Histograms

March 19, 2017

Frequency distributions in ROOT are handled by a set of classes derived from the histogram class TH1, in our case TH1F. The letter F stands for "float", meaning that the data type float is used to store the entries in one histogram bin.

root [0] TF1 efunc("efunc","exp([0]+[1]*x)",0.,5.);

root [1] efunc.SetParameter(0,1);

root [2] efunc.SetParameter(1,-1);

root [3] TH1F h("h","example histogram",100,0.,5.);

root [4] for (int $i = 0; i < 1000; i++$) {h.Fill(efunc.GetRandom());}

root [5] h.Draw();

In line 3 a histogram is instantiated, with a name, a title, a certain number of bins (100 of them, equidistant, equally sized) in the range from 0 to 5. Here use pseudo-random numbers generated with the method TF1::GetRandom to fill this histogram with data, which in turn uses an instance of the ROOT class TRandom created when ROOT is started. Data is entered in the histogram at line 4 using the method TH1F::Fill in a loop construct. As a result, the histogram is filled with 1000 random numbers distributed according to

the defined function. The histogram is displayed using the method TH1F::Draw(). The class TH1F does not contain a convenient input format from plain text files. One number per line stored in the text file expo.dat is read in via an input stream and filled in the histogram until end of file is reached.

root [1] TH1F h("h","example histogram",100,0.,5.);

root [2] ifstream inp; double x;

root [3] inp.open("expo.dat");

root [4] while (inp ≫ x) { h.Fill(x); }

root [5] h.Draw();

root [6] inp.close();

ROOT supports histograms up to three dimensions. The histogram classes are split into further categories, depending on the set of possible bin values:

TH1C, TH2C and TH3C contain one byte per bin (maximum bin content = 255)

TH1S, TH2S and TH3S contain one short per bin (maximum bin content = 65 535)

TH1I, TH2I and TH3I contain one integer per bin (maximum bin content = 2 147 483 647)

TH1F, TH2F and TH3F contain one float per bin (maximum precision = 7 digits)

TH1D, TH2D and TH3D contain one double per bin (maximum precision = 14 digits)

ROOT also supports profile histograms. The inter-relation of two measured quantities X and Y can always be visualized with a two-dimensional histogram or scatter-plot. Profile histograms are used to display the mean value of Y and its RMS for each bin in X. If Y is an unknown but single-valued approximate function of X, it will have greater precision in a profile histogram than in a scatter plot.

TProfile : one dimensional profiles

TProfile2D : two dimensional profiles

All ROOT histogram classes are derived from the base class TH1.

0.1 Creating Histograms

Calling the Clone() method of an existing histogram

Making a projection from a 2-D or 3-D histogram

Reading a histogram from a file

// using various constructors

TH1* h1 = new TH1I("h1", "h1 title", 100, 0.0, 4.0);

TH2* h2 = new TH2F("h2", "h2 title", 40, 0.0, 2.0, 30, -1.5, 3.5);

TH3* h3 = new TH3D("h3", "h3 title", 80, 0.0, 1.0, 100, -2.0, 2.0, 50, 0.0, 3.0);

// cloning a histogram

TH1* hc = (TH1*)h1-¿Clone();

// projecting histograms

// the projections always contain double values !

TH1* hx = h2-¿ProjectionX(); // ! TH1D, not TH1F

TH1* hy = h2-¿ProjectionY(); // ! TH1D, not TH1F

0.1.1 Constant or Variable Bin Width

To construct a histogram, provide the name and title of histogram and for each dimension: the number of bins, the minimum x (lower edge of the first bin) and the maximum x (upper edge of the last bin).

TH2* h = new TH2D(

/* name */ “h2”,

/* title */ “Hist with constant bin width”,

/* X-dimension */ 100, 0.0, 4.0,

/* Y-dimension */ 200, -3.0, 1.5);

When employing this constructor, you will create a histogram with constant (fixed) bin width on each axis.

To create histograms with variable bin widths, pass an array (single or double precision) of bin edges. When the histogram has n bins, then there are $n + 1$ distinct edges, so the array you pass must be of size $n + 1$.

const Int_t NBINS = 5;

Double_t edges[NBINS + 1] = $\{0.0, 0.2, 0.3, 0.6, 0.8, 1.0\}$;

// Bin 1 corresponds to range [0.0, 0.2]

// Bin 2 corresponds to range [0.2, 0.3] etc...

TH1* h = new TH1D(

/* name */ “h1”, /* title */ “Hist with variable bin width”,

/* number of bins */ NBINS,

/* edge array */ edges

);

Each histogram object contains three TAxis objects: fXaxis , fYaxis, and fZaxis. The bin edges are always stored internally in double precision.

Examine the actual edges / limits of the histogram bins by accessing the axis parameters.

const Int_t XBINS = 5; const Int_t YBINS = 5;

Double_t xEdges[XBINS + 1] = {0.0, 0.2, 0.3, 0.6, 0.8, 1.0};

Double_t yEdges[YBINS + 1] = {−1.0, −0.4, −0.2, 0.5, 0.7, 1.0};

TH2* h = new TH2D("h2", "h2", XBINS, xEdges, YBINS, yEdges);

TAxis* xAxis = h-¿GetXaxis(); TAxis* yAxis = h-¿GetYaxis(); cout << "Third bin on Y-dimension: " << endl; // corresponds to [-0.2, 0.5]

cout << "\tLower edge: " << yAxis-¿GetBinLowEdge(3) << endl;

cout << "\tCenter: " << yAxis-¿GetBinCenter(3) << endl;

cout << "\tUpper edge: " << yAxis-¿GetBinUpEdge(3) << endl;

0.2 Bin Numbering

All histogram types support fixed or variable bin sizes. 2-D histograms may have fixed size bins along X and variable size bins along Y or vice-versa.

For all histogram types: nbins , xlow , xup

Bin#0 contains the underflow. Bin#1 contains the first bin with low-edge (xlow INCLUDED). The second to last bin (bin# nbins) contains the upper-edge (xup EXCLUDED). The Last bin (bin# nbins+1) contains the overflow. In case of 2-D or 3-D histograms, a "global bin number is defined. For example, assuming a 3-D histogram h with binx, biny, binz, the function returns a global/linear bin number. This global bin is useful to access the bin information independently of the dimension.

A histogram can be re-binned via the TH1::Rebin() method. It returns a new histogram with the re-binned contents. If bin errors were stored, they are recomputed during the re-binning.

0.3 Filling Histograms

h1-¿Fill(x);

h1-¿Fill(x,w); // with weight

h2-¿Fill(x,y);

h2-¿Fill(x,y,w);

h3-¿Fill(x,y,z);

h3-¿Fill(x,y,z,w);

The Fill method computes the bin number corresponding to the given x, y or z argument and increments this bin by the given weight. The Fill() method returns the bin number for 1-D histograms or global bin number for 2-D and 3-D histograms. If TH1::Sumw2() has been called before filling, the sum of squares is also stored. One can increment a bin number directly by calling TH1::AddBinContent(), replace the existing content via TH1::SetBinContent(), and access the bin content of a given bin via TH1::GetBinContent().

Double_t binContent = h-¿GetBinContent(bin);

By default, the number of bins is computed using the range of the axis. You can change this to re-bin automatically by setting the automatic re-binning option:

h-¿SetBit(TH1::kCanRebin);

Once this is set, the Fill() method will automatically extend the axis range to accommodate the new value specified in the Fill() argument. The used method is to double the bin size until the new value fits in the range, merging bins two by two. The TTree::Draw() method extensively uses this automatic binning option when drawing histograms of vari-

ables in TTree with an unknown range. The automatic binning option is supported for 1-D, 2-D and 3-D histograms. During filling, some statistics parameters are incremented to compute the mean value and root mean square with the maximum precision. In case of histograms of type TH1C, TH1S, TH2C, TH2S, TH3C, TH3S a check is made that the bin contents do not exceed the maximum positive capacity (127 or 65 535). Histograms of all types may have positive or/and negative bin contents.

0.4 Random Numbers and Histograms

TH1::FillRandom() can be used to randomly fill a histogram using the contents of an existing TF1 function or another TH1 histogram (for all dimensions).

root[] TH1F h1("h1", "Histo from a Gaussian", 100, -3, 3);

root[] h1.FillRandom("gaus", 10000);

TH1::GetRandom() can be used to get a random number distributed according the contents of a histogram. To fill a histogram following the distribution in an existing histogram you can use the second signature of TH1::FillRandom(). Next code snipped assumes that h1 is an existing histogram (TH1).

root[] TH1F h2("h2", "Histo from existing histo", 100, -3, 3);

root[] h2.FillRandom(&h1, 1000);

The distribution contained in the histogram h1 (TH1) is integrated over the channel contents. It is normalized to one. The second parameter (1000) indicates how many random numbers are generated.

Getting 1 random number implies:

Generating a random number between 0 and 1 (say r1)

Find the bin in the normalized integral for r1

Fill histogram channel

0.5 Adding, Dividing, and Multiplying

Histograms objects (not pointers) TH1F h1 can be multiplied by a constant using:

h1.Scale(const)

A new histogram can be created without changing the original one by doing:

TH1F h3 = 8*h1;

To multiply two histogram objects and put the result in a 3rd one do:

TH1F h3 = h1*h2;

The same operations can be done with histogram pointers TH1F *h1, *h2 following way:

h1→Scale(const) TH1F h3 = 8*(*h1); TH1F h3 = (*h1)*(*h2);

If a histogram has associated error bars (TH1::Sumw2() has been called), the resulting error bars are also computed assuming independent histograms. In case of divisions, binomial errors are also supported.

0.6 Projections

0.7 Drawing Histograms

0.8 Making a Copy of an Histogram

0.9 Normalizing Histograms

0.10 Saving/Reading Histograms to/from a File

0.11 Miscellaneous Operations

0.11.1 χ^2 test

Double_t TH1::Chi2Test(const TH1* h2, Option_t* option = "UU", Double_t* res = 0) const

χ^2 test for comparing weighted and unweighted histograms

Function: Returns p-value. Other return values are specified by the 3rd parameter

Parameters h2 the second histogram

option

"UU" = experiment experiment comparison (unweighted-unweighted)

"UW" = experiment MC comparison (unweighted-weighted). Note that the first histogram should be unweighted

"WW" = MC MC comparison (weighted-weighted)

"NORM" = to be used when one or both of the histograms is scaled but the histogram originally was unweighted by default underflows and overflows are not included:

"OF" = overflows included

"UF" = underflows included

"P" = print chi2, ndf, p_value, igood

"CHI2" = returns chi2 instead of p-value

"CHI2/NDF" = returns χ^2/ndf

res not empty - computes normalized residuals and returns them in this array

Comparison of two histograms expect hypotheses that two histograms represent identical distributions. To make a decision p-value should be calculated. The hypotheses of identity is rejected if the p-value is lower then some significance level. Traditionally significance levels 0.1, 0.05 and 0.01 are used. The comparison procedure should include an analysis of the residuals which is often helpful in identifying the bins of histograms responsible for a significant overall χ^2 value. Residuals are the difference between bin contents and expected bin contents. Most convenient for analysis are the normalized residuals. If hypotheses of identity are valid, normalized residuals are approximately independent and identically distributed random variables having $N(0,1)$ distribution. Analysis of residuals expect test of above mentioned properties of residuals. Notice that indirectly the analysis of residuals increase the power of χ^2 test.

Consider two histograms with the same binning and the number of bins equal to r. Let us denote the number of events in the i-th bin in the first histogram as n_i and as m_i in the second one. The total number of events in the first and second histogram are respectively

equal to:

$$N = \sum_{i=1}^{r} n_i \ ,$$
$$M = \sum_{i=1}^{r} m_i \ .$$

The hypothesis of identity (homogeneity) is that the two histograms represent random values with identical distributions. It is equivalent that there exist r constants $p_1, \cdots, p_r$, such that

$$\sum_{i=1}^{r} p_i = 1 \ . \tag{1}$$

The probability of belonging to the i-th bin for some measured value in both experiments is equal to p_i. The number of events in the i-th bin is a random variable with a distribution approximated by a Poisson probability distribution

$$\frac{e^{-Np_i}(Np_i)^{n_i}}{n_i!} \tag{2}$$

for the first histogram and with distribution

$$\frac{e^{-Mp_i}(Mp_i)^{m_i}}{m_i!} \tag{3}$$

for the second histogram. If the hypothesis of homogeneity is valid, then the maximum likelihood estimator of p_i, $i = 1, \cdots, r$, is

$$\hat{p}_i = \frac{n_i + m_i}{N + M} \tag{4}$$

and then

$$\chi^2 = \sum_{i=1}^{r} \frac{(n_i - N\hat{p}_i)^2}{N\hat{p}_i} + \sum_{i=1}^{r} \frac{(m_i - M\hat{p}_i)^2}{M\hat{p}_i} = \frac{1}{MN} \sum_{i=1}^{r} \frac{(Mn_i - Nm_i)^2}{n_i + m_i} \tag{5}$$

has approximately a $\chi^2_{(r1)}$ distribution. The comparison procedure can include an analysis of the residuals which is often helpful in identifying the bins of histograms responsible for

a significant overall χ^2 value. Most convenient for analysis are the adjusted (normalized) residuals

$$r_i = \frac{n_i - N\hat{p}_i}{\sqrt{N\hat{p}_i}\sqrt{(1 - N/(N+M))(1 - (n_i + m_i)/(N+M))}} \tag{6}$$

If hypotheses of homogeneity are valid then residuals r_i are approximately independent and identically distributed random variables having $N(0,1)$ distribution. The application of the χ^2 test has restrictions related to the value of the expected frequencies Np_i, Mp_i, $i = 1, \cdots, r$. A conservative rule formulated is that all the expectations must be 1 or greater for both histograms. In practical cases when expected frequencies are not known the estimated expected frequencies $M\hat{p}_i$, $N\hat{p}_i$, $i = 1, \cdots, r$ can be used.

0.11.2 KolmogorovSmirnov test

Double_t TMath::KolmogorovTest(Int_t na, const Double_t* a, Int_t nb, const Double_t* b, Option_t* option)

Statistical test whether two one-dimensional sets of points are compatible with coming from the same parent distribution, using the Kolmogorov test. That is, it is used to compare two experimental distributions of unbinned data.

Input: *a*,*b*: One-dimensional arrays of length n_a, n_b, respectively. The elements of *a* and *b* must be given in ascending order. option is a character string to specify options "D" Put out a line of "Debug" printout "M" Return the Maximum Kolmogorov distance instead of prob

Output: The returned value prob is a calculated confidence level which gives a statistical test for compatibility of *a* and *b*. Values of prob close to zero are taken as indicating a small probability of compatibility. For two point sets drawn randomly from the same

parent distribution, the value of prob should be uniformly distributed between zero and one. in case of error the function return -1 If the 2 sets have a different number of points, the minimum of the two sets is used.

Method: The Kolmogorov test is used. The test statistic is the maximum deviation between the two integrated distribution functions, multiplied by the normalizing factor rdmax $\times \sqrt{\frac{n_a \times n_b}{(n_a + n_b)}}$.

The nuts-and-bolts of the TMath::KolmogorovTest() algorithm is a for-loop over the two sorted arrays a and b representing empirical distribution functions. The for-loop handles 3 cases: when the next points to be evaluated satisfy $a > b$, $a < b$, or $a = b$:

for(Int_t i=0;i¡na+nb;i++) { if (a[ia-1] ¡ b[ib-1]) { rdiff -= sa; ia++; if (ia ¿ na) {ok = kTRUE; break;} }else if (a[ia-1] ¿ b[ib-1]) { rdiff += sb; ib++; if (ib ¿ nb) {ok = kTRUE; break;} } else{ rdiff += sb - sa; ia++; ib++; if (ia ¿ na){ok = kTRUE; break;} if (ib ¿ nb) {ok = kTRUE; break;} } rdmax = TMath::Max(rdmax,TMath::Abs(rdiff)); }

For the last case, $a = b$, the algorithm advances each array by one index in an attempt to move through the equality. However, this is incorrect when one or the other of a or b (or both) have a repeated value, call it x. For the KS statistic to be computed properly, rdiff needs to be calculated after all of the a and b at x have been tallied (this is due to the definition of the empirical distribution function; another way to convince yourself that the old CERNLIB method is wrong is that it implies that the function defined as the difference between a and b is multi-valued at x besides being ugly, this would invalidate Kolmogorov's theorem).

0.12 Alphanumeric Bin Labels

0.13 Histogram Stacks

0.14 TH2Poly

0.15 Profile Histograms

0.16 Iso Surfaces

0.17 3D Implicit Functions

0.18 TPie

0.19 The User Interface for Histograms