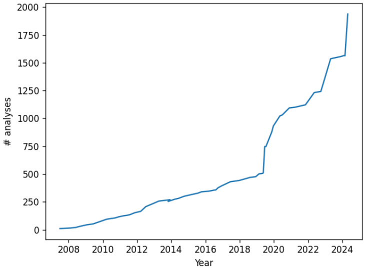
Rivet/YODA status

Christian Gütschow

Pythia Week, Oxford

30 April 2024

Introduction



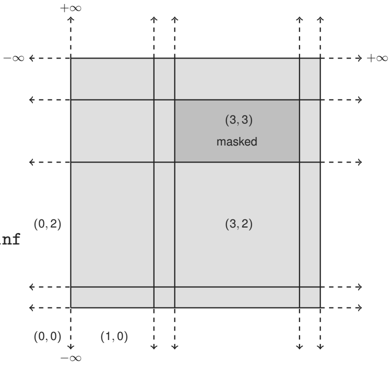
- Rivet4 and YODA2 released!
  [rivet.hepforge.org] [yoda.hepforge.org]
- YODA: lightweight and general purpose library for binned statistical data analysis
- first released in 2013,
  now undergone a ground-up re-design
- see also detailed write-ups for YODA [arxiv:2312.15070] and Rivet [arxiv:2404.15984] respectively
- Rivet now requires `C++17`, drops support for HepMC2 and Python2
- some syntax changes to be expected, cf. [migration guide]

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

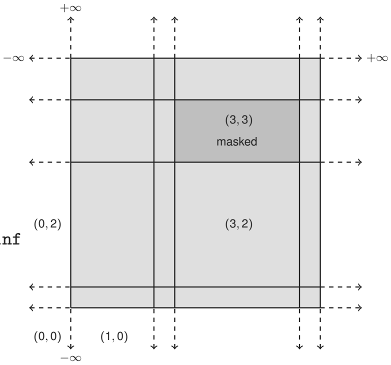
Bin partitioning

- 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

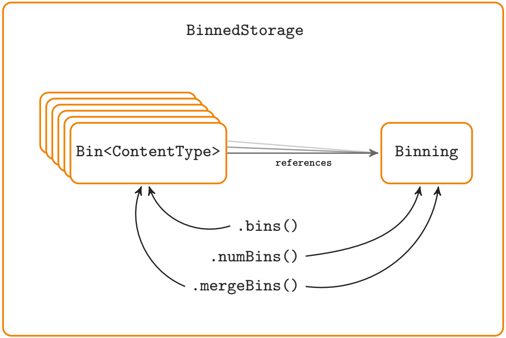
- new `Axis` class templated on edge type
- (classic) continuous axis triggered by `std::is_floating_point` trait
  - N bins defined by $N+1$ edges, plus under- and overflow bin
  - active uses of IEEE 754 FP standard; infinity binning:
    bin edges: `-inf -1.0 -0.5 0.0 0.5 1.0 +inf`
    bin widths: `+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, ...



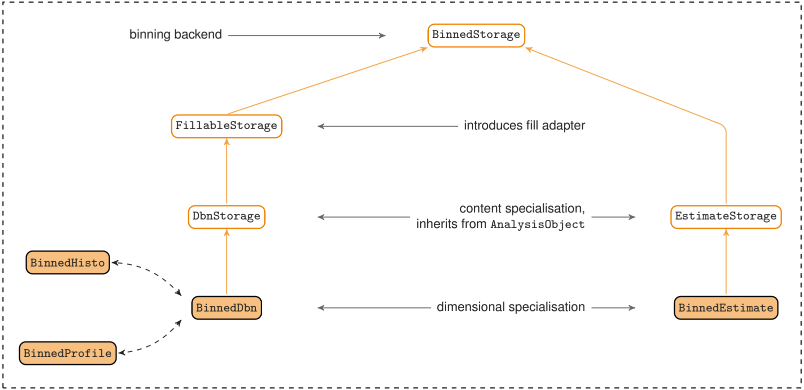
- `Binning` class permits slicing and marginalisaing across global fill-space and translates local indices into a global index and vice versa

A new base class for all binned quantities

- new `BinnedStorage` class can hold arbitrary bin-content 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)



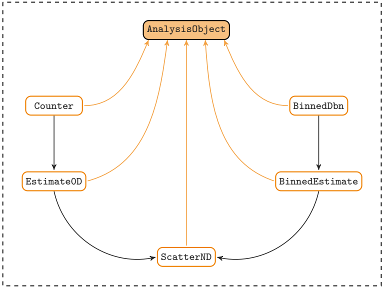
Distinguishing live and inert objects



- 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`

Type and dimensionality reductions

- live `BinnedDbn` objects reduce to inert `BinnedEstimate` objects
  - slice along axis `n` using `mkHistos<n>()` to yield a `vector<Histo1D>` from `Histo2D` etc.
- 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)

Retaining YODA1-style type names

- BinnedDbn<FillDim, BinnedAxisT, ...> now default user-facing object for all live distributions (i.e. histograms and profiles)
- BinnedEstimate<BinnedAxisT, ...> now default user-facing object for all inert types (e.g. HepData measurements)
- syntactic sugaring recovers more familiar/convenient type names, e.g.
  - 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>
  - Estimate1D = EstimateND<1> = BinnedEstimate<double>
  - Scatter2D = ScatterND<2>

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

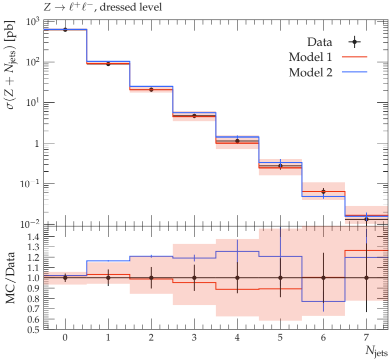
- generalising the existing V2 ASCII format to arbitrary dimensions and supporting `std::string`-based edges required a little restructuring:

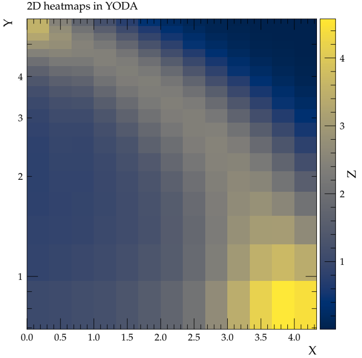
```
BEGIN YODA_HISTO1D_V3 /H1D_d
Path: /H1D_d
Title:
Type: Histo1D
---
# Mean: 3.470588e-01
# Integral: 1.700000e+01
Edges(A1): [0.000000e+00, 5.000000e-01, 1.000000e+00]
# sumW          sumW2           sumW(A1)        sumW2(A1)       numEntries
0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00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

New Rivet4 major release!

- Rivet 4 adopts YODA2 for histogramming backend
  - all reference data shipped with Rivet has been converted to the new `Estimate` types
  - HepData already supports YODA2 by default: writes out `BinnedEstimate` objects
- post-`finalize()` objects written out in their *inert* state!
- `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
- `matplotlib`-based plotting machinery now the default script
  - old script based on LaTeX-`pstricks` still available as `rivet-mkhtml-tex`

Projection streamlining

- Clean-up of projection arguments in favour of self-documenting scoped `enum` classes:
  `FastJets::Algo::KT` → `JetAlg::KT`
  `JetAlg::Muons::NONE` → `JetMuons::NONE`
  `JetAlg::Invisibles::DECAY` → `JetInvisibles::DECAY`
- Boolean arguments for treatment of tau/muon decay products replaced with `TauDecaysAs` and `MuDecaysAs` `enum` classes
- `DressedLepons` renamed `LeptonFinder`, akin to existing `JetFinder` and `ParticleFinder`
  - old `DressedLeptons` now alias for `vector<DressedLepton>`
  - similarly, `ZFinder` renamed `DileptonFinder`
- `WFinder` removed entirely!
  - significantly improves self-documentation of analysis code, clarifying previously obscure model-dependent assumptions woven into the measurement data
  - new `closestMatchIndex()` metafunction to help identify W candidates, e.g.

```
const int bestmatch = closestMatchIndex(leptons, pmiss, Kin::mass, 80.4*GeV);
```

More API updates

- smearing of NLO sub-events now generalised to arbitrary dimensions and axis types
  - originally introduced by Leif in Rivet3 for 1D histograms

- Rivet's custom `BinnedHistogram` class got replaced with a `HistoGroup` class (a `FillableStorage` with a "group axis" and a `BinnedHisto` as bin content)

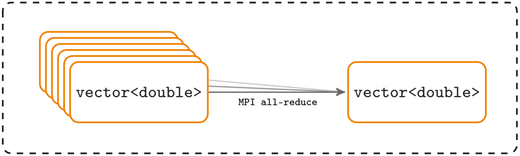
```
Histo1DGroupPtr _hist; //< Histo1DGroup = HistoGroup<double,double>
...
book(_hist, { 1.0, 2.0, 3.0, 4.0 });
for (auto& bin : hist->bins()) {
  book(bin, 1, 1, bin.index());
}
...
_hist->fill(val1, val2);
...
normalize(_hist); // or: normalizeGroup(_hist) if the grouped sumW is to be used
divByGroupWidth(_hist); // divide by bin width along group axis
```

- New interface to `HDF5` and `HighFive` for storing and loading analysis-specific auxiliary data

- New (optional) interface to ONNX Runtime as (current) best option for ML preservation

Better support for massively parallel applications

- YODA2 inheritance structure makes it straightforward to `serialize` object data
  - numerical content of `AnalysisHandler` can be translated into `std::vector<double>`
  - contiguous arrays of primitive types lend themselves much better to MPI communication
- memory block of data can be loaded back into an `AnalysisHandler` for `deserialize`-ing and `finalize`-ing



- also: reduced I/O load from parsing info files in the initialisation phase
- more profiling and optimisations envisaged for the future

Summary

- histograms are a powerful tool and often taken for granted
- 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 since before Christmas – check it out: [yoda.hepforge.org]
- Rivet 4.0.0 has been out since Feb 29 – check it out: [rivet.hepforge.org]
- plans for the future: performance optimisations, alternative YODA-`HDF5` output format, primary particle definition, flavour-sensitive k_{T} clustering . . . stay tuned!

Backup

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;
};
```

YODA2: 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

YODA2: 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

YODA2: 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

YODA2: 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

YODA2: Design principles II

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

YODA2: 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"

YODA2: 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

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