Consistent multi-differential histogramming and summary statistics with YODA2

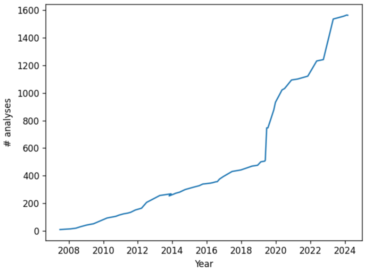
Andy Buckley (Glasgow), Louie Corpe (Clermond),
Matthew Filipovich (Oxford) Christian Gütschow (UCL),
Nick Rozinsky (Glasgow), Simon Thor (KTH),
Yoran Yeh (UCL), Jamie Yellen (Glasgow)

ACAT2024, Stony Brook

14 March 2024

Introduction

- Yet more Objects for Data Analysis!
  [yoda.hepforge.org]
- lightweight and general purpose library for binned statistical data analysis
- first released in 2013
- written in C++ and programmatically usable from C++ and Python, complemented by a set of command-line tools for dataset inspection, manipulation and combination



- emerged from the sub-field of Monte Carlo event generator analysis and tuning in HEP, but library is deliberately agnostic of any particular application
- tools wrapping around YODA include [Rivet] and [Contur]
  - as such, widely used for event generator analysis, tuning, analysis preservation and reinterpretation efforts

Summary statistics

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

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

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

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

Weighted moments

Weighted mean and variance:

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

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

with weighted Bessel factor:

$$\mathcal{B} = \frac{N_\text{eff}}{N_\text{eff} - 1} = \frac{(\sum_n w_n)^2}{(\sum_n w_n)^2 - \sum_n w_n^2}$$

for effective fill count:

$$N_\text{eff} = \frac{(\sum_n w_n)^2}{\sum_n w_n^2}$$

Histograms

- generalise measured variable x to vector variable-space Ω
  - composed of vectors ω with differential volume elements $\mathrm{d}\Omega$
- partition Ω into disjoint (sub)set of bins $\{\Omega_b\} \subset \Omega$

Histograms

- generalise measured variable x to vector variable-space Ω
  - composed of vectors ω with differential volume elements $\mathrm{d}\Omega$
- partition Ω into disjoint (sub)set of bins $\{\Omega_b\} \subset \Omega$
- moments in each bin b converge to summary properties of that bin's variable-space partition

$$\langle \omega^{(i)} \rangle_b \equiv \int_{\omega \in \Omega_b} \omega^{(i)} f(\omega)\, \mathrm{d}\Omega$$

$$\langle \omega^{(i)} \omega^{(j)} \rangle_b \equiv \int_{\omega \in \Omega_b} \omega^{(i)} \omega^{(j)} f(\omega)\, \mathrm{d}\Omega$$

Histograms

- generalise measured variable x to vector variable-space Ω
  - composed of vectors ω with differential volume elements $\mathrm{d}\Omega$
- partition Ω into disjoint (sub)set of bins $\{\Omega_b\} \subset \Omega$
- moments in each bin b converge to summary properties of that bin's variable-space partition

$$\langle \omega^{(i)} \rangle_b \equiv \int_{\omega \in \Omega_b} \omega^{(i)} f(\omega)\, \mathrm{d}\Omega$$

$$\langle \omega^{(i)} \omega^{(j)} \rangle_b \equiv \int_{\omega \in \Omega_b} \omega^{(i)} \omega^{(j)} f(\omega)\, \mathrm{d}\Omega$$

- need to recover unbinned values when expanding the partition to whole space
- need to recover differential properties of the pdf itself as $\Omega_b \to \mathrm{d}\Omega(\omega)$
- merging bins must converge to the same result as having originally constructed a lower-dimensional or less finely binned partition of space

Design principles I

- Differential consistency
  - unlike list of (weighted) fill counts, histogram is a binned best-estimate of a continuous distribution
  - crucial to take $f(\boldsymbol{x}) \equiv \mathrm{d}P/\mathrm{d}\boldsymbol{x}$ notation literally since optimal estimation requires non-uniform binning

Design principles I

- Differential consistency
  - unlike list of (weighted) fill counts, histogram is a binned best-estimate of a continuous distribution
  - crucial to take $f(\boldsymbol{x}) \equiv \mathrm{d}P/\mathrm{d}\boldsymbol{x}$ notation literally since optimal estimation requires non-uniform binning
- Continuous aggregation
  - histograms need to be “live” objects containing update-able variables
  - single pass over all events in memory à la `numpy` or `Excel` often not feasible in HEP

Design principles I

- Differential consistency
  - unlike list of (weighted) fill counts, histogram is a binned best-estimate of a continuous distribution
  - crucial to take $f(\boldsymbol{x}) \equiv \mathrm{d}P/\mathrm{d}\boldsymbol{x}$ notation literally since optimal estimation requires non-uniform binning
- Continuous aggregation
  - histograms need to be “live” objects containing update-able variables
  - single pass over all events in memory à la `numpy` or `Excel` often not feasible in HEP
- 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

Design principles I

- Differential consistency
  - unlike list of (weighted) fill counts, histogram is a binned best-estimate of a continuous distribution
  - crucial to take $f(\boldsymbol{x}) \equiv \mathrm{d}P/\mathrm{d}\boldsymbol{x}$ notation literally since optimal estimation requires non-uniform binning
- Continuous aggregation
  - histograms need to be “live” objects containing update-able variables
  - single pass over all events in memory à la `numpy` or `Excel` often not feasible in HEP
- 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
  - ability to project higher- into lower-dimensional binnings without biasing integral quantities
  - including integrally consistent constructions of binned profiles from higher-dimensional histograms

Design principles II

- Separation of style from substance
  - invariance of statistical data while varying plotting style

Design principles II

- Separation of style from substance
  - invariance of statistical data while varying plotting style
- Separation of binning from bin-content
  - enables distinction between *live* (permits further data-taking) and *inert* classes of data-object, with the latter being a specific representation as "values and uncertainties"

Design principles II

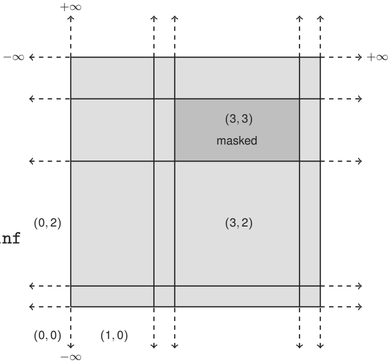
- Separation of style from substance
  - invariance of statistical data while varying plotting style
- Separation of binning from bin-content
  - enables distinction between *live* (permits further data-taking) and *inert* classes of data-object, with the latter being a specific representation as “values and uncertainties”
- User friendliness
  - aim to provide a “clean” programmatic interface expressed in terms of statistical and data-analytic concepts and hence well-matched to the goals and skill-sets of data scientists
  - hide the complexity of advanced language features used internally to make high levels of abstraction possible while enforcing statistical consistency and type-safety
  - intentionally limited to binned statistical analysis only, with zero library dependencies for core C++ operation, to assist embedding into applications

Experience from YODA1

- design goals partially established already at the time of YODA1 release in 2013, but structural issues motivated a ground-up rewrite
- limited data-object dimensionality and only continuous-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

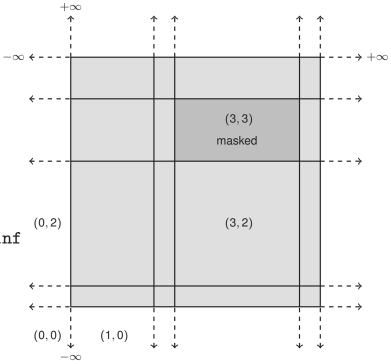
Bin partitioning

- new `Axis` class templated on edge type
- (classic) continuous axis triggered by `std::is_floating_point` trait
  - N bins defined by $N+1$ edges, plus under- and overflow bin
  - active uses of IEEE 754 FP standard; 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 partitioning

- new `Axis` class templated on edge type
- (classic) continuous axis triggered by `std::is_floating_point` trait
  - N bins defined by $N+1$ edges, plus under- and overflow bin
  - active uses of IEEE 754 FP standard; 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`



- (new) discrete axis for all other types
  - bins along discrete axis only have their edge label
  - N bins defined by N edges, plus otherflow bin
  - useful for multiplicities, cutflows, . . .
- `Binning` class permits slicing and marginalisaing across global fill-space and translates local indices into a global index and vice versa

Bin content

- `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)
  - access to axis-specific quantities via templated accessor methods
  - CRTP used to mix in axis-specific method names for first three dimensions

Bin content

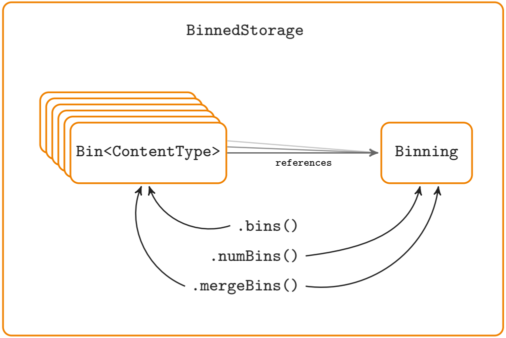
- `Bin` wrapper class that links bin content with the local and global binning properties
  - every bin has a `dVol()` method (also `dLen()`, `dArea()` aliases in 1D and 2D)
  - access to axis-specific quantities via templated accessor methods
  - CRTP used to mix in axis-specific method names for first three dimensions
- Live content: `Dbn`
  - distribution class from YODA1, now generalised to arbitrary dimensions
  - keeps track of *exact* first and second order moments (and mixed moments $\sum_n w_n x_n y_n$)
  - fill provides `fill` method accepting next coordinate set, optional weight and optional fill fraction

Bin content

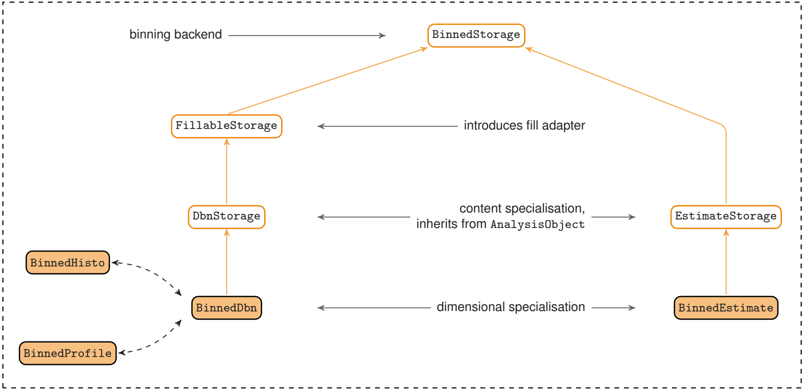
- `Bin` wrapper class that links bin content with the local and global binning properties
  - every bin has a `dVol()` method (also `dLen()`, `dArea()` aliases in 1D and 2D)
  - access to axis-specific quantities via templated accessor methods
  - CRTP used to mix in axis-specific method names for first three dimensions
- Live content: `Dbn`
  - distribution class from YODA1, now generalised to arbitrary dimensions
  - keeps track of *exact* first and second order moments (and mixed moments $\sum_n w_n x_n y_n$)
  - fill provides `fill` method accepting next coordinate set, optional weight and optional fill fraction
- Inert content: `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

Combined bin partitioning and content

- 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)



A generic storage for binned quantities



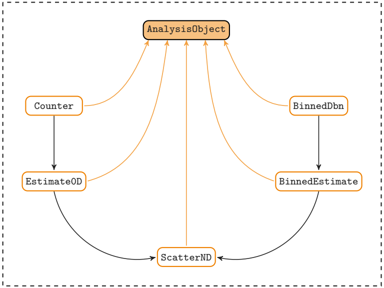
- new `FillableStorage` class inherits from `BinnedStorage`
  - introduces a fill adapter that handles the bin-content manipulation for each `fill` call
  - `fill` function returns bin position (global index) or `-1` if a coordinate was `nan`

Standard histograms and profiles

- 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>

Type and dimensionality reductions

- 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)

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<1>(); //< marginalise over secomd binned axis
Histo1D h9 = p1.mkMarginalProfile<0>(); //< 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
```

YODA I/O

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

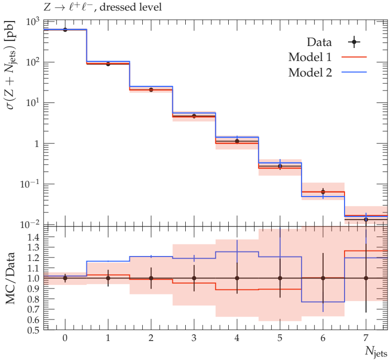
```
BEGIN YODA_HISTO1D_V3 /H1D_d
Path: /H1D_d
Title:
Type: Histo1D
---
# Mean: 3.470588e-01
# Integral: 1.700000e+01
Edges(A1): [0.000000e+00, 5.000000e-01, 1.000000e+00]
# sumW          sumW2           sumW(A1)        sumW2(A1)       numEntries
0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00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

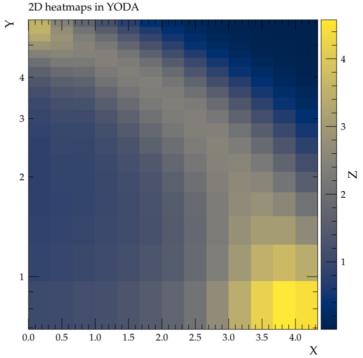
- already the default on HepData! (old format still available via `YODA1` option)

- YODA2 reader can still read old ASCII format from YODA1

Plotting

- `matplotlib`-based plotting machinery produces **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 rendering data from statistical analysis

Summary

- 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
- 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
- YODA 2.0.0 has been out for a couple of months now – check it out: [yoda.hepforge.org]

Backup

Unweighted moments

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

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

$$\sigma^2_\mathrm{U}(\hat{x}) \equiv \frac{\sum_{n=1}^{N} (x_n - \langle x \rangle)^2}{N-1}$$

$$= \langle x^2 \rangle_\mathrm{U} - \langle x \rangle^2_\mathrm{U}$$

$$= \frac{\sum_{n=1}^{N} x_n^2}{N-1} - \frac{\left(\sum_{n=1}^{N} x_n\right)^2}{(N-1)^2}$$

Counts and efficiencies

Closely related quantities are Poisson mean and variance:

$$\langle \hat{x} \rangle_{\mathrm{P}} \equiv N$$

$$\sigma^2_{\mathrm{P}}(\hat{x}) \equiv N$$

Classic Monte Carlo scaling then given by

$$\frac{\sigma_{\mathrm{P}}(\hat{x})}{\langle \hat{x} \rangle_{\mathrm{P}}} = \frac{\sqrt{N}}{N} = \frac{1}{\sqrt{N}}$$

Sample efficiency for selected events N_{sel} from a known number of total events N is

$$\hat{\epsilon} \equiv \frac{N_{\mathrm{sel}}}{N}$$

Binomial statistics gives an estimator for the uncertainty on the efficiency

$$\hat{\sigma}^2(\hat{\epsilon})_{\mathrm{B}} = \frac{\hat{\epsilon}(1 - \hat{\epsilon})}{N}$$

Connection to differential calculus

- statistical histogram: a **discrete approximation to entire probability density function** $f(\Omega) = \mathrm{d}P/\mathrm{d}\Omega$ or population density $\mathrm{d}N/\mathrm{d}\Omega$, not just a collection of fill counts

Connection to differential calculus

- statistical histogram: a **discrete approximation to entire probability density function** $f(\Omega) = \mathrm{d}P/\mathrm{d}\Omega$ or population density $\mathrm{d}N/\mathrm{d}\Omega$, not just a collection of fill counts
- bin measure $\mathrm{d}\Omega$ (or $\Delta\Omega$) representing the volume element of the bin
  crucial for differential consistency

Connection to differential calculus

- statistical histogram: a **discrete approximation to entire probability density function** $f(\Omega) = \mathrm{d}P/\mathrm{d}\Omega$ or population density $\mathrm{d}N/\mathrm{d}\Omega$, not just a collection of fill counts
- bin measure $\mathrm{d}\Omega$ (or $\Delta\Omega$) representing the volume element of the bin crucial for differential consistency
- $\Delta N/\Delta\Omega = [N(\Omega + \Delta\Omega) - N(\Omega)]\,/\Delta\Omega \overset{\Delta\Omega\to 0}{=} \mathrm{d}N/\mathrm{d}\Omega$ **necessitates division by bin width**
  - generally not desirable for finite bins to have the same width
  - using non-uniform bin sizes ensures statistical relative uncertainty on bin populations is equally distributed across histogram
  - failing to divide by the bin measure distorts the distribution away from its physical shape

Connection to differential calculus

- statistical histogram: a **discrete approximation to entire probability density function** $f(\Omega) = \mathrm{d}P/\mathrm{d}\Omega$ or population density $\mathrm{d}N/\mathrm{d}\Omega$, not just a collection of fill counts
- bin measure $\mathrm{d}\Omega$ (or $\Delta\Omega$) representing the volume element of the bin crucial for differential consistency
- $\Delta N/\Delta\Omega = \left[N(\Omega + \Delta\Omega) - N(\Omega)\right]/\Delta\Omega \overset{\Delta\Omega\to 0}{=} \mathrm{d}N/\mathrm{d}\Omega$ **necessitates division by bin width**
  - generally not desirable for finite bins to have the same width
  - using non-uniform bin sizes ensures statistical relative uncertainty on bin populations is equally distributed across histogram
  - failing to divide by the bin measure distorts the distribution away from its physical shape
- actual bin populations are better computed using a discrete binning expressed in terms of finite probabilities rather than densities
  - awkward workaround: multiply each density by the fill volume
  - prefer to refer to this not as a histogram but a bar chart, reflecting its typical use

Profiles

- useful class of histogram mixing binned and unbinned variable subspaces
- allow characterisation of the unbinned dimensions Υ via their moments as projected into each partition of the bin-space Θ
  - allow statistical aggregation of finite samples into "independent variable" bins $\theta \in \Theta_b$, while characterising the mean dependence of the unbinned dependent variables y on θ
  - linearity of statistical moments again ensures consistency when merging bins

Profiles

- useful class of histogram mixing binned and unbinned variable subspaces
- allow characterisation of the unbinned dimensions Υ via their moments as projected into each partition of the bin-space Θ
  - allow statistical aggregation of finite samples into "independent variable" bins $\theta \in \Theta_b$, while characterising the mean dependence of the unbinned dependent variables y on θ
  - linearity of statistical moments again ensures consistency when merging bins
- unbinned space Υ can in general be multidimensional but canonical bin value then ambiguous
- definiteness retained for single-dimensional unbinned space with moments $\langle y \rangle$ and $\langle y^2 \rangle$
  - profile canonical bin value is the mean $\langle y(\Theta) \rangle$ as a function of binned coordinates
  - nominal uncertainty given by standard error $\hat{\sigma}_{\bar{y}}(\theta) = \hat{\sigma}_b / \sqrt{N_b}$ for effective sample count N_b in bin $b \subset \theta$

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;
};
```

Support of YODA2 in Rivet4

- Rivet adopted YODA2 starting with its 4.0 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 support for massive MPI applications

- 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 for the Rivet4 series