data electrons that we'll use in comparison. The function `associate_particles` defined at line 27 associates the `source` (electrons) to the `pick_from` (MC electrons). This is done using the `very_near` function defined at line 32 that returns a list of MC electrons that are within

$\Delta R < 0.1$. Note the lambda capture grabs the electron in the form of the variable p that is passed in. Not all electrons have a match, hence the has_match being defined at line 38, and using it to protect the matched particle definition in line 40. Note that lines 38-40 extend the data model so data electrons now have the bool has_match and an Monte Carlo particle mc. These can be referred to, as they are in line 46, for easy difference plotting.

As a quick last observation - note the mix between per-event and array programming here. The signature of DeltaR is clearly per-electron, but where it is called in line 30 is array programming. The system, with the logic of ServiceX and awkward, allows this shift to occur naturally. It isn't perfect - the abstraction does leak - for example if DeltaR can't be called at the ServiceX level.

4 Status

A prototype exists in GitHub and is working [11]. This prototype was used to develop the concepts in the code. It dispatches code to run against ServiceX and awkward as mentioned earlier. There are, as described, clear short-comings with the code. At the time of writing, a new version is being created to fix a number of the architectural mistakes that were made during prototyping.

- The ServiceX backend is being upgraded to handle the full range of supported func_adl queries to reduce the amount of data that needs to be shipped from ServiceX to an analysis cluster. For example, the ServiceX sub-system could run all nodes found in Figure 1, but due to limitations in the hep_tables plug-in, it cannot.
- A backend that supports the coffea library is being created. This includes the ability to run processors in multiple places and support for the coffea data model (called nanoaod). One of the biggest benefits here will be the ability to run on clusters. This will replace the awkward plug-in, as awkward is used by coffea.
- A full type system is being implemented. If known, this means the system will not have to infer types, and reduce mistakes. It will also mean errors are flagged earlier (e.g. accessing data that doesn't exist).
- As the user's intent is known at the start, it is possible to cache the users query. Thus the second time the same query is run it should take very little time - just a lookup. This is currently implemented by the ServiceX backend. We are investigating adding this more generically to the system.
- Histogram definition and filling is being implemented as something understood by dataframe_expressions. This will allow histograms (or other similar objects) to be filled in parallel and then combined.

The approach hep_tables takes, like many in Python, does have some issues. Some are a function of this implementation and some are a function of Python's programming model.

- Assumptions are made about which level an array operation occurs. For example, does a.jets.Count() count the number of jets in each event (returning an array), or count the number of events (returning a scalar)?
- While lambda capture enables many important patterns, and makes it more clear what is going on, its readability is not the most straight forward. This is a generic problem with Python: it is not yet possible to analyze the source code of a Python pragmatically in a robust way. See the double-loop example in Section 2.1 for an example of code that is hard to translate.

- It isn't obvious how functions over sequences should work when more than one sequence is involved. With a single sequence it is straight forward - for example `abs(a.jets.pt)` should return an array of the absolute value of jet pt's. But what about the tuple `(abs(a.jets.pt), abs(a.jets.pt))`? Should that return a single array, with two entries in each? Or should it return a 2D array: it isn't obvious what the physicists intent for the implied loops are here.
- Many common libraries, like `seaborn`, are used for plotting. This is temptingly close to being able to use these libraries. However, these libraries usually require the data in-memory, and that can be quite expensive for a large analysis dataset.
- Loop algorithms are not well suited to this style of declarative programming. For example, track finding - where you don't have a definite number of steps.
- Control logic is not captured - if a decision has to be made on the result of a query, the query must be rendered first. As one thinks forward to adopting differentiable programming, as the `JAX` package [12], this problem will have to be better understood.

5 Conclusions

This paper has described the `dataframe_expressions` and `hep_tables` package. The former records the user's intent by tracking common and extended array operations. The latter renders the actions across multiple backends. This project was started as a prototype to understand if something like this was possible, without having to re-write the complete eco-system. As both `numpy` and `awkward` use a dispatch mechanism, this turns out to be possible.

The packages have been used for some simple analysis examples, and are currently undergoing a re-write to make them robust enough to be used for a more sophisticated analysis.

References

[1] R.W.L. Jones, D. Barberis, J. Phys. Conf. Ser. **219**, 072037 (2010)

[2] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases: The Logical Level*, 1st edn. (Addison-Wesley Longman Publishing Co., Inc., USA, 1995), ISBN 0201537710

[3] J. Pivarsk, J. Nandi, D. Lange, P. Elmer, EPJ Web Conf. **214**, 06026 (2019)

[4] C.R. Harris et al., Nature **585**, 357 (2020), `2006.10256`

[5] B. Galewsky, R. Gardner, L. Gray, M. Neubauer, J. Pivarski, M. Proffitt, I. Vukotic, G. Watts, M. Weinberg, EPJ Web Conf. **245**, 04043 (2020)

[6] T. Oliphant, C. Banks, *Homebrew*, https://www.python.org/dev/peps/pep-3118/ (2020), accessed: 2020-02-28

[7] *The numpy array interface*, https://numpy.org/devdocs/reference/arrays.interface.html (2020), accessed: 2020-02-28

[8] Dask Development Team, *Dask: Library for dynamic task scheduling* (2016)

[9] *Ray - a simple, universal API for building distributed applications* (2020), accessed: 2020-02-28

[10] *Vaex - Out-of-Core DataFrames for Python* (2020), accessed: 2020-02-28

[11] *The `hep_tables` GitHub repository*, https://github.com/gordonwatts/hep_tables (2021), accessed: 2021-06-08

[12] S.S. Schoenholz, E.D. Cubuk, *JAX M.D. A Framework for Differentiable Physics*, in *Advances in Neural Information Processing Systems* (Curran Associates, Inc., 2020), Vol. 33, `https://papers.nips.cc/paper/2020/file/83d3d4b6c9579515e1679aca8cbc8033-Paper.pdf`