MAX-PLANCK-INSTITUT FÜR KERNPHYSIK

Boost.Histogram

Hans Dembinski[1]
[1]MPIK Heidelberg, Germany

DIANA-HEP meeting, 1 Oct 2018

boost.histogram in a nutshell

- Multi-dimensional histogram in C++, current release v3.2
  - Header-only
  - Feature set based on ROOT histograms, GSL histograms, scikit-hep/histbook
- Source: https://github.com/hdembinski/histogram
- Docs: http://hdembinski.github.io/histogram/doc/html
- Thoroughly unit-tested: line coverage **99.94 %**
- Selected features
  - Automatic memory efficient handling of bin counters
  - No-overflow guarantee
  - Supports many axis types, e.g. circular axis and category axis
  - Supports weighted increments
  - **Faster** than other libs in benchmarks (but see details)
- Planned features
  - Support of profiles
  - Support library with histogram transformations
- Accepted for inclusion in Boost
- Python bindings under development (used to be included in earlier releases) https://github.com/hdembinski/histogram-python

Why Boost C++ libraries?

- Boost
  - Free peer-reviewed portable C++ source libraries
  - Greatly extends functionality of C++ stdlib
  - Popular in science and industry
  - Often first step towards C++ standardization

- Why adding a histogram library to Boost?
  - Standard-alone basic component in statistics software
    - Compact self-contained library is possible
  - Stop reinvention of the wheel
    - Solution must be useful for everyone
    - Solution must be customizable and fast (policy-based design)
  - Complement existing sub-libraries with statistics tools
    - Accumulators
    - Math
    - Random (superset of std::random)

C++ example

```
#include <boost/histogram.hpp> // all-in-one header

int main() {
  namespace bh = boost::histogram;
  using namespace bh::literals; // enables _c suffix

  auto h = bh::make_static_histogram( bh::axis::regular<>(6, -1.0, 2.0, "x") );

  auto data = { -0.4, 1.1, 0.3, 1.7 };
  auto h = std::for_each(data.begin(), data.end(), h);

  for (auto it = h.begin(); it != h.end(); ++it) {
    const auto bin = it.bin(0_c);
    std::cout << "bin " << it.idx(0) << " x in [" << bin.lower() << ", " << bin.upper() << "): ”
              << it->value() << " +/- " << std::sqrt(it->variance()) << std::endl;
  }
}
/* program output: (note that under- and overflow bins appear at the end)
bin 0 x in [-1.0, -0.5): 0 +/- 0
...
bin 5 x in [ 1.5, 2.0): 1 +/- 1
bin 6 x in [ 2.0, inf): 0 +/- 0
bin -1 x in [-inf, -1): 0 +/- 0 */
```

Policy-based design

a variant of static polymorphism

histogram<*typename* **Axes**, *typename* **Storage**>

- Host class
- Defines public n-dimensional interface
- Converts n-dimensional index to internal sequential counter address

Options for **Axes = Sequence of axis types**

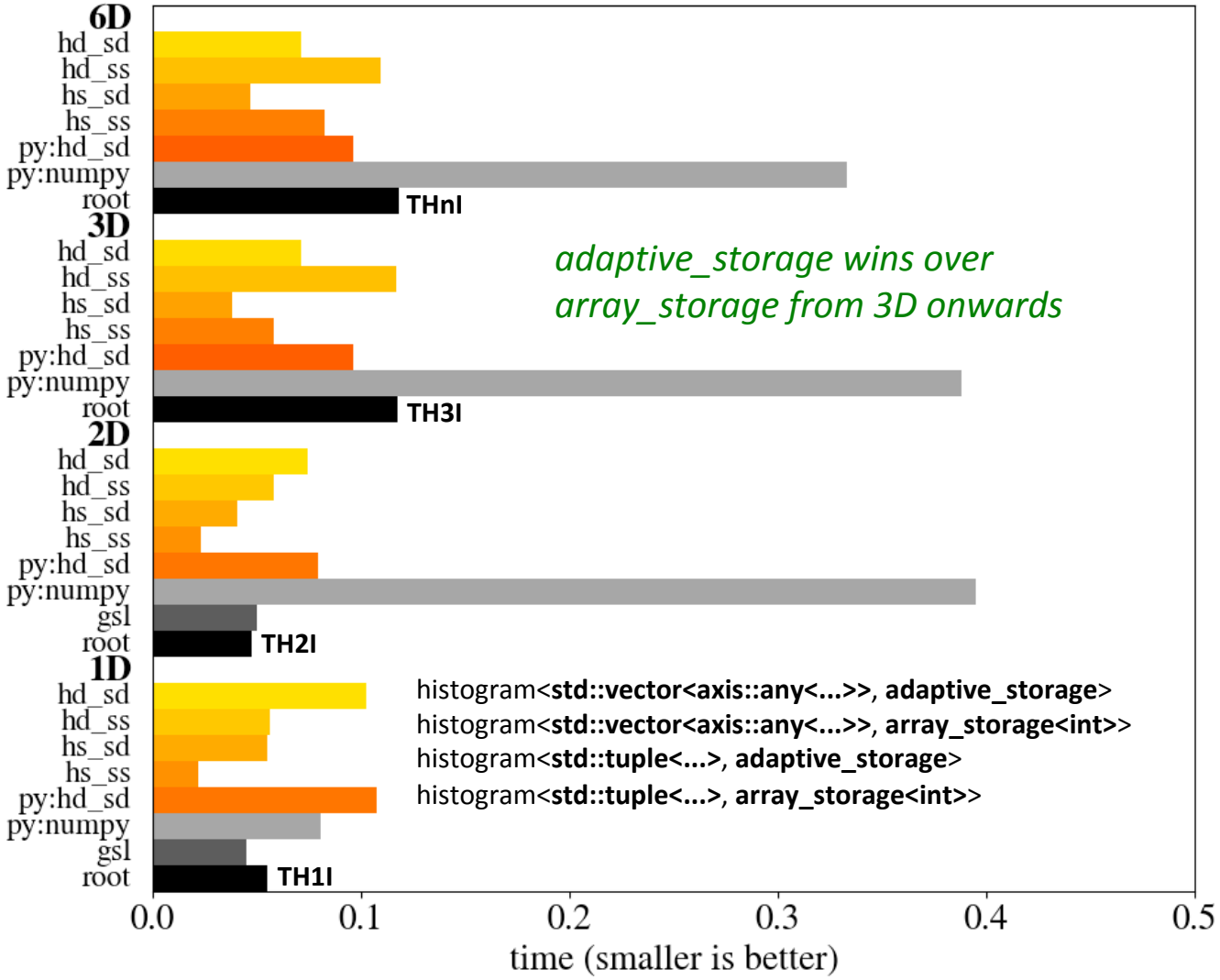
- Static sequence **std::tuple<...>**
  - When number and axis types are known at compile-time
  - Very fast execution speed
- Dynamic sequence **std::vector<axis::any<...>>**
  - When number and axis types are only known at run-time
  - Reduced code execution speed (about a factor 2)
  - Python-bindings require this

Options for **Storage**

- Static counters **array_storage<T>**
  - Full control over counter type, but...
  - Choice of T may not be safe/efficient and difficult to predict before seeing the data
- Dynamic counters **adaptive_storage**
  - Cannot overflow
  - Adaptive memory consumption
  - Runtime cost over-compensated by better utilization of CPU cache
- *Add your own, e.g. mmap'd file, stack-based buffer, ...*

Benchmarks

2.9 GHz Macbook Pro, 1 million bins placed along 1, 2, 3, and 6 dimensions



Histogram interface

- Make histogram with factory functions

  template <typename... Ts> *HistogramType* make_static_histogram(Ts... **ts**)
  template <typename... Ts> *HistogramType* make_dynamic_histogram(Ts... **ts**)
  (+ variants that allow to specific storage type)

- Fill histogram

  template <typename... Ts> void operator()(Ts... **ts**)
  template <typename... Ts> void operator()(weight_type<U> **w**, Ts... **ts**)

- Access i-th axis

  AxisType axis(*CompileTimeNumber*) const
  axis::any<...> axis(unsigned **i**) const

- Access bin

  element_type at(Ts... **ts**) const
  element_type operator[](T **index**) const

- Iterate over bins

  const_iterator begin() const
  const_iterator end() const

Returns fancy iterator with extra methods to access bin information

AxisType concept

- Functor which maps values to bin indices
- Optionally supports labels
- Optionally supports extra bins for under-/overflow
- Optionally is streamable/serializable

```
int operator()(value_type x) const
```
converts from value to index

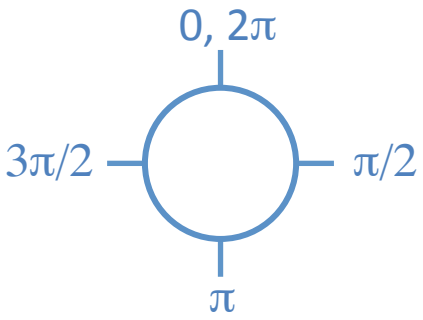
```
bin_type operator[](int i) const
```
converts from index to bin type

Almost arbitrary bin_type, may represent interval or single value

Built-in axis types

- **regular**
  regular<>(10, 0, 1)
  regular<transform::log>(10, 1, 1e5)
  ...

- **circular**
  circular<>(4, 0, two_pi)



- **variable**
  variable<>({0.1, 0.3, 0.9}) : 2 bins [0.1, 0.3), [0.3, 0.9)

- **integer**
  integer<>(3, 6) : 3 bins 3, 4, 5

- **category**
  category<std::string>({“red”, “blue”}) : 2 bins “red” and “blue”

Add your own!

StorageType concept

- Fixed-sized container of bin counters
- May manage bin counters dynamically
- Provides read access to counter value
- Allows incrementing counters and adding values
  - Optionally specialize when adding weights
- Supports adding other same-sized storage
- Supports scaling all counters by number
- Optionally is streamable/serializable

```
element_type operator[](std::size_t index) const
void increase(std::size_t index)
template <typename T> void add(std::size_t index, const T& x)
storage_type& operator+=(const storage_type& other) storage_type&
operator*=(const scale_type& x)

template <typename T> void add(std::size_t index, weight_type<T>&& w)
```

Boost review: Sep 17-26

- 30+ emails exchanged
  https://lists.boost.org/Archives/boost/2018/09/243468.php
  https://lists.boost.org/Archives/boost/2018/09/243340.php
- 5 full reviews
  - Very detailed review by Steven Watanabe
  - All in favor, no against
- 20+ new issues, several bugs found o_O
- To-do before final submission
  - Bugs will be fixed
  - Docs will be further improved
  - Some methods will change names for better consistency
  - Default backend will not support weighted fills anymore
  - Features to add
    - Profile support
    - Algorithm support library
    - Support for std::vector, std::array, std::map as storage backends
    - Controlled access to private data (for external serialization code, etc.)

Generalized histogram

- Traditional histogram only accepts numbers
- Boost.histogram can accept any type: numbers, strings, user types...
  - User needs to provide specialized axis type
  - Built-in axis types templated to handle many possibilities

- Unify histograms, weighted histograms, and profiles
  - Use freedom of specializing counter types
  - Histograms: N
    - Semantics handled by standard integral types
  - Weighted histograms: Σw, Σw^2
    - Semantics handled by built-in weight_counter type
  - Profiles: N, Σx, Σx^2
    - Semantics will be handled by built-in profile_counter type
  - Should allow arbitrary Boost.Accumulators as counters

Summary and Outlook

- boost.histogram
  - Header-only C++11, only Boost as dependency
  - Source: https://github.com/hdembinski/histogram
  - Docs: http://hdembinski.github.io/histogram/doc/html

- histogram-python
  - Will follow once boost.histogram is complete
  - Will use pybind11 and copy of required Boost headers (no full Boost installation required)
  - Source: https://github.com/hdembinski/histogram-python
  - Contributions & collaboration welcome
    - Optional hdf5 serialization wanted
    - Platform-independent binary pickle?

Backup

Python example

Based on former Python bindings

```
import histogram as bh
import numpy as np
h = bh.histogram(
    bh.axis.regular(10, 0.0, 5.0, "radius", uoflow=False),
    bh.axis.circular(4, 0.0, 2 * np.pi, "phi")
)
x = np.random.randn(1000) # generate x
y = np.random.randn(1000) # generate y
radius = (x ** 2 + y ** 2) ** 0.5
phi = np.arctan2(y, x)

h(radius, phi)

count_matrix = np.asarray(h) # access histogram counts (no copy)

print(count_matrix)
# program output:
# [[37 26 33 37]
#  [60 69 76 62]
#  ...
#  [ 0 1 0 0]
#  [ 0 0 0 0]]
```