PYTHON IN HIGH-ENERGY PHYSICS

Hans Dembinski, MPIK Heidelberg

21 Mar 2019

ABOUT ME

- Cosmic ray/HEP physicist now in LHCb
- Trying to solve the Muon Puzzle in air showers
- Active in the Boost C++ and Scikit-HEP Python communities
- My OSS projects
  - Boost::Histogram
  - pyhepmc
  - iminuit (maintainer)

TAKE-HOME MESSAGE

- HEP software is still dominantly C++ (ROOT)…
  - … but half the analyses in LHCb already in Python (survey 2018)
  - Next major release ROOT 7 will resolve fundamental design issues
- OSS initiatives in Python and C++ offer alternatives to ROOT
  - Scikit-HEP Project: uproot, iminuit, …
  - Boost::Histogram with Python frontend
- Bright future for Python in HEP
  - Python can easily bind to C++ libraries with **pybind11**
  - Python itself can be made fast with **Numba**
  - Growth of Python ecosphere outperforms growth of C++ ecosphere

HIGH-ENERGY PHYSICS

Big Data: billions of events, Petabytes of data

- Need fast code to execute on computing clusters
- Hierarchical data structures: Trees (event variables, track variables)

Computing uses consumer hardware (no Crays)

- Run same code on laptop and cluster (almost)

Physicists traditionally prefer to use one language for everything

- Past: libraries and analysis code written in C++ (Fortran before)
- Current: write libraries in C++ and analysis code in C++ or Python
- Trend: more Python, less C++

ROOT FRAMEWORK

- Latest release 6.16/00
- Large meta-library
  - IO, data structures, histograms, fitting, graphics, databases, OS interaction, …
- High-level statistics tools
  - RooFit, RooStats, TMVA

WHAT ROOT DOES WELL

ROOT IO: `TFile` & `TTree` have no equal

- Portable binary hierarchical data format
- Transparent compression
- Allows partial reads & partial recovery from failed writes
- Fast interactive data exploration with `TTree::Draw`

Cling: ROOT's C++ runtime interpreter

- Fully standard compliant (based on LLVM)
- Run C++ code like a script or compile for fast execution
- Replaced CINT from ROOT 5

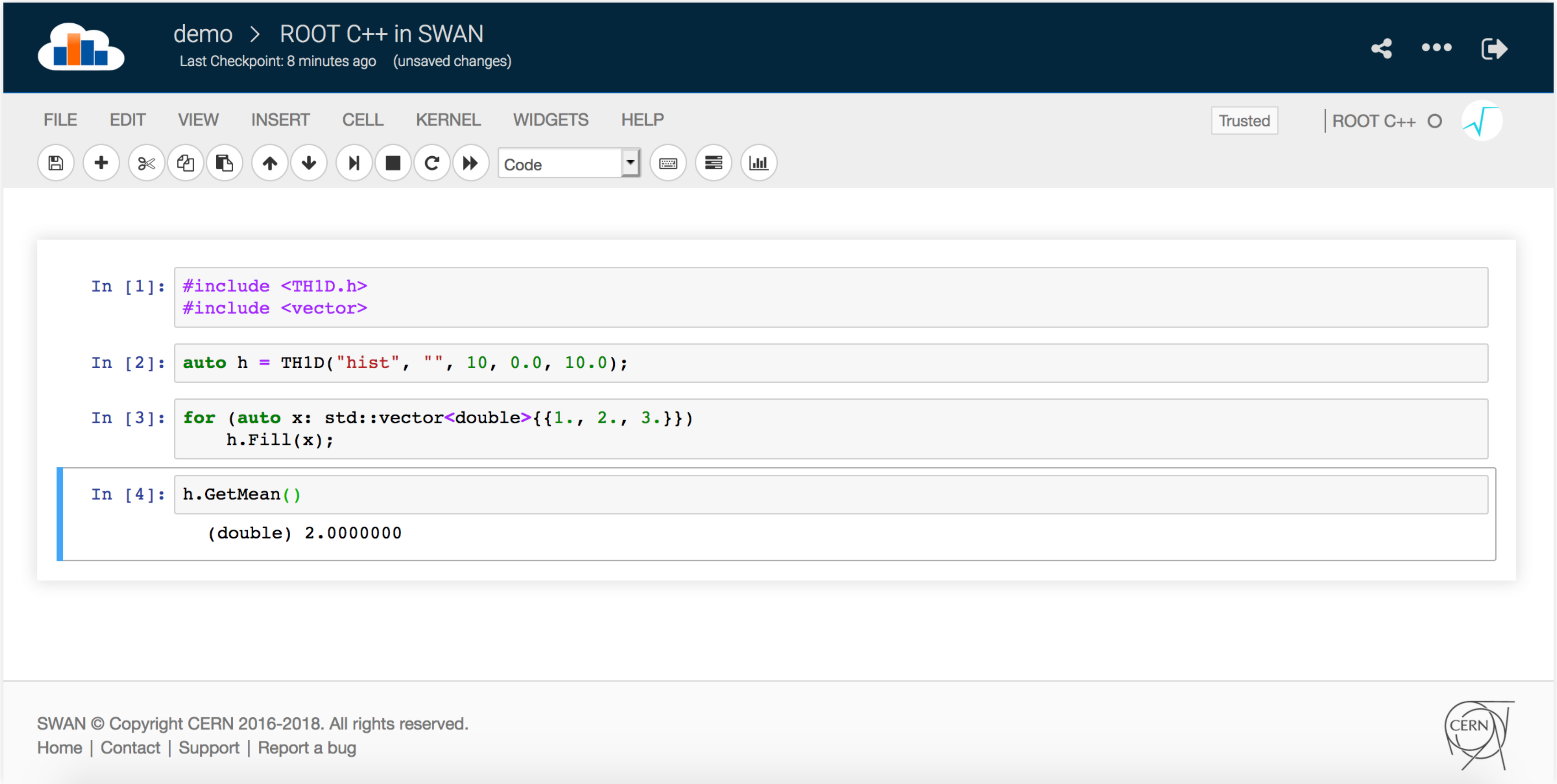
PyROOT: Auto-generated Python bindings

- Wraps arbitrary C++ code to Python without extra effort **(when it works)**

Backward compatibility

ROOTBOOKS IN SWAN

Jupyter on top of CERNBox with Python and ROOT C++ kernels



WHAT ROOT DOES NOT SO WELL

Brittle automatic memory management

- No. 1 user complaint, see my LHCb talk at ROOT Users' Workshop, slide 11

ROOT tried to replace ~~the C++ standard~~ any library

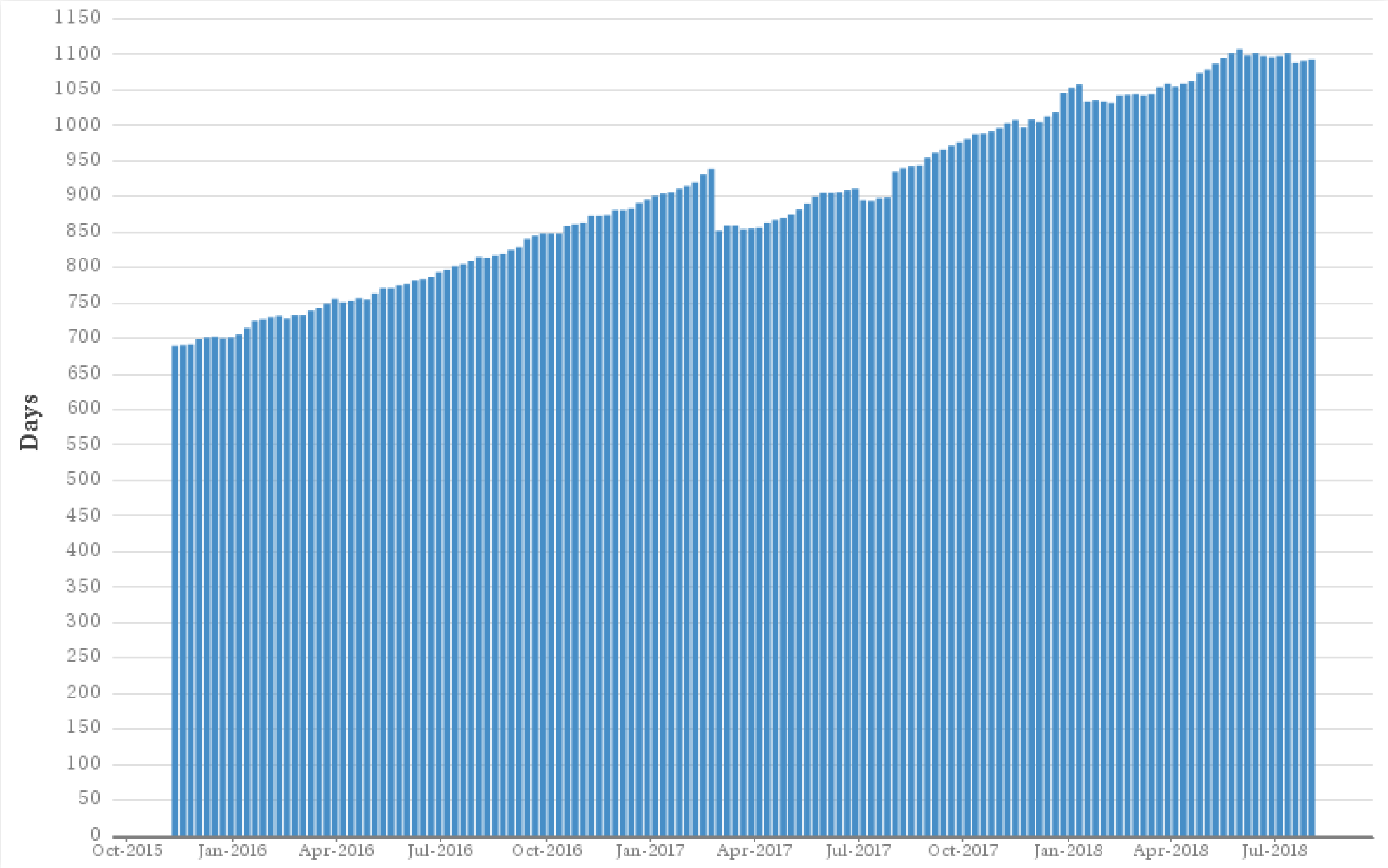
- Not-invented here syndrome and vendor lock-in
- Standard interfaces duplicated in ROOT with added maintenance burden
- Users forced to learn ROOT style instead of idiomatic C++

Maintenace nightmare

- Bugs bugs bugs, and many of them open for years
- Too small developer team for too large code base
- Little support from industry and OSS community

Design issues: leaking abstractions, lack of RAII, inconsistencies

AVERAGE BUG LIFETIME IN ROOT



DESIGN ISSUES

Actual ROOT code

```
TFile* outfile = new TFile(...); // stack allocation usually does not work
TH1D* histogram = new TH1D(...); // ROOT wants everything on the heap
// ...fill histogram...
histogram->Write(); // how does histogram know where to write to?
outfile->Close(); // histogram also silently deleted here?
delete outfile; // histogram also silently deleted here?
```

Desired ROOT code

```
TFile outfile("output.root", "recreate"); // stack allocation works
TH1D histogram(...);
// ...fill histogram...
outfile << histogram; // ostreaming, just like in std iostreams
outfile.close(); // no coupling of life-time of TFile and TH1D
```

THE FUTURE: ROOT 7

First release in 20 years to break backward-compatibility

- Required to fix historic mistakes in interfaces and memory management
- “We will use standard C++ types, standard interface behavior”

Nice new things

- RHist replaces previous histograms
- RDataFrame replaces TTree
- Better (automatic) parallelization
- Better graphics

Many talks about ROOT 7 at ROOT Users’ Workshop 2018

WHY ROOT 7 WILL NOT WIN THE DAY

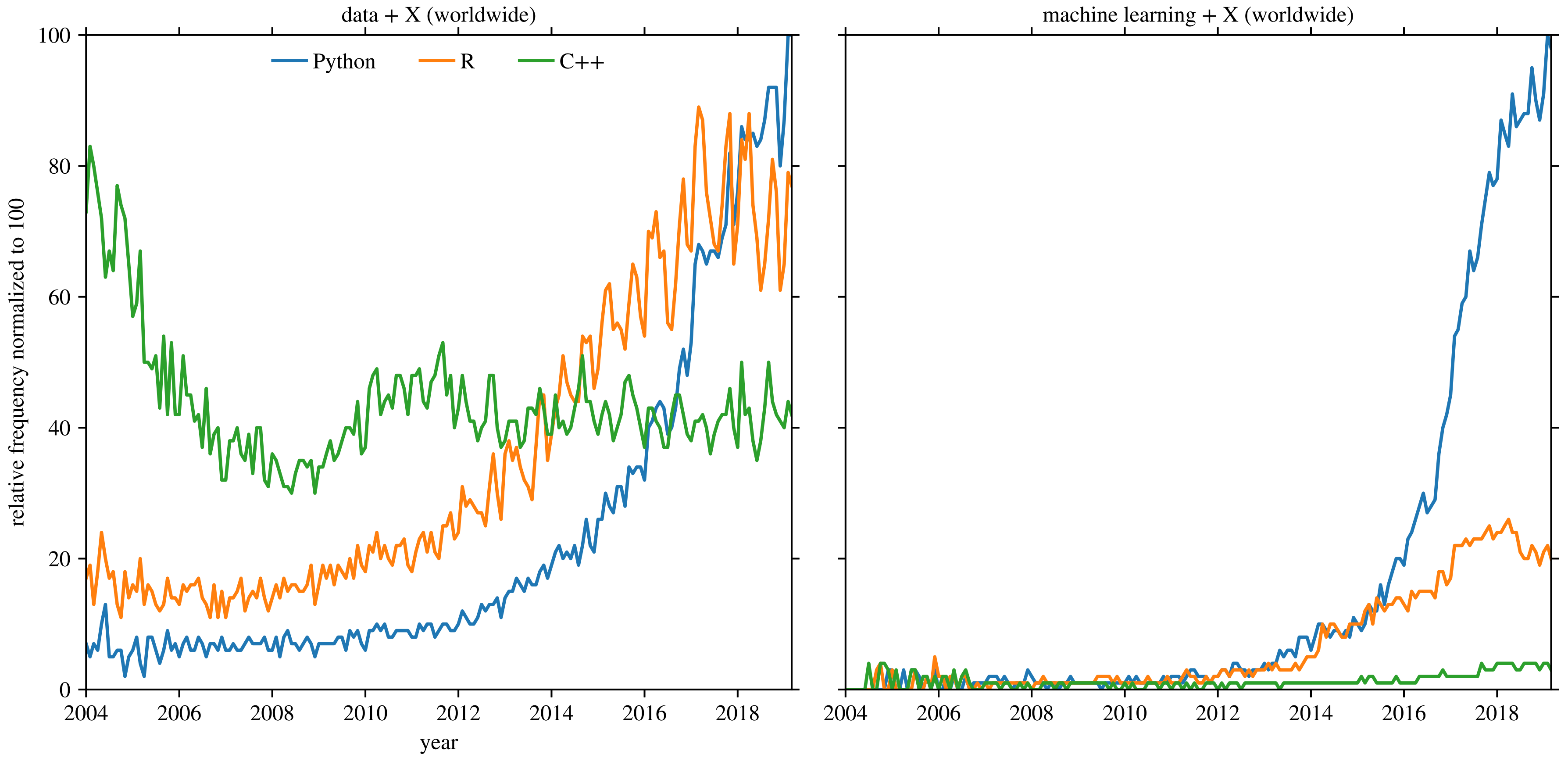
- ROOT 7 is a big improvement, but…
- Big Data community is moving away from C++ towards Python
  - Industry-powered machine learning tools are in Python
  - ML tools draw people to Python ecosphere
  - Python gives you access to better and faster evolving libraries
  - Why would you ever go back?
- Manpower problem remains
  - Still large amounts of *tech debt* which binds manpower
    - Can either fix bugs or develop new features
  - Loosing race againsts other libraries which attract more manpower
    - ROOT core team are good people, but cannot compete with OSS community
    - Support unlikely to come from OSS community/industry

PYTHON

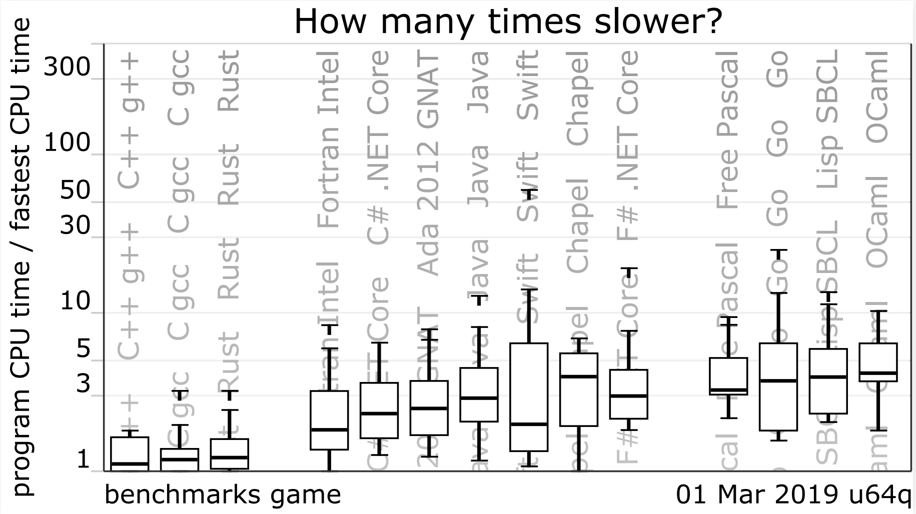
- Now the dominant language in scientific computing
  - Comfortable syntax for analysis scripts
  - Easy to learn *and* master
  - Rich and vibrant ecosphere
    - NumPy, matplotlib, scipy, scikit-learn, pandas, Jupyter
    - Anaconda, PyTorch, TensorFlow, Keras, …
  - Easy to write and distribute new libraries
- Adopted by industry leaders: Google, Instragram, Facebook, …
- Adopted by leading (astro)particle physics experiments
  - IceCube Neutrino Observatory, CTA, CERN, …

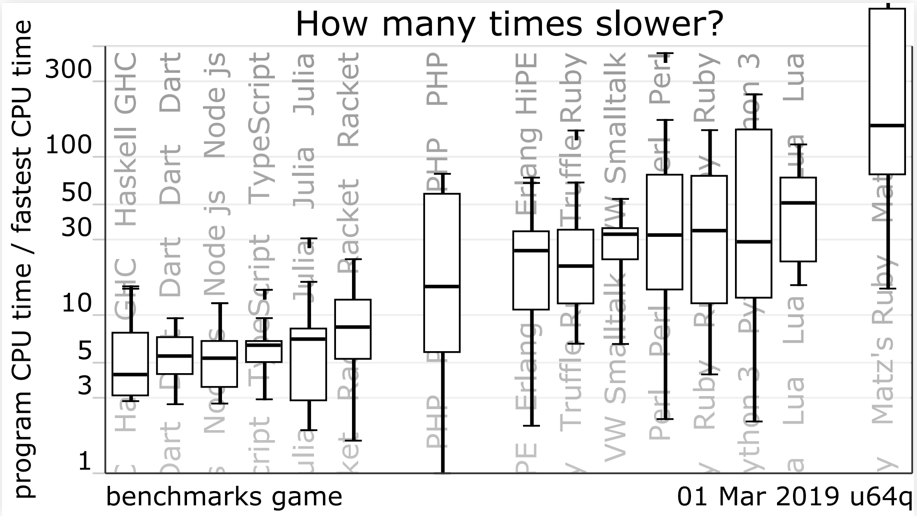
Really, *everything*. Even CMake or pybind11.

GOOGLE TRENDS



BUT PYTHON IS SLOW...!





Source: The Benchmark Game

… OR IS IT?

- Use a fast Python library (written in C/C++, Fortran, …)
  - NumPy, CuPy, SciPy, …
- Use a JIT in your Python session: **Numba**
- Use a faster Python interpreter: **PyPy**
- Use Python as a glue language
  - Python configures and steers fast C/C++/Fortran code
  - Passes memory buffers from one library to the next
  - Examples: ROOT, LHCb Core Software, IceCube Framework…
  - Generate bindings with …
    - **pybind11**, cffi, f2py, ctypes, Cython, Boost.Python, SWIG, PyROOT, …

NUMPY

- SIMD programming: **S**ingle **I**nstruction on **M**ultiple **D**ata
- Compute one array at a time instead of one value at the time
- Python loops and functions are slow, NumPy calls them in C

Pro	Contra
Easy to use	Creates temporary arrays which could be avoided
Quite fast	Not so readable/fast when instruction has branches
Often compact readable code	Learning-curve: Thinking in arrays, NumPy API

```python
import numpy as np
x = np.random.rand(1000)

# good
a = 2 * x + 1
b = np.log(x ** 4)
c = x > 0.5  # creates a boolean array, can be used to filter x

# not so good: compute 2 x if x < 2 and else x + 3
d = np.where(x < 2, 2 * x, x + 3)
```

- Doesn’t work when instructions differ for each element
  - MC simulation of multiple particle trajectories
  - Mandelbrot fractal (no. of iterations vary in each pixel)

NUMBA: JIT COMPILER FOR PYTHON

1. Translates Python code into AST (types are inferred)
2. Applies optimizations (vectorization, parallelization)
3. Compiles AST with LLVM into machine code

Pro	Contra
Easy to use	Not all Python types supported
Really fast pythonic code	Only works on functions and methods (not classes)
Supports auto-parallelization	Learning-curve: understanding Numba errors
Supports GPU computation	
Use NumPy as input and output	

Numba is pretty smart: inlines nested JITed functions, …

Just import `njit` and decorate your function

```
from numba import njit
import numpy as np
x = np.random.rand(1000)

def func_with_branch_numpy(x):  # 11 µs
    return np.where(x < 0.5, 2 * x, x + 3)

@njit
def func_with_branch_numba(x):  # 0.9 µs
    result = np.empty_like(x)
    for i, xi in enumerate(x):
        if xi < 0.5:
            result[i] = 2 * xi
        else:
            result[i] = xi + 3
    return result
```

Numba is **12x** faster than NumPy on my laptop

PYPY: JIT-ENABLED INTERPRETER

Alternative JIT-enabled Python interpreter written in RPython

Pro	Contra
Ideally: Use PyPy and code gets fast	**Not all Python libraries work: e.g. SciPy**
Expressions are JIT-compiled as needed	A bit cumbersome to install
Can optimize classes	Lagging behind CPython syntax (stable: 3.5)
Can do global code optimizations	NumPy code may run slower
Numpy, matplotlib work	NumPyPy incomplete

Official Download and Install Page

Portable binaries for Linux

```
mkdir -p $HOME/pypy
URL = https://bitbucket.org/squeaky/portable-pypy/downloads/pypy3.5-7.0.0-
  linux_x86_64-portable.tar.bz2
wget -O - $URL | tar xjf - --strip-components=1 -C $HOME/pypy
$HOME/pypy/bin/virtualenv-pypy $HOME/pypy/venv
source $HOME/pypy/venv/bin/activate
```

Mac OS X binary

```
mkdir -p $HOME/pypy
URL = https://bitbucket.org/pypy/pypy/downloads/pypy3.5-v7.0.0-osx64.tar.bz2
wget -O - $URL | tar xjf - --strip-components=1 -C $HOME/pypy
pip install --user virtualenv
virtualenv $HOME/pypy/venv -p $HOME/pypy/bin/pypy3
source $HOME/pypy/venv/bin/activate
```

- PyPy3.5-7.0: **1.7x** faster than NumPy in CPython
  - Numba in CPython **7x** faster than PyPy3.5-7.0
- Could not compile NumPy on OSX (works on Linux)
  - setuptools doesn't add `-stdlib=libc++` on Darwin platform 😤

```
import random
x = [random.uniform(0, 1) for i in range(1000)]

def func_with_branch(x):  # 6.3 µs
    result = [0.0] * 1000  # using [0] * 1000 here gives a slowdown of 2!
    for i, xi in enumerate(x):
        if xi < 0.5:
            result[i] = 2 * xi
        else:
            result[i] = xi + 3
    return result
```

… but you can write plain pythonic code and it is fast

SCIKIT-HEP PROJECT

Online community which develops Python stack for HEP

- Supported by IRIS-HEP, NSF funded software institute
- Leading members from Princeton, Cincinnati U, Washington U…

Join us on Gitter: https://gitter.im/HSF/PyHEP

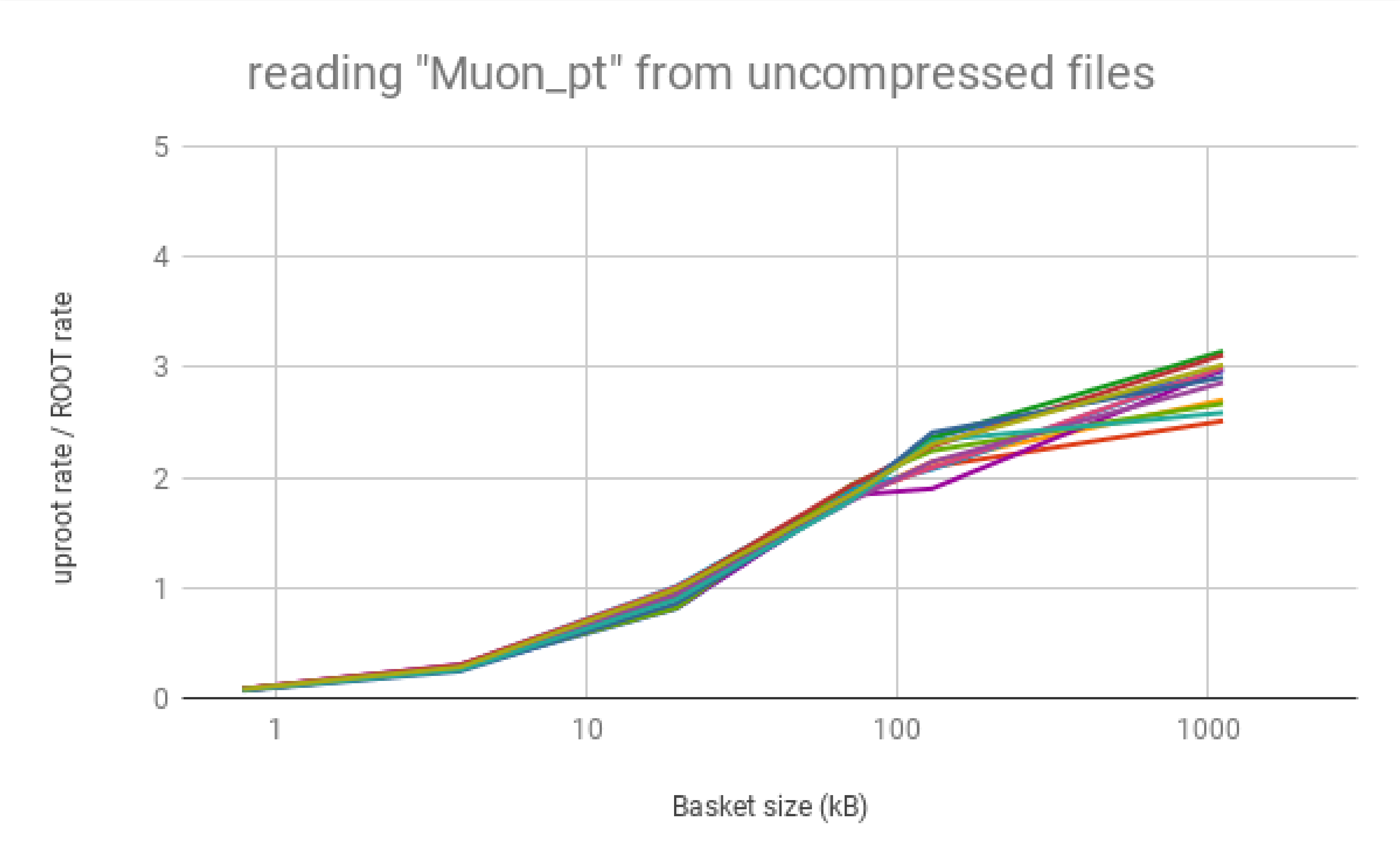
Scikit-HEP forum: scikit-hep-forum@googlegroups.com

On Github: https://github.com/scikit-hep

Home of uproot, iminuit, boost-histogram, particle, pyhepmc, …

UPROOT

- Implementation ROOT I/O in **pure Python and Numpy**
- Read/write ROOT trees, histograms, TGraphs, T(Lorentz)Vectors
- Can read data fields of any other ROOT type
- Up to **3x faster** than C++ ROOT
- Does not depend on C++ ROOT (just one `pip install` away)
- Extensible, see uproot-methods repository
- Powered by awkward-array
  - Hierarchical array implemented on top of standard Numpy arrays
  - See Jim Pivarski’s talk for interesting details


reading "Muon_pt" from uncompressed files
uproot rate / ROOT rate
5
4
3
2
1
0
1
10
100
1000
Basket size (kB)

```python
import numpy as np
import uproot

f = uproot.open("~/Data/sct/mc/00058786_00000001_5.sct.root")
print(f.keys())
# [b'sct;6', b'sct;5']

f['sct'].show()
# evt_run                  (no streamer)            asdtype('>i4')
# ...
# vtx_x                    (no streamer)            asjagged(asdtype('>f4'))

f['sct/evt_evnum'].array()
# array([5881230, 5881230, ..., 5878628, 5878628], dtype=int32)

pz = f['sct/trk_pz'].array()
# <JaggedArray [[4186.4 5212.5 3073.3] [] [6479.1 3533.5] ...]>

from matplotlib import pyplot as plt
plt.hist(np.log10(pz.flatten()))  # plot log10(pz) distribution

for pxi in f['sct/trk_px'].array(): print(np.mean(pxi))
# 150.75218 nan -79.71784 -120.3935 nan -146.99773 12.007137 ...
```

IMINUIT

The Python wrapper of C++ MINUIT2 library

- Other wrappers (pyminuit, pyminuit2) discontinued
- Bindings generated with Cython (will switch to pybind11)
- Python 2.7 to 3.7 on Linux, Mac, Windows
- New: PyPy support (PyPy3.5-7.0)

Does not depend on C++ ROOT

- Simply install with `pip` or `conda`

Many good OSS minimizers: scipy, libnlopt, …

MINUIT's unique feature is error computation with Hesse & MINOS

```python
from iminuit import Minuit

def f(x, y, z):
    return (x - 2) ** 2 + (y - 3) ** 2 + (z - 4) ** 2

m = Minuit(f)     # Minuit automagically detects parameter names!

m.migrad()        # run optimiser
print(m.values)   # {'x': 2,'y': 3,'z': 4}

m.hesse()         # run Hesse error estimator
print(m.errors)   # {'x': 1,'y': 1,'z': 1}
```

- Minuit can do much more
  - Parameters with limits
  - Fixed parameters
  - Pretty Jupyter output
  - Builtin plotting of error contours and function minimum

BOOST-HISTOGRAM

Python wrapper (alpha stage) for Boost::Histogram in C++

Boost::Histogram will be first released with Boost-1.70 in April

- Generalized multi-dimensional histograms and profiles in idiomatic C++14
- Use buitin axis types or add your own
  - regular, variable, circular, category; all growing or non-growing
  - Support for complex binning schemes, like hexagonal binning
- Easy and safe to use in default configuration
- Very customizable for power users
  - Get the highest speed for given task
  - Write new specialized axis and storage types that we didn't think of
- TMP under the hood makes execution fast and interface easy to use

```python
from boost.histogram import histogram
from boost.histogram.axis import regular, category

hist = histogram(category(("red", "blue")),
                 regular(4, 0.0, 1.0))

# input doesn't have to be numerical
hist(["red", "red", "blue"],
     [0.1  , 0.4  , 0.9   ])

counts = hist.view

# returns numpy array view into histogram counts:
# [[1, 1, 0, 0],
#  [0, 0, 0, 1]]
```

SUMMARY AND OUTLOOK

HEP software is still dominantly C++, but bright future for Python

- Python can be very fast with Numba
- Python can integrate with C/C++ libraries using pybind11
- If you can write fast code in Python, why would you use C++?

OSS initiatives in Python and C++ offer alternatives to ROOT

- Scikit-HEP Project: uproot, iminuit, …
- Boost::Histogram with Python frontend
- Specialized HEP-style plots in development, to be included in matplotlib

BACKUP: PYBIND11 VS. CYTHON

- Cython: transpiler for custom Python/C mixed dialect
  - Learning curve: need to learn this dialect
  - Designed for C; C++ only partially supported
  - Clumsy syntax, workarounds needed for missing features and bugs
  - Cython adds problems instead of solving them
- pybind11
  - Based on the brilliant Boost::Python library
  - No transpiler, just a header-only C++11 library
  - Uses TMP to automate boilerplate code
  - Automated handling of refcounts
  - Full power of C++, no workarounds, explicit ownership of memory
  - Excellent docs

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
namespace py = pybind11;

py::array_t<double> func_with_branch(py::array_t<double> x) {
  auto result = py::array_t<double>(x.shape(0));
  auto rd = result.mutable_data();
  auto xd = x.data();
  for (ssize_t i = 0, n = x.shape(0); i < n; ++i) {
    if (xd[i] < 0.5) {
      rd[i] = 2 * xd[i];
    } else {
      rd[i] = xd[i] + 3;
    }
  }
  return result;
}

PYBIND11_MODULE(example, m) {
  m.def("func_with_branch", &func_with_branch); // 1.7 µs (compiled with -O3)
}
```

6.5x faster than NumPy version, but **1.9x slower** than Numba