uproot

Rapidly moving data from ROOT to Numpy and Pandas

Jim Pivarski

Princeton University – DIANA-HEP

February 28, 2018

The what and why of uproot

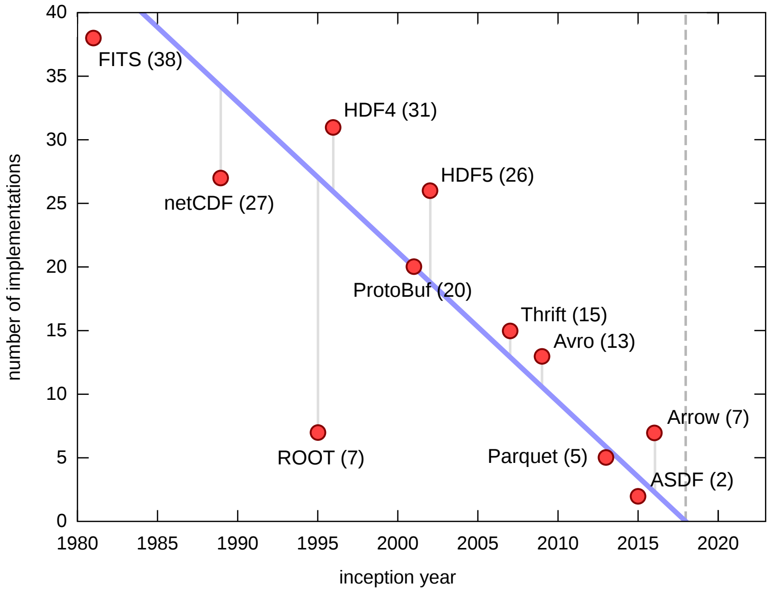
What is uproot?

A pure Python + Numpy implementation of ROOT I/O.

Why does it exist?

1. To extract columnar data (branches) from a ROOT file without invoking the event-handling infrastructure of the ROOT framework.
2. As a faster and fewer-dependencies alternative to root_numpy and root_pandas.
3. To express the semantics and conventions of the ROOT file format independently of ROOT, in lieu of a formal specification.

Why reimplement ROOT I/O?



It's more common to define a specification and implement many interpreters than not.

Most connect the format to different contexts, such as different languages.

(Take these numbers with a grain of salt: HDF5 is *criticized* because many of its bindings depend on only two C libraries!)

ROOT I/O implementations

ROOT	C++	ROOT itself	The ROOT Team
FreeHEP I/O → spark-root	Java/Scala	For Spark and other Big Data projects that run on Java	Started by Tony Johnson in 2001, updated by Viktor Khristenko
RIO → inlib/exlib	C++	Intended as an alternative, now embedded in GEANT-4	Guy Barrand
JsRoot	Javascript	For interacting with ROOT in web browsers or standalone	Bertrand Bellenot, Sergey Linev (in the ROOT Team)
go-hep/rootio	Go	HEP analysis ecosystem in Go	Sebastien Binet
uproot	Python	For quickly getting ROOT data into Numpy and Pandas for machine learning	Jim Pivarski (me)
alice-rs/root-io	Rust	ALICE ecosystem in Rust	Christian Bourjau

Why Python + Numpy?

- Physicists are already using Python for data analysis.
  - PyROOT has excellent coverage of the ROOT ecosystem, but calling individually wrapped C++ methods from Python is slow and the two languages have different (often conflicting) memory management.
  - Performance-oriented tools like root_numpy and root_pandas compile into a specific version of ROOT, which complicates upgrades. Also, asking for arrays through interfaces designed for event processing is a severe performance penalty.
- The scientific Python ecosystem, including much of machine learning, is designed around a fundamental abstraction called the Numpy array.
- Working with computer scientists is easier when you can say, “pip install uproot.”
- Implemented correctly, Python + Numpy doesn’t have to be slow.
  - Finding the columnar data in a ROOT file may be done in slow Python, as long as decompression and array manipulations are done by compiled code.

More context: BulkIO

Last summer, Brian Bockelman started a project called “BulkIO” to bypass the event processing framework in ROOT itself, providing direct access to branches as columns.

I added a BulkIO-to-Numpy interface in PyROOT in the same pull request [#943] and intend to maintain it, when it’s approved.

I wrote uproot while the pull request is in progress, based on the same BulkIO technique.

A tour of uproot

A tour of uproot

Install uproot and download a sample file.

```
$ pip install uproot --user
$ wget http://scikit-hep.org/uproot/examples/Zmumu.root
```

Start using it in Python.

```
>>> import uproot
>>> file = uproot.open("Zmumu.root")   # or root:// or http://
```

ROOT files, directories, and trees are like Python dicts with `keys()` and `values()`.

```
>>> file.keys()
['events;1']
>>> tree = file["events"]
>>> tree.keys()
['Type', 'Run', 'Event', 'E1', 'px1', 'py1', 'pz1', 'pt1', 'eta1',
 'phi1', 'Q1', 'E2', 'px2', 'py2', 'pz2', 'pt2', 'eta2', 'phi2',
 'Q2', 'M']
```

A tour of uproot

uproot's main purpose is to read branches from ROOT files as Numpy arrays.

```
>>> tree["px1"].array()
array([-41.195287,  35.118049,  35.118049, ...,  32.377491,
        32.377491,  32.485393])

>>> tree.arrays(["px1", "py1", "pz1"])
{'px1': array([-41.195287,  35.118049,  35.118049, ...,  32.377491,
               32.377491,  32.485393]),
 'py1': array([ 17.43324 , -16.570362, -16.570362, ...,   1.199405,
                1.199405,   1.20135 ]),
 'pz1': array([-68.964961, -48.775246, -48.775246, ..., -74.532430,
              -74.532430, -74.808372])}
```

A tour of uproot

Iteration lets us fetch data in batches— large enough to be efficient in Python but small enough to fit in memory.

```
>>> for arrays in tree.iterate(entrysteps=10000):
...     do_something(arrays)    # all arrays in chunks of 10k events

>>> for arrays in uproot.iterate("/path/to/files*.root", "Events"):
...     do_something(arrays)    # this is like a TChain
```

The array-fetching methods share most parameters: you can specify a subset of branches in `iterate` just as you would in `arrays`.

If unspecified, `entrysteps` defaults to the ROOT file's cluster size, reading whole baskets at a time.

A tour of uproot

One of these array-fetching methods fills a Pandas DataFrame.

```
>>> tree.pandas.df(["pt*", "eta*", "phi*"])
          eta1      eta2      phi1      phi2     pt1     pt2
0    -1.217690 -1.051390  2.741260 -0.440873  44.7322  38.8311
1    -1.051390 -1.217690 -0.440873  2.741260  38.8311  44.7322
2    -1.051390 -1.217690 -0.440873  2.741260  38.8311  44.7322
3    -1.051390 -1.217690 -0.440873  2.741260  38.8311  44.7322
...        ...       ...       ...       ...      ...      ...
2300 -1.482700 -1.570440 -2.775240  0.037027  72.8781  32.3997
2301 -1.570440 -1.482700  0.037027 -2.775240  32.3997  72.8781
2302 -1.570440 -1.482700  0.037027 -2.775240  32.3997  72.8781
2303 -1.570440 -1.482700  0.037027 -2.775240  32.3997  72.8781

[2304 rows x 6 columns]
```

Features like this are easy now that the core ROOT-reading functionality is in place.

A tour of uproot

uproot follows Pythonic customs: high-level yet explicit. For example, asking for the same array twice reads it from the file twice.

```
>>> arrays = tree.arrays()                     # reads all data
>>> arrays = tree.arrays()                     # reads all data again
```

That is, unless you give it a cache (anything that acts like a dict).

```
>>> cache = {}
>>> arrays = tree.arrays(cache=cache)          # reads all data
>>> arrays = tree.arrays(cache=cache)          # gets it from the cache
>>> len(cache)
20

>>> limitedcache = uproot.cache.MemoryCache(5*1024)     # limit to 5 kB
>>> arrays = tree.arrays(cache=limitedcache)
>>> len(limitedcache)
18
```

A tour of uproot

Same for concurrency: it's single-threaded unless you give it an executor (anything with a `map` method returning a non-blocking generator of results).

```
>>> from concurrent.futures import ThreadPoolExecutor

>>> executor = ThreadPoolExecutor(8)           # 8 threads
>>> arrays = tree.arrays(executor=executor)    # read, decompress all
                                               # baskets in parallel
```

There's also a non-blocking form that returns a function. Processing happens in the background until you call this function, which waits for and returns the array data.

```
>>> wait = tree.arrays(executor=executor, blocking=False)
>>> wait
<function wait at 0x7a1d744515f0>
>>> wait()                                     # now get the arrays
...
```

A tour of uproot

A lot of ROOT trees are not flat tables, but contain arbitrary-length lists of particle attributes. Numpy only deals with n-dimensional arrays, so we need a new container.

```
>>> tree = uproot.open("http://scikit-hep.org/uproot/"
...                          "examples/HZZ.root")["events"]
>>> ja = tree.array("Jet_E")
>>> ja
jaggedarray([[],
             [44.137363],
             [],
             ...,
             [55.95058],
             [229.57799  33.92035],
             []])
>>> ja[0]
array([], dtype=float32)
>>> ja[1]
array([44.137363], dtype=float32)
```

A tour of uproot

This is much faster than root_numpy's behavior (arrays as objects in an object array) because uproot only loads the information needed to identify subarrays (contiguous in memory) without constructing them all (randomly in memory).

```
>>> ja.content
array([44.13, 230.34, 101.35 ... 55.95, 229.57, 33.92], dtype=float32)
>>> ja.offsets
array([0, 0, 1 ... 2771, 2773, 2773])
```

There are also string types, vector of strings, vector of vector of numbers, etc. Eventually, uproot will be able to interpret any data type. To see if your branch's type is currently supported, call `tree.show()` and check the third column for `None`.

```
>>> tree.show()
NJet              (no streamer)     asdtype('>i4')
Jet_E             (no streamer)     asjagged(asdtype('>f4'))
Jet_ID            (no streamer)     asjagged(asdtype('>bool'))
...
```

A tour of uproot

uproot uses a ROOT file's streamer info to know how to deserialize classes, staying abreast of changes. Thus, any type of object may be extracted from a ROOT file.

```
$ wget "https://github.com/HEPData/hepdata-submission/blob/master/examples/submission/"\
"TestHEPSubmission/root_file.root?raw=true" -O hepdata.root

>>> file = uproot.open("hepdata.root")
>>> dict(file.classes())
{'hpx;1': <class uproot.rootio.TH1F>, 'hpxpy;1': <class uproot.rootio.TH2F>,
 'hprof;1': <class uproot.rootio.TProfile>, 'ntuple;1': <class uproot.rootio.TNtuple>}
>>> histogram = file["hpx"]
>>> histogram.fTitle
'This is the px distribution'
>>> histogram.fFunctions[0]
<TPaveStats 'stats' at 0x71cb4586df90>

>>> histogram.fXaxis.__dict__          # all the fields are there, without interpretation
{'fTitleFont': 42, 'fLabelColor': 1, 'fNdivisions': 510, 'fXmin': -4.0,
 'fTimeDisplay': False, 'classversion': 1, 'fLabelFont': 42, 'fNbins': 100,
 'fLabels': None, 'fXbins': [], 'fTitleColor': 1, 'fLabelOffset': 0.004999999888241291,
 'fName': 'xaxis', 'fLast': 0, 'fAxisColor': 1, 'fLabelSize': 0.03500000014901161,
 'fTitleOffset': 1.0, 'fTitle': '', 'fFirst': 0, 'fXmax': 4.0,
 'fTickLength': 0.029999999329447746, 'fTimeFormat': '', 'fBits2': 0,
 'fTitleSize': 0.03500000014901161}
```

A tour of uproot

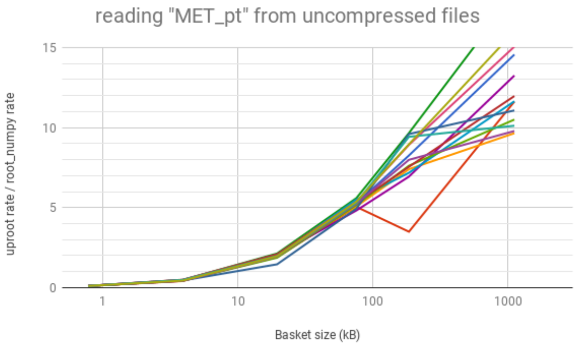
All that remains is to give these objects Pythonic interpretations.

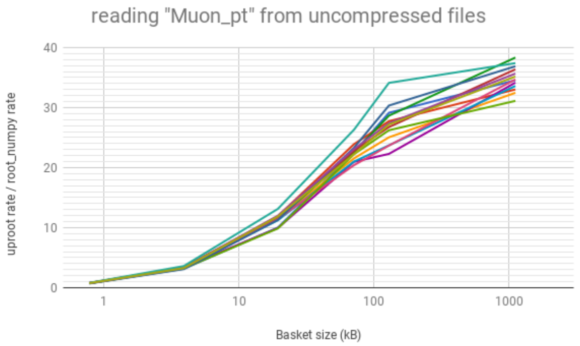
```
>>> histogram.bokeh.plot()              # basic implementation, somewhat clunky
>>> histogram.holoviews.plot()          # only works in a Jupyter notebook

>>> histogram.fit(lambda x, a, b, c: a*exp(-(x - b)**2 / c))     # dreaming...

>>> histogram.show()                    # surprisingly useful
                  0                                                       2410.8
                  +------------------------------------------------------------+
[-inf, -3)   0    |                                                            |
[-3, -2.4)   68   |**                                                          |
[-2.4, -1.8) 285  |*******                                                     |
[-1.8, -1.2) 755  |*******************                                         |
[-1.2, -0.6) 1580 |***************************************                     |
[-0.6, 0)    2296 |*********************************************************   |
[0, 0.6)     2286 |*********************************************************   |
[0.6, 1.2)   1570 |***************************************                     |
[1.2, 1.8)   795  |********************                                        |
[1.8, 2.4)   289  |*******                                                     |
[2.4, 3)     76   |**                                                          |
[3, inf]     0    |                                                            |
                  +------------------------------------------------------------+
```

Performance

Performance: comparison to root_numpy (uncompressed)

In terms of functionality, uproot is most similar to root_numpy. Depending on basket size and whether the array is flat (fixed width per event) or jagged (variable width), uproot can be as much as 40 times faster.

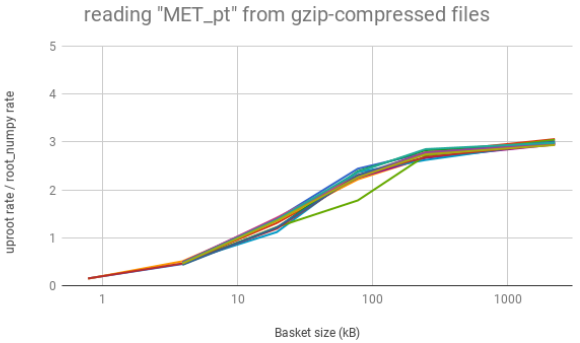


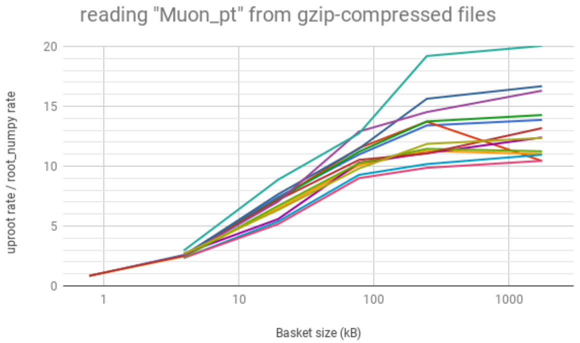


The 13 lines are different physics distributions from CMS NanoAOD (mileage varies). Speedup increases with basket size because more work is done in Numpy, not Python.

Performance: comparison to root_numpy (gzip-compressed)

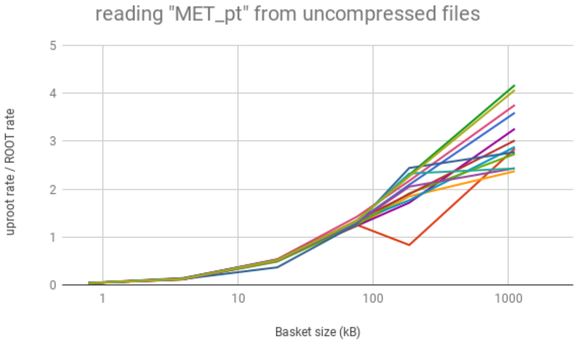
The distinction drops to a factor of 3 when flat data are compressed, but still there's a factor of 10 for jagged data because of the way root_numpy handles this type.

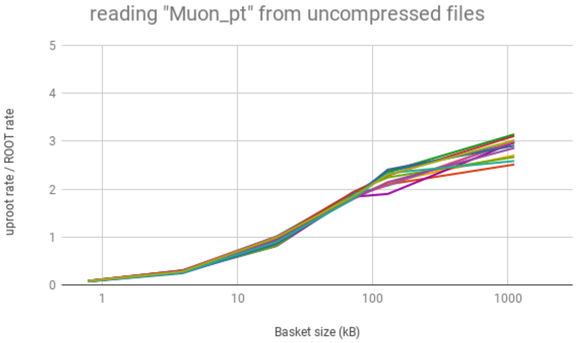




Performance: comparison to ROOT (uncompressed)

For sufficiently large baskets, uproot even compares favorably to ROOT because uproot bypasses the event processing framework.

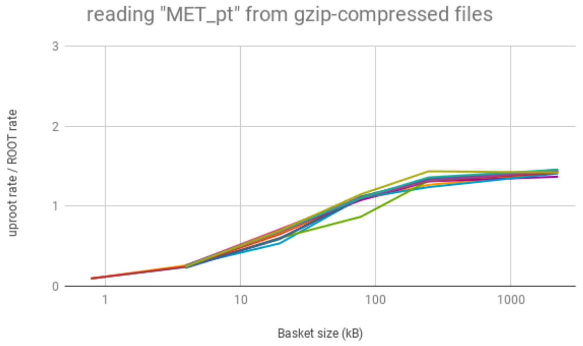


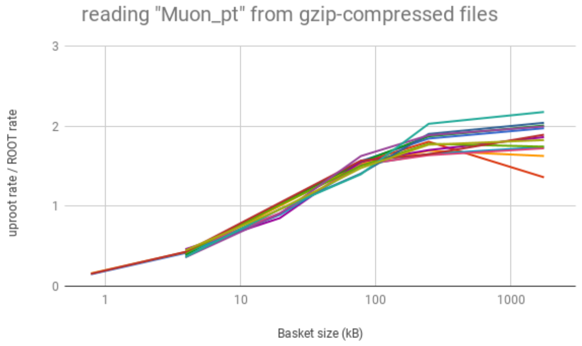


When the BulkIO feature is added to ROOT, it will be about 30 times faster than this baseline, so ROOT `TBranch::GetEntry` $\ll$ uproot $\ll$ ROOT BulkIO.

Performance: comparison to ROOT (gzip-compressed)

As you add compression, the distinction washes out because both processes spend more of their time in the same decompression algorithm.





For read performance, no compression $\sim$ lz4 $\ll$ gzip $\ll$ lzma.

If you are preparing data for uproot

. . . make large baskets:

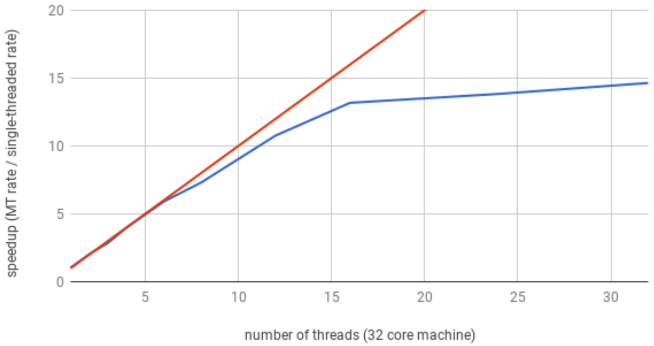
```
// 100 kB to 1 MB baskets
tree->Branch("branch", &data, "branch/F", 1024*1024);

// negative flush size is size of all branches in a cluster
tree->SetAutoFlush(-1024*1024 * numbranches);
```

. . . and use LZ4 compression:

```
file->SetCompressionAlgorithm(ROOT::kLZ4);    // ROOT::kLZ4 is 4
file->SetCompressionLevel(3);
```

Last performance plot: multithreaded scaling

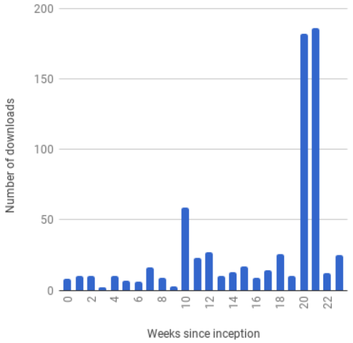
Although Python locks all threads at each step in its interpreter (the “GIL”), compiled code escapes this limitation and can scale on multicore machines.

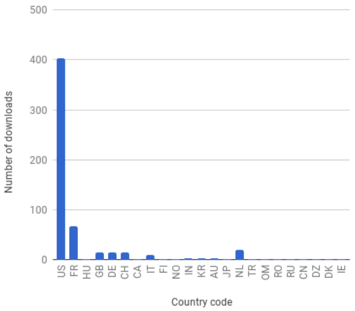


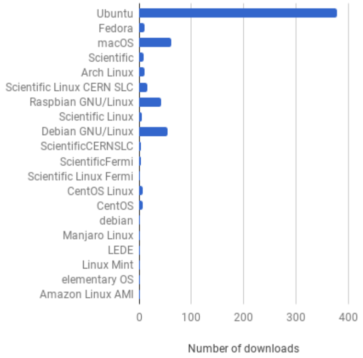
In the above, we iterated through a thousand lzma-compressed branches (lzma requires the most CPU time to decompress), distributed as described on page 13.

Activity, status, and future

Download statistics (via “pip install”)







Number of unique country code/OS distribution/version combinations (“users”): 127.

Client in NL requests every version; others request only the latest.

Popular with Linux laptops (Ubuntu); I want to know who’s using it on a Raspberry Pi!

GitHub issues not posted by me: 33; stars: 72; watchers: 10.

Status and future

- uproot has reached a stable plateau, I'm mostly responding to bugs and feature requests now— no deep overhauls.
- This summer, Pratyush Das (an undergrad) will be adding write support for file output.
- uproot will always be an I/O-only library. However, it may accrue features that *connect* it to other packages, such as

  ```
  tree.pandas.df()
  ```

  and

  ```
  histogram.bokeh.plot()
  ```

  which can eventually make it part of a Pythonic analysis environment.

Status and future

Yesterday, Chris Burr implemented a pyparsing-based translator from ROOT's TFormula language to numexpr. This would make it possible for uproot to reproduce all functionality currently found in root_numpy.

Is there interest in

```
import uproot.root_numpy as root_numpy
```

as a drop-in replacement?

Last slide: thanks for listening!

```
pip install uproot --user
```

https://github.com/scikit-hep/uproot

http://uproot.readthedocs.io

https://groups.google.com/forum/#!forum/uproot-users/join

BACKUP

Code used for performance studies

For all (prewarmed cache eliminates dependence on disk, which was SSD anyway):

```
$ vmtouch -t filename
```

For uproot:

```
import uproot
tree = uproot.open(filename)[treename]

startTime = time.time()
tree.array(branchname)
return time.time() - startTime
```

For root_numpy:

```
import root_numpy
file = ROOT.TFile(filename)
tree = file.Get(treename)

startTime = time.time()
root_numpy.tree2array(tree, [branchname])
return time.time() - startTime
```

Code used for performance studies

For C++ ROOT:

```
TFile *file = new TFile(filename);
TTree *tree;
file->GetObject(treename, tree);

float MET_pt;
float Muon_pt[8];

TBranch *branch;
if (branchname == "Muon_pt") {
  branch = treein->GetBranch("Muon_pt");
}
else if (branchname == "MET_pt") {
  branch = treein->GetBranch("MET_pt");
}

branch->SetAddress(&MET_pt);
branch->SetAddress(&Muon_pt);
```

```
struct timeval startTime, endTime;

Long64_t nEvents = tree->GetEntries();
Long64_t iEvent;
gettimeofday(&startTime, 0);
for (iEvent = 0; iEvent < nEvents; ++iEvent) {
    // not tree->GetEntry(iEvent) because
    // that would touch all branches!
    branch->GetEntry(iEvent);
}
gettimeofday(&endTime, 0);

double microsecs =
   1000000*(endTime.tv_sec - startTime.tv_sec)
         + (endTime.tv_usec - startTime.tv_usec);

return microsecs / 1000000.0;
```