Documentation: Statistical Analysis software

Kate Pachal

January 4, 2013

1 Introduction

2 Summary of Classes

Below is a summary of all classes and functions currently available. They are ordered according to ease of explanation.

2.1 MjjHistogram

A simple class to store related formats of an input histogram, along with some of its properties. The MjjHistogram can be initialized using either a basic data TH1F straight from the data processing or a TH1F normalized by bin width and luminosity, but the form of the input must be specified. The goal of the design is to pass MjjHistograms instead of TH1F's to all user-accessible functions such that the correct form, whether basic or normalized, can be accessed by the program without the user needing to know which is required in a given situation.

MjjHistogram(TH1F * inputHisto, string option, double thislumi)

Constructor. `option` must be "basic" or "normalized". If initialized using a basic TH1F, the normalized one is immediately created, and vice versa, so the MjjHistogram will always have both "basic and "normalized" histograms initialized. Third input is luminosity; this is stored as a property of the MjjHistogram and used for all normalization and unnormalization.

∼MjjHistogram()

TH1D * getHistogram()

Returns basic histogram (event numbers, not normalized).

TH1D * getNormalizedHistogram()

Returns histogram normalized by bin width and luminosity.

TH1D * getTrimmedHistogram()

Returns a trimmed version of the basic histogram. This trimmed histogram is NOT created automatically and will only exist if trimHistoOutsideBinRange(int firstBin, int lastBin) has already been called.

int getFirstBinWithData()

Returns bin number of first bin containing data in basic and normalized histograms. This value is unchanged by trimming the histogram.

int getLastBinWithData()

Similar to above. Returns bin number of last bin with data in basic and normalized histograms.

double getLumi()

Returns value of luminosity stored by this MjjHistogram. This value cannot be changed after it is set in initialization, but can be read out at any time.

trimHistoOutsideBinRange(int firstBin, int lastBin)

Trims basic histogram such that all bins outside of the range [`firstBin, lastBin`] are empty. Note that both bins passed as parameters are contained in the range and will still contain data in the trimmed form.

2.2 MjjStatisticalTest

A base class from which individual statistical tests derive. It contains only a constructor, a destructor, and the virtual function doTest, whose specifics are determined by the statistical test in question. The user should not have to use this at any point but should rather use the classes corresponding to specific tests (see below).

2.3 MjjLogLikelihoodTest

A class derived from MjjStatisticalTest and used to calculate the negative log likelihood value in a comparison between two histograms.

MjjLogLikelihoodTest()

Constructor. No parameters required.

~MjjLogLikelihoodTest

double doTest(MjjHistogram & h_data, MjjHistogram & h_background, int firstBinToUse=-1, int lastBinToUse=-1)

Returns the negative log likelihood value of a comparison between h_data and h_background. Take the product of the TMath::PoissonI value comparing data to background in a bin for each bin between firstBintoUse and lastBinToUse inclusive. In the case that firstBinToUse, lastBinToUse are not specified, they are interpreted to be the first and last bins containing data. The returned value is the negative logarithm of this product.

2.4 MjjChi2Test

A class derived from MjjStatisticalTest and used to calculate the χ^2 value in a comparison between two histograms.

MjjChi2Test()

Constructor. No parameters required.

~MjjChi2Test

double doTest(MjjHistogram & h_data, MjjHistogram & h_background, int firstBinToUse=-1, int lastBinToUse=-1)

Returns the χ^2 value of a comparison between h_data and h_background. Where d_i is the content of bin i in h_data, b_i is the content of bin i in h_background, and δb is the error on bin i in h_background, this function returns a value given by

$$\sum_{i=\text{firstBinToUse}}^{\text{lastBinToUse}} \frac{(d-b)^2}{b+(\delta b)^2}$$

Note that this value depends on the method used to set the bin errors in h_background. This has not yet been adjusted to handle differences between MC and data. It is currently using error bar values appropriate to data.

2.5 MjjFitFunction

This at present feels like a clumsily written class. An effort will shortly be made to improve the inner workings; however, it is probable that none of the functions accessible to the user will change.

An MjjFitFunction defines the TF1 that will be used to fit a histogram, along with a number of methods to simplify control. Two different functions are currently implemented. First, there is a basic four-parameter function of the form generally used by the dijets group:

$$f_{\text{fourParam}}(m) = p_0(1-x)^{p_1}x^{p_2+p_3\ln x}$$

Second, there is a five-parameter function equivalent to the one above but with an added term that incorporates a sample signal MC file. A signal template function in the form of a TH1F is used to set the value of the function such that at each bin, the value returned by the function is:

$$f_{\text{fiveParam}}(m) = f_{\text{fourParam}}(m) + p_4 * \frac{\text{n}}{\text{w}}$$

where n is the content of the bin corresponding to mass m in the signal template and w is the width of that bin. Thus this function fits a distribution as though expecting a signal of the same shape and location as the signal template, while varying the normalization.

Each MjjFitFunction contains one, and only one, initialized TF1*. Whether this is a simple four-parameter function or one which incorporates signal is specified at initialization and cannot later be changed. The class is designed such that the user should be able to set and get parameter and range values without ever needing to directly access the function. However, should it be necessary, calling getFitFunction() will return the TF1*.

MjjFitFunction(const double & mjjLow, const double & mjjHigh, string option)

Constructor. `mjjLow` and `mjjHigh` specify the range which the function will fit. These cannot be left blank. `option` must be ”fourparam” or ”fourparamwithsignal” and determines the functional form to be used.

~MjjFitFunction()

TF1 * getFitFunction()

It should not be neccessary to use this function, but it can be called in case the stored TF1 * needs to be accessed directly.

void setParameterDefaults(double p0, double p1=1, double p2=1, double p3=1, double p4=1, double p5=1);

This saves the input parameter values as defaults of the MjjFitFunction. If the specified option is ”fourparam” the first four parameters will be saved regardless of the number actually passed – if fewer than four, the others will be set to 1; if greater than 4, the others will be ignored. If the specified option is ”fourparamwithsignal” the first five parameters will similarly be used. Therefore the user must be careful to remember the nature of the function being handled and use this method appropriately.

void restoreParameterDefaults()

Resets all function parameters to the stored default values. Note that if setParameterDefaults has not yet been called, these are all equal to 1.

void getCurrentParameterValues(double& p0, double& p1, double& p2, double& p3)

void getCurrentParameterValues(double& p0, double& p1, double& p2, double& p3, double& p4)

This overloaded function takes pointers to four or five doubles and sets their values equal to the current function parameters. If the user passes the wrong number of doubles for the function, a helpfully reminding error message will be sent.

void setCurrentParameterValues(double p0, double p1=1, double p2=1, double p3=1, double p4=1, double p5=1)

Sets parameters to the passed values. If the specified option is ”fourparam” the first four passed values will be used; if fewer than four, the others will be set to 1 and if greater than 4, the others will be ignored. If the specified option is ”fourparamwithsignal” the first five parameters will similarly be used.

void setSignalTemplate(MjjHistogram & inputTemplate)

The input MjjHistogram will be treated as a sample signal and used to specify the shape of a ”fourparamwithsignal” function according to the equation given earlier. An error message will be returned if this method is called with a function that does not allow signal.

void doWindowExclusion(bool yesOrNo)

Work in progress.

double getMinXVal()

Returns low end of function range specified in constructor.

double getMaxXVal()

Returns high end of function range specified in constructor.

void MakeHistFromFunction(TH1F * hist, double xmin=-1, double xmax=-1, std::string option=”atCenter”)

This fills the passed histogram based on the value of the MjjFitFunction in each bin. If `xmin` and `xmax` are specified, hist is filled between (and including) the bins corresponding to these endpoints. If no endpoints are specified, hist is filled over its entire range. `option` determines how the desired bin content is determined from the function. The default option, `"atCenter"`, means the bin content is set to the value of the MjjFitFunction at the center of the bin. If `option=="average"`, bin content is set to the integral of the function between the bin endpoints divided by width of the bin. If `option=="integral"`, bin content is just set to the integral of the function between the bin endpoints. In all cases, the error on each bin is set to the square root of the bin content.

2.6 MjjFitter

This class fits a given function to a given histogram. Instead of a TH1F, the fitter requires an MjjHistogram. This simplifies matters for the user by allowing the program to take care of necessary normalizations. At the moment, the user is not given the option of tinkering with the fit mechanisms without altering the class. Ideally, the main program should be able to simply call Fit() on any function and trust the MjjFitter to succeed.

MjjFitter(double minFit=0, double maxFit=0, int printL=5)

Constructor. The parameters `minFit` and `maxFit` specify the x values between which to fit. This is something which will be cleaned up in future: bin numbers rather than x values should be passed. The last parameter corresponds to the MINUIT print level. 0 is standard, 3 or 5 will increase level of detail in print statements, and -1 suppresses all but error statements.

∼MjjFitter()

bool Fit(MjjHistogram & mjjHistogram, MjjFitFunction & mjjFitFunction, const TString & fitOption = "ROOT_Chi2")

This method receives as input the MjjHistogram to fit, the MjjFitFunction to use, and an option specifying what type of fit to perform. The default option is "ROOT_Chi2" corresponding to a basic ROOT fit without using MINUIT: this is the fastest option. The most detailed and desireable option is "Minuit_ML," corresponding to a minimum log likelihood fit performed via a direct call to MINUIT2. A third option, "GSL_ML", exists in the code but has not been properly implemented yet. Fit() returns a value of true if the fit converged and false if it failed.

Fit() with "Minuit_ML" begins by performing a basic ROOT fit to get good start values for the fit parameters. A ROOT Minuit2Minimizer is then created using the passed function. Parameters of the minimizer are set by the function rather than by the user: as long as it is being used to fit a dijet mass spectrum, it is possible to fix these to an optimal value and they are then protected. Parameters of the Minuit2Minimizer are initialized using the start values determined by the basic ROOT fit and are not bounded. See the Minuit2 manual for an explanation of the advantages of using parameters without limits.

The minimization is then performed. If it fails (Minuit2Minimizer-¿Status() ¿ 1), up to five retries are allowed. For each retry the starting parameter values are varied slightly away from those initially found by the basic ROOT fit. If any one of the retries succeeds, these parameters are kept and the fit returns true. If all five fail, the fit returns false and stops. In the case of a successful fit and if the MjjFitter has a printLevel> -1, final values of the parameters as well as χ^2 and minimum log likelihood tests are printed out for the user.

2.7 MjjSignificanceTests

This class has three constituent methods for determining the significance of the difference between two histograms. All three methods return a TH1F * whose bins have the same spacing and structure as the input histograms. Each bin of the output represents the significance of the difference between the corresponding bin in the two inputs. For each of the methods below, let d be the content of the relevant bin in h_data, δd be the error on that bin, b be the content of the relevant bin in h_background, and δb be the error on that bin.

MjjSignificanceTests()

Constructor. This takes no parameters. The histograms to be tested are passed to the individual methods.

∼MjjSignificanceTests()

TH1F * findRelativeDifference(MjjHistogram & h_data, MjjHistogram & h_background)

If $b == 0$ then bin content and bin error are both set to zero. Otherwise, content of each bin is set to

$$(d - b)/b$$

The bin error is set to

$$|\frac{\delta d}{b}|$$

TH1F * findSignificanceOfDifference(MjjHistogram & h_data, MjjHistogram & h_background)

For each bin with nonzero content in h_data, bin content is set to

$$\frac{(d-b)}{\sqrt{(\delta b)^2 + (\delta d)^2}}$$

Bin errors are set to zero.

TH1F * findResidual(MjjHistogram & h_data, MjjHistogram & h_background)

This gives the familiar histogram which the Dijets group uses in the bottom of the m_{jj} plot. For each bin with nonzero content in h_data, a probability value given by pVal = PoissonPval(d, b, "twoSides") is calculated (see probabilityFunctions below). This probability is converted into a form expressed in number of standard deviations using probToSigma(pVal) (again, see probabilityFunctions). Because of the functions used for these calculations, very insignificant differences are represented by negative numbers while more significant differences are positive numbers. However, this is not a very useful way to view the plot: we want the sign of the bin content in the residual histogram to represent whether the data falls above or below the background in a given bin. Therefore all bins with negative (insignificant) values are set to zero and bins with positive (significant) values are set to that magnitude but in a positive direction if $b > d$ and a negative direction otherwise. All bin errors are set to zero.

2.8 MjjStatisticsBundle

This is simply a struct containing the objects:
TH1F * statisticsFromPseudoexperimentsHist;
double originalStatistic;
vector <double> statisticsFromPseudoexperiments;

It is a convenient way to summarize all the information calculated by the MjjPseudoExperimenter regarding the relationship of a histogram to a prediction and is the object returned by the two main functions of that class. The contents of the MjjStatisticsBundle can just be accessed directly by their names above. For an example, see util/testSearchWithSystematics.cxx.

2.9 MjjPseudoExperimenter

Runs a user-specified number of pseudoexperiments to determine the probability of finding a m_{jj} histogram given a prediction in the form of either another histogram or a function. The results are returned as an MjjStatisticsBundle containing the statistical measure of the original histogram to the prediction, the set of these statistical measures comparing every pseudoexperiment to the prediction in the form of a vector, and a TH1F * made by plotting these individual values. The user will likely rarely be interested in the individual values, so the pseudoexperiment histogram and the original statistic will be the most valuable information. The vector is largely included in case specific values are required by other functions for further tests.

MjjPseudoExperimenter()

Constructor. No parameters. Everything is specified in the call to the method.

~MjjPseudoExperimenter()

MjjStatisticsBundle getPseudoExperimentStatsOnHistogram(MjjHistogram & templateHist, MjjHistogram & observedHist, int firstBinToUse=1, int lastBinToUse=1e3, const TString & option="logLikelihood", int nExperiments=1000)

It is important that the user passes the histograms in the correct order: the first will be taken as the template, so this should be the "expected" values. The second is treated as the observation. If firstBinToUse and lastBinToUse are not specified then the entire range of the histogram is used. The TString "option" can be "logLikelihood" or "Chi2." Finally, the last int is the number of pseudoexperiments to be performed.

The statistical test specified by "option" is performed on the two input histograms using an MjjChi2Test or an MjjLogLikelihoodTest and the value returned is stored in the MjjStatisticsBundle as "originalStatistic." Then the requested number of pseudoexperiments is performed. For each pseudoexperiment, a new histogram is created by independently Poisson fluctuating the contents of every bin in observedHist. The same statistical test is now performed to compare the pseudoexperiment histogram to the templateHist and the calculated value is stored in a vector (saved to the MjjStatisticsBundle as statisticsFromPseudoexperiments). Once the pseudoexperiments are finished, the vector of statistics is used to make a histogram. This histogram has reasonably fine binning – nBins is set to 1/10 of the number of pseudoexperiments – so it can be rebinned conveniently by a plotter later.

MjjStatisticsBundle getPseudoExperimentStatsOnFunction(MjjFitFunction & functionToFit, MjjHistogram & h_background, int firstBinToUse=1, int lastBinToUse=1e3, const TString & option="logLikelihood", int nExperiments=1000)

Here the input is a little simpler for the user. An observed histogram and the function that has been fit to it are passed as parameters. The originalStatistic is calculated from the input observed histogram and a histogram generated from the function. For each pseudoexperiment, a new histogram is created by fluctuating the input h_background bin-by-bin as before. This time, however, the function is used to fit the resulting histogram. A new histogram is generated from the fitted function, the statistical test is performed on the pseudoexperiment histogram and the histogram from its fit, and that value is stored in the vector of results. The pseudoexperiment histogram is generated from this vector just as described in the previous method.

A few points to note: If the fit to a pseudoexperiment histogram fails, that pseudoexperiment is discarded and is not counted towards to total number to be performed. The parameters of the MjjFitFunction are reset to their default values before fitting to replicate the fit to the real observed data as nearly as possible.

Note than when using this method, the user should decrease the number of pseudoexperiments if possible. Each pseudoexperiment is much slower when a fit needs to be performed.

2.10 MjjBumpHunter

This is a very simple bump hunter. A lot of the functionality of Giorgios's bump hunter was removed because it was never actually being used, so this is the trimmed-down version that does only what we actually need. Unless otherwise specified, bumps have a minimum width of two bins, a maximum width of half the total number of bins, and in the current implementation can be either excesses or deficits.

2.10.1 MjjBumpHunter(double gridStep)

Constructor. The parameter is unfortunate and hopefully a workaround can be found. It is used to specify the accuracy with which the probability of a certain bump is calculated. Smaller values increase accuracy but take a little longer. In the original code this seems to be set to 0.1, so that value is used in the examples here as well.

2.10.2 ~MjjBumpHunter()

void SetBumpHunterWidthLimits(int minWidth, int maxWidth)

This lets the user control the minimum and maximum bump widths in number of bins. Note that if minWidth is set to less than one it will be reset to one by the program.

double getBumpHunterPval(MjjHistogram & dataHist, MjjHistogram & bkgHist)

This method actually runs the bump hunter and returns the negative log of the most unusual pval found. It stores the edges of the bump that gave this pval; they can be accessed with the following two methods.

double getLowEdgeOfBump()

Returns the x value of the low edge of the biggest bump (corresponding to the pval from the above function). If this is called before running the bump hunter it will just return 0.

double getHighEdgeOfBump()

Returns the x value of the high edge of the biggest bump. If this is called before running the bump hunter it will just return 0.

2.11 probabilityFunctions

This is not a class but a set of functions for doing basic and recurring probability calculations. A brief description of the available functions is below.

double probToSigma(double prob)

Converts a probability to its equivalent in numbers of standard deviations. This is not as straightforward as it sounds: it is used to calculate the residual and is the function that returns negative values for very probable outcomes and positive values for more significant deviations from this expectation. Mathematically, for an input probability $0 < P < 1$ it returns

$$\sqrt{2}\ \text{TMath::ErfInverse}(1 - 2 * P)$$

double sigmaToProb(double sigma)

Reverses the calculation of the function above. For an input σ, returns

$$\frac{1}{2}(1 - \text{TMath::Erf}(\frac{\sigma}{\sqrt{2.0}}))$$

double PoissonPval(const double& d, const double& b, std::string option="twoSides")

Returns the integral of the poisson probability to measure d ¿= observed d, given we expect b. The input parameter d is taken to be the observed value. Three options for the final parameter were available in the original code, but there is no obvious reason to use anything other than "twoSides" at this point. Changing the option affects how the function differentiates between cases where $d > b$ and those where $b > d$.

double integrateGaussian(const double& mean, const double& sigma, const double& x1, const double& x2)

Returns the integral under a Gaussian of the input mean and standard deviation between the boundaries of x1 and x2.

double integrateNormalDistribution(const double& x1, const double& x2)

Returns the integral under the normal distribution between x1 and x2; that is, integrateGaussian(0,1,x1,x2).

3 Usage Examples

Two programs making use of this structure can be found in StatisticalAnalysis2012/util/. SearchPhaseNoSystematics demonstrates the use of the Fitter and FitFunction classes, creation and use of an MjjHistogram, the MjjStatisticalTests, and the MjjSignificanceTests. The other, testSearchWithSystematics, uses every class currently available including the BumpHunter and the PseudoExperimenter. However, unlike SearchPhaseNoSystematics, this program is constantly being modified as I test new things and so may not be the most instructional at any given time.

Note that neither code creates any actual plots. It is recommended that the user save all output histograms to a root file and then have a separate and easily modified plotter somewhere else. This is very different from the way things were done in Giorgios's code, and it could be very tempting to slip back into making this a plotter as well, just because we have immediate and easy access here to a number of useful quantities such as the limits of a bump found by bumpHunter or the output χ^2 value of a fit. Instead I am currently saving such values to a root file as well. What I'm not yet sure of is the best way to do this. Perhaps it would be worth making a TH1F with a single entry to store such a number. At the moment, as demonstrated in the code, I am using TVectorD's to store doubles as they can be written to root files.

An example of reading these values back out of a root file and using them in a couple of basic plots can be found in StatisticalAnalysis2012/plotting/makeMjjPlots.C. This macro currently produces the legendary figure 1 with and without bump hunter limits, as well as an (ugly) yellow pseudoexperiment plot (negative log likelihood in this case) with an arrow to the value of the statistic in data.

4 Ongoing Issues

4.1 How to handle an intially failed fit

4.2 Difference in statistical treament of data, MC

4.3 Implementation of BKG_CONV_* flags

5 Future Plans

5.1 Bayesian likelihoods

5.2 Generalized output format; shared plotter?

5.3 Thorough validation