YODA2: the Rise of Multi-Differential Scalability in Statistical Analysis

Christian Gütschow

MCnet Youngsters' meeting

07 December 2023

Introduction

- histograms are a powerful tool and often taken for granted
- summary statistics grouped into binned ranges of e.g. an independent variable
- fixed data size regardless of how many “fill" events are aggregated into them
- directly linked to core concepts in differential and integral calculus
- unique in HEP: “running average" of fills as opposed to all-events-at-once approach as implemented in `numpy` or `Excel`

Summary statistics

Analytic first- and second-order statistical moments for probably density function $f(x) \equiv \mathrm{d}P/\mathrm{d}x$

$$\langle x \rangle \equiv \int_{x \in X} x f(x) \mathrm{d}x$$

$$\langle x^2 \rangle \equiv \int_{x \in X} x^2 f(x) \mathrm{d}x$$

$$\sigma^2(x) \equiv \langle x^2 \rangle - \langle x \rangle^2$$

Unweighted moments

Unweighted mean and variance for finite-size sample with $1 \leq n \leq N$:

$$\langle x \rangle_{\mathrm{U}} \equiv \frac{\sum_{n=1}^{N} x_n}{N}$$

$$\begin{aligned}\sigma_{\mathrm{U}}^2(x) &\equiv \frac{\sum_{n=1}^{N} (x_n - \langle x \rangle)^2}{N-1} \\ &= \langle x^2 \rangle_{\mathrm{U}} - \langle x \rangle_{\mathrm{U}}^2 \\ &= \frac{\sum_{n=1}^{N} x_n^2}{N-1} - \frac{\left(\sum_{n=1}^{N} x_n\right)^2}{(N-1)^2}\end{aligned}$$

- ➜ Poisson mean and variance closely related quantities, both given by the count N
- ➜ classic Monte Carlo scaling then given by $\sigma_{\mathrm{P}}(x)/\langle x \rangle_{\mathrm{P}} = \sqrt{N}/N = 1/\sqrt{N}$

Weighted moments

Weighted mean and variance:

$$\langle x \rangle = \frac{\sum_n w_n x_n}{\sum_n w_n}$$

$$\sigma^2(x) = \frac{\sum_n w_n \left(x_n - \sum_m w_m x_m\right)^2}{\left(\sum_n w_n\right) - 1} = \frac{\left(\sum_n w_n x_n^2\right) \cdot \sum_n w_n - \left(\sum_n w_n x_n\right)^2}{\left(\sum_n w_n\right)^2 - \sum_n w_n^2}$$

- effective number of entries given by $N_{\text{eff}} = \left(\sum_n w_n\right)^2 / \sum_n w_n^2$
- effective variance given by $\sigma^2_{\text{eff}}(x) = \left(\sum_n w_n x_n\right)^2 / \sum_n w_n$
- histogram: binned approximation to continuous probability density in binned variable space
- profile: mean and standard error of a dependent variable
  as a function of binning coordinates
- covariance in multiple binning dimensions also requires binning cross-terms

Design principles

- Separation of style from subtance
  - invariance of statistical data across rendered representations of data
- Differential consistency
  - a histogram is not a bar chart but the best density estimate for a continuous distribution $\delta N/\delta\Omega$
  - $\delta N/\delta\Omega = \left[N(\Omega+\delta\Omega) - N(\Omega)\right]/\delta\Omega \stackrel{\delta\Omega\to 0}{=} \mathrm{d}N/\mathrm{d}\Omega$ **necessitates division by bin width**
- Weighted statistical moments
  - weighted statistical moments required to compute the key summary statistics of their bins
  - a *profile* also stores the statistical moments of a further unbinned quantity
- Integral consistency
  - high-dimensional binnings should be reducable to lower-dimension objects without biasing integral quantities
- partially established already at the time of YODA1 release in 2013, but structural issues motived a ground-up rewrite

Limitations of YODA1

- limited data-object dimensionality and only continous-valued axes supported
- inability to store arbitrary data-types in binnings
- correct but limited treatment of overflow bins
- no unified scheme for local and global bin indexing in multiple dimensions
- internal code duplication to support C++ and Python APIs for several different dimensionalities and binned-content types
- mismatching of the “inert" scatter datatype from e.g. HepData to the binned “live” objects from MC runs
- limited and inconvenient implementation of uncertainty breakdowns and correlations on scatter types

Variadic templates and parameter packs

➜ Metaprogramming using C++17 takes care of generalisation to arbitrary dimensions:

```
#include <iostream>
#include <string>
#include <tuple>
#include <vector>

template <typename... Args>
class MyHisto {
public:
  MyHisto(const std::vector<Args>& ... edges)
    : _axes(edges ...) { }

  size_t dim() const { return sizeof...(Args); }

  template<size_t I>
  void printBinning() const {
    if constexpr (I < sizeof...(Args)) {
      std::cout << "Axis" << (I+1) << "has";
      std::cout << std::get<I>(_axes).size();
      std::cout << "bins." << std::endl;
      printBinning<I+1>();
    }
  }

  void print() const {
    std::cout << dim() << "D:" << std::endl;
    printBinning<0>();
  }

private:
  std::tuple<std::vector<Args>...> _axes;
};
```

Binning, Axes, Indexing

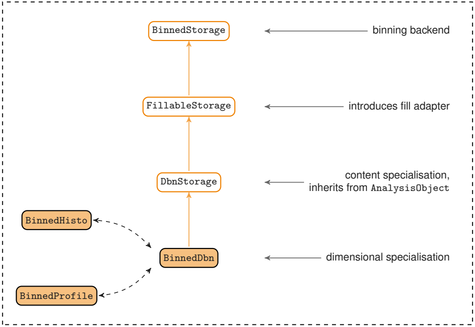
- new `Axis` class templated on edge type
- (classic) continuous axis for floating point edges
  - N bins defined by $N + 1$ edges, plus under- and overflow bin
  - infinity binning:
    bin edges: `-inf -1.0 -0.5 0.0 0.5 1.0 +inf`
    bin widths: `+inf 0.5 0.5 0.5 0.5 +inf`
  - bin along continuous axis has lower/upper edge, midpoint, width
- (new) discrete axis for all other types (useful for multiplicities, cutflows, . . .)
  - bins along discrete axis only have their edge label
  - N bins defined by N edges, plus otherflow bin
- `Binning` class to translate local indices into a global index and vice versa
- `Bin` wrapper class that links bin content with the local and global binning properties
  - every bin has a `dVol()` method (also `dLen()`, `dArea()` aliases in 1D and 2D)

Natively supportent bin-content types

- `Dbn`
  - the classic YODA1 distribution, now generalised to arbitrary dimensions
  - keeps track of *exact* first and second order moments
- `Estimate`
  - a central value with an associated error breakdown
  - errors encoded as labelled uncertainty pairs corresponding to {down,up} variations of a nuisance parameter
  - support for correlated/uncorrelated treatment of different NPs
  - arithmetic operations respect (un-)correlated error treatment
- `Point`
  - used to represent marker coordinates on a canvas

A generic storage for binned quantities

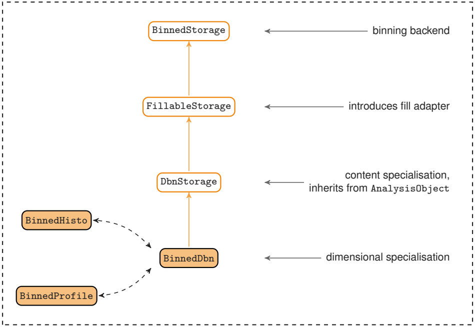
- new `BinnedStorage` class can hold arbitrary types
  - supports index-based `bin(i)` and coordinate-based `binAt(x)` lookups
  - supports bin masking (`mask(i)`, `maskAt(x)`) to emulate “gaps" (in place of bin erasure)



- new `FillableStorage` class inherits from `BinnedStorage`
  - introduces a fill adapter that handles the bin-content manipulation for each `fill` call
  - `fill` function returns bin position (global index) or `-1` if a coordinate was `NaN`
  - still supported from YODA1: fractional fills

Dimensional specialisations

- intermediate `DbnStorage` layer introduces `Dbn`-specific methods (e.g. global integral, variance etc.)
- `BinnedDbn` is the user-facing type with various aliases for familiar classes
  - mixes in axis-specific method names (`xMean()`, `yEdges()`, etc.)



- `BinnedHisto<double,int> = BinnedDbn<2,double,int>`
- `BinnedProfile<string> = BinnedDbn<2,string>`
- `Histo2D = HistoND<2> = BinnedHisto<double,double> = BinnedDbn<2,double,double>`
- `Profile1D = ProfileND<1> = BinnedProfile<double> = BinnedDbn<2,double>`

Example: construction and filling

```
// declaration examples
Histo1D h1; // histogram with 1 continuous axis
Profile2D p1; // profile with 2 continuously binned axes + 1 unbinned axis
HistoND<5> h2; // histogram with 5 continuous axes
```

```
// constructor examples
Histo1D h3(10, 0, 100); // 10 bins between 0 and 100
const std::vector<double> edges = {0, 10, 20, 30, 40, 50};
Histo1D h4(edges);
BinnedHisto<int, std::string> h5({ 1, 2, 3 }, { "A", "B", "C" });
```

```
// fill examples
Histo1D h6(5, 0.0, 1.0);
h6.fill(0.2);
Profile1D p2(5, 0.0, 1.0);
p2.fill(0.2, 3.5);
```

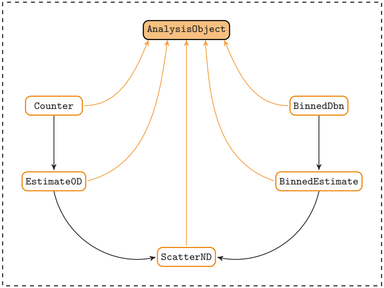
```
// marginalisation examples
Histo2D h7 = p1.mkHisto(); //< marginalise over unbinned axis
Histo1D h8 = h7.mkMarginalHisto<2>(); //< marginalise over secomd binned axis
Histo1D h9 = p1.mkMarginalProfile<1>(); //< marginalise over first binned axis
```

Example: looping and indexing

```
size_t nbinsX = 4, nbinsY = 6;
double lowerX = 0, lowerY = 0;
double upperX = 4, upperY = 6;
Histo2D h2(nbinsX, lowerX, upperX,
           nbinsY, lowerY, upperY);

// loop over bins and fill with increasing weight
double w = 0;
for (auto& b : h2.bins()) { //< iterators passes through using templated bin wrappers
  h2.fill(b.xMid(), b.yMid(), ++w);
}

for (size_t idxY = 0; idxY < h2.numBinsY(true); ++idxY) { //< true includes overflows
  for (size_t idxX = 0; idxX < h2.numBinsX(true); ++idxX) { //< true includes overflows
    std::cout << "\t(" << idxX << "," << idxY << ")\t=\t";
    std::cout << h2.bin(idxX, idxY).sumW();
  }
  std::cout << std::endl;
}
std::cout << std::endl;

# H2 bins using local indices + under/overflows:
#   (0,0) = 0 (1,0) =  0 (2,0) =  0 (3,0) =  0 (4,0) =  0 (5,0) = 0
#   (0,1) = 0 (1,1) =  1 (2,1) =  2 (3,1) =  3 (4,1) =  4 (5,1) = 0
#   (0,2) = 0 (1,2) =  5 (2,2) =  6 (3,2) =  7 (4,2) =  8 (5,2) = 0
#   (0,3) = 0 (1,3) =  9 (2,3) = 10 (3,3) = 11 (4,3) = 12 (5,3) = 0
#   (0,4) = 0 (1,4) = 13 (2,4) = 14 (3,4) = 15 (4,4) = 16 (5,4) = 0
#   (0,5) = 0 (1,5) = 17 (2,5) = 18 (3,5) = 19 (4,5) = 20 (5,5) = 0
#   (0,6) = 0 (1,6) = 21 (2,6) = 22 (3,6) = 23 (4,6) = 24 (5,6) = 0
#   (0,7) = 0 (1,7) =  0 (2,7) =  0 (3,7) =  0 (4,7) =  0 (5,7) = 0
```

Overview of user-facing types

- live `BinnedDbn` objects reduce to inert `BinnedEstimate` objects
  - with `Estimate1D = EstimateND<1> = BinnedEstimate<double>`
  - slice along axis `n` using `EstimateND<N>().mkEstimates<n>();` to yield vector of `EstimateND<N-1>`
- 0-dimensional variants with live `Counter` reducing to `Estimate0D`



- both live and inert types reduce to `Scatter` objects for plotting
- all user-facing types inherit from the `AnalysisObject` base class, which provides the attribute system to store metadata
- all types support global scaling operations; arbitrary transformations (e.g. lambda functions) can also be applied to all *inert* data types (estimates, points)

YODA ASCII V3

- generalising the existing V2 ASCII format to arbitrary dimensions and supporting `std::string`-based edges required a little restructuring:

```
BEGIN YODA_HISTO1D_V3 /H1D_d
Path: /H1D_d
Title:
Type: Histo1D
---
# Mean: 3.470588e-01
# Integral: 1.700000e+01
Edges(A1): [0.000000e+00, 5.000000e-01, 1.000000e+00]
# sumW          sumW2           sumW(A1)        sumW2(A1)       numEntries
0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00
1.000000e+01    1.000000e+02    1.000000e+00    1.000000e-01    1.000000e+00
7.000000e+00    4.900000e+01    4.900000e+00    3.430000e+00    1.000000e+00
0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00
END YODA_HISTO1D_V3

BEGIN YODA_BINNEDHISTO<S>_V3 /H1D_s
Path: /H1D_s
Title:
Type: BinnedHisto<s>
---
# Mean: 3.750000e-01
# Integral: 8.000000e+00
Edges(A1): ["A"]
# sumW          sumW2           sumW(A1)        sumW2(A1)       numEntries
5.000000e+00    2.500000e+01    0.000000e+00    0.000000e+00    1.000000e+00
3.000000e+00    9.000000e+00    3.000000e+00    3.000000e+00    1.000000e+00
END YODA_BINNEDHISTO<S>_V3
```

- already the default on HepData! (old format still available via `YODA1` option)

- YODA2 reader can still read old ASCII layout from YODA1

Support of YODA2 in Rivet

- Rivet will adopt YODA2 starting with its 3.2 series
  - all reference data shipped with Rivet has been converted to the new types
  - HepData already supports YODA2 by default: writes out `BinnedEstimate` objects
- `TypeRegister`: edge combination of `double`, `int` and `string` pre-registered for 1D and 2D objects, others can be registered on the fly:
  - `RIVET_REGISTER_TYPE(YODA::BinnedHisto<double,int,string,double>)`
  - `RIVET_REGISTER_BINNED_SET(double, double, string, int)`
- routines adjusted to use discrete binning where appropriate
- Rivet's custom `BinnedHistogram` class got replaced with a `HistoGroup` class (a `FillableStorage` with a "group axis" and a `BinnedHisto` as bin content)

```
Histo1DGroupPtr _hist; //< Histo1DGroup = HistoGroup<double,double>
...
book(_hist, { 1.0, 2.0, 3.0, 4.0 });
for (auto& bin : hist->bins()) {
  book(bin, 1, 1, bin.index());
}
...
_hist->fill(val1, val2);
...
normalize(_hist); // or: scale(_hist, crossSection()/sumOfWeights());
divByGroupWidth(_hist); // divide by bin width along group axis
```

Better HPC support

- YODA2 inheritance structure makes it straightforward to `serialize` the data
  - numerical content of `AnalysisHandler` can be translated into `std::vector<double>`
  - arrays of primative types lend themselves better to MPI communication
- corresponding `deserialize` method to load the data block back into an `AnalysisHandler` for merging
- reduced I/O load from parsing info files in the initialisation phase
- more profiling and optmisations envisaged post-3.2.0 release
  - HPC-friendly HDF5-based output format in the works (as alternative to ASCII)

New plotting system

- `matplotlib`-based plotting already available since 3.1.8
- will become default from 3.2.0 (old-style plotting still available via `rivet-mkhtml-tex`)
- replaces old in-house `dat` “format” replaced with **self-consistent Python scripts** allowing for better customisation of plots (no YODA installation required)
- plots drawn from `Scatter` objects
  - final abstraction layer to seperate style choices for renderinf data from statistical analysis

Summary

- a decade after its first release, YODA backend underwent a ground-up redesign
- statistical analysis objects generalised to arbitrary dimensions and edge types along different axes – with the help of modern C++ design patterns
- the YODA 2.0.0 alpha release has been out for a few weeks, first production-ready release expected to follow shortly
- at the same time the Rivet 3.2 series migrates its histogramming system to YODA2 which allows many simplifications and API improvements