5 Fitting Histograms

To fit a histogram you can use the Fit Panel on a visible histogram using the GUI, or you can use the \texttt{TH1:Fit} method. The Fit Panel, which is limited, is best for prototyping. The histogram needs to be drawn in a pad before the Fit Panel is available. The \texttt{TH1:Fit} method is more powerful and used in scripts and programs.

The Fit Panel

To display the Fit Panel right click on a histogram to bring up the context menu, and then select the menu option: FitPanel.

The first sets of buttons are the predefined functions of ROOT that can be used to fit the histogram. You have a choice of several polynomials, a Gaussian, a landau, and an exponential function. You can also define a function and call it "user". It will be linked to the user button on this panel.

You have the option to specify Quiet or Verbose. This is the amount of feedback printed on the root command line on the result of the fit. When a fit is executed the image of the function is drawn on the current pad. By default the image of the histogram is replaced with the image of the function. Select Same Picture to see the function drawn and the histogram on the same picture.

Select \texttt{W: Set all weights to 1} to set all errors to 1.
Select \texttt{E: Compute best errors} to use the Minos technique to compute best errors.

When fitting a histogram, the function is attached to the histogram's list of functions. By default the previously fitted function is deleted and replaced with the most recent one, so the list only contains one function. You can select \texttt{+ : Add to list of functions} to add the newly fitted function to the existing list of functions for the histogram. Note that the fitted functions are saved with the histogram when it is written to a ROOT file. By default, the function is drawn on the pad displaying the histogram.

Select \texttt{N: Do not store/draw function} to avoid adding the function to the histogram and to avoid drawing it.

Select \texttt{0: Do not draw function} to avoid drawing the result of the fit.

Select \texttt{L: Log Likelihood} to use log likelihood method (default is \texttt{chisquare} method).

The slider at the bottom of the panel allows you to set a range for the fit. Drag the edges of the slider towards the center to narrow the range. Draw the entire range to change the beginning and end. To returns to the original setting, you need to press Defaults. To apply the fit, press the \texttt{Fit} button.
The Fit Method

To fit a histogram programmatically, you can use the `TH1::Fit` method. Here is the signature of `TH1::Fit` and an explanation of the parameters:

```c
void Fit(const char *fname ,Option_t *option,
         Option_t *goption,Axis_t xxmin,Axis_t xxmax)
```

*fname*: The name of the fitted function (the model) is passed as the first parameter. This name may be one of ROOT pre-defined function names or a user-defined function.

The following functions are predefined, and can be used with the `TH1::Fit` method.

- **gaus**: A Gaussian function with 3 parameters:
  \[ f(x) = p0*\exp(-0.5*((x-p1)/p2)^2) \]

- **expo**: An exponential with 2 parameters:
  \[ f(x) = \exp(p0+p1*x) \]

- **polN**: A polynomial of degree N:
  \[ f(x) = p0 + p1*x + p2*x^2 +... \]

- **landau**: A Landau function with mean and sigma. This function has been adapted from the CERNLIB routine G110 denlan.

*option*: The second parameter is the fitting option. Here is the list of fitting options:

- "W" Set all errors to 1
- "I" Use integral of function in bin instead of value at bin center
- "L" Use log likelihood method (default is chi-square method)
- "U" Use a user specified fitting algorithm
- "Q" Quiet mode (minimum printing)
- "V" Verbose mode (default is between Q and V)
- "E" Perform better errors estimation using Minos technique
- "M" Improve fit results
- "R" Use the range specified in the function range
- "N" Do not store the graphics function, do not draw
- "O" Do not plot the result of the fit. By default the fitted function is drawn unless the option "N" above is specified.
- "+" Add this new fitted function to the list of fitted functions (by default, the previous function is deleted and only the last one is kept)
- "B" Disable the automatic computation of the initial parameter values for the standard functions like poln, expo, and gaus.
- "LL" An improved Log Likelihood fit in case of very low statistics and when bin contents are not integers. Do not use this option if bin contents are large (greater than 100).

*goption*: Third parameter is the graphics option (goption), it is the same as in the `TH1::Draw` (see Draw Options above).

*xxmin, xxmax*: Fourth and fifth parameters specify the range over which to apply the fit. By default, the fitting function object is added to the histogram and is drawn in the current pad.

Fit with a Predefined Function

To fit a histogram with a predefined function, simply pass the name of the function in the first parameter of `TH1::Fit`. For example, this line fits histogram object `hist` with a Gaussian.

```c
root[] hist.Fit("gaus");
```

For pre-defined functions, there is no need to set initial values for the parameters, it is done automatically.
Fit with a User-Defined Function

You can create a TF1 object and use it in the call the TH1::Fit. The parameter in to the Fit method is the NAME of the TF1 object.

There are three ways to create a TF1.
1. Using C++ like expression using x with a fixed set of operators and functions defined in TFormula.
2. Same as #1, with parameters
3. Using a function that you have defined

Creating a TF1 with a Formula

Let's look at the first case. Here we call the TF1 constructor by giving it the formula: \( \sin(x)/x \).

```
root[] TF1 *f1 = new TF1("f1","\sin(x)/x",0,10)
```

You can also use a TF1 object in the constructor of another TF1.

```
root[] TF1 *f2 = new TF1("f2","f1*2",0,10)
```

Creating a TF1 with Parameters

The second way to construct a TF1 is to add parameters to the expression. For example, this TF1 has 2 parameters:

```
root[] TF1 *f1 = new TF1("f1","[0]*x*sin([1]*x)",-3,3);
```

The parameter index is enclosed in square brackets. To set the initial parameters explicitly you can use the SetParameter method.

```
root[] f1->SetParameter(0,10);
```

This sets parameter 0 to 10. You can also use SetParameters to set multiple parameters at once.

```
root[] f1->SetParameters(10,5);
```

This sets parameter 0 to 10 and parameter 1 to 5. We can now draw the TF1:

```
root[] f1->Draw()
```
Creating a TF1 with a User Function

The third way to build a TF1 is to define a function yourself and then give its name to the constructor. A function for a TF1 constructor needs to have this exact signature:

\[
\text{Double_t fitf(Double_t *x, Double_t *par)}
\]

The two parameters are:

- **Double_t *x**: a pointer to the dimension array. Each element contains a dimension. For a 1D histogram only x[0] is used, for a 2D histogram x[0] and x[1] is used, and for a 3D histogram x[0], x[1], and x[2] are used. For histograms, only 3 dimensions apply, but this method is also used to fit other objects, for example an ntuple could have 10 dimensions.
- **Double_t *par**: a pointer to the parameters array. This array contains the current values of parameters when it is called by the fitting function.

The following script `$ROOTSYS/tutorials/myfit.C` illustrates how to fit a 1D histogram with a user-defined function. First we declare the function.

```c
// define a function with 3 parameters
Double_t fitf(Double_t *x, Double_t *par)
{
    Double_t arg = 0;
    if (par[2] != 0) arg = (x[0] - par[1])/par[2];
    Double_t fitval = par[0]*TMath::Exp(-0.5*arg*arg);
    return fitval;
}
```

Now we use the function:

```c
// this function used fitf to fit a histogram
void fitexample()
{
    // open a file and get a histogram
    TFile *f = new TFile("hsimple.root");
    TH1F *hpx = (TH1F*)f->Get("hpx");

    // Create a TF1 object using the function defined above.
    // The last three parameters specify the number of parameters
    // for the function.
    TF1 *func = new TF1("fit", fitf, -3, 3, 3);

    // set the parameters to the mean and RMS of the histogram
    func->SetParameters(500, hpx->GetMean(), hpx->GetRMS());

    // give the parameters meaningful names
    func->SetParNames( "Constant", "Mean_value", "Sigma" );

    // call TH1::Fit with the name of the TF1 object
    hpx->Fit("fit");
}
```

Fixing and Setting Bounds for Parameters

Parameters must be initialized before invoking the Fit method. The setting of the parameter initial values is automatic for the predefined functions: poln, exp, gaus. You can disable the automatic computation by specifying the "B" option when calling the Fit method. When a function is not predefined, the fit parameters must be initialized to some value as close as possible to the expected values before calling the fit function.
To set bounds for one parameter, use `TF1::SetParLimits`:

```c
func->SetParLimits(0,-1,1);
```

When the lower and upper limits are equal, the parameter is fixed. This statement fixes parameter 4 at 10.

```c
func->SetParameters(4,10)
func->SetParLimits(4,77,77);
```

However, to fix a parameter to 0, one must call the `FixParameter` function:

```c
func->SetParameter(4,0)
func->FixParameter(4,0);
```

Note that you are not forced to fix the limits for all parameters. For example, if you fit a function with 6 parameters, you can:

```c
func->SetParameters(0,3.1,1.e-6,-1.5,0,100);
func->SetParLimits(3,-10,-4);
func->FixParameter(4,0);
```

With this setup, parameters 0->2 can vary freely, parameter 3 has boundaries [-10,-4] with initial value –8, and parameter 4 is fixed to 0.

### Fitting Sub Ranges

By default, `TH1::Fit` will fit the function on the defined histogram range. You can specify the option "R" in the second parameter of `TH1::Fit` to restrict the fit to the range specified in the `TF1` constructor. In this example, the fit will be limited to –3 to 3, the range specified in the `TF1` constructor.

```c
root[] TF1 *f1 = new TF1("f1","[0]*x*sin([1]*x)",-3,3);
root[] hist->Fit("f1","R");
```

You can also specify a range in the call to `TH1::Fit`:

```c
root[] hist->Fit("f1","",",",-2,2)
```

For more complete examples, see `$ROOTSYS/tutorials/myfit.C` and `$ROOTSYS/tutorials/multifit.C`.

### Example: Fitting Multiple Sub Ranges

The script for this example is in macro `multifit.C` in `$ROOTSYS/tutorials/` directory. It shows how to use several Gaussian functions with different parameters on separate sub ranges of the same histogram.

To use a Gaussian, or any other ROOT built in function, on a sub range you need to define a new `TF1`. Each is 'derived' from the canned function `gaus`. 

![Example of several fits in subranges](image-url)
Here we fill a histogram with bins defined in the array $x$ (see $\text{ROOTSYS/tutorials/multifit.C}$).

When fitting simple functions, such as a Gaussian, the initial values of the parameters are automatically computed by ROOT. In the more complicated case of the sum of 3 Gaussian functions, the initial values of parameters must be set. In this particular case, the initial values are taken from the result of the individual fits.

The use of the "+" sign is explained below:

Adding Functions to the List

The example $\text{ROOTSYS/tutorials/multifit.C}$ also illustrates how to fit several functions on the same histogram. By default a Fit command deletes the previously fitted function in the histogram object. You can specify the option "+" in the second parameter to add the newly fitted function to the existing list of functions for the histogram.

Note that the fitted function(s) are saved with the histogram when it is written to a ROOT file.
Combining Functions

You can combine functions to fit a histogram with their sum. Here is an example, the code is in $ROOTSYS/tutorials/FitDemo.C. We have a function that is the combination of a background and lorenzian peak. Each function contributes 3 parameters.

\[y(E) = a_1 + a_2E + a_3E^2 + A_p \left(\frac{G}{2} p\right)\left(\frac{(E-m)^2 + (G/2)^2}{(E-m)^2 + (G/2)^2}\right)\]

- **background**
  - \[par[0] = a_1\]
  - \[par[1] = a_2\]
  - \[par[2] = a_3\]

- **lorenzian peak**
  - \[par[0] = A_p\]
  - \[par[1] = G\]
  - \[par[2] = m\]

The combination function (fitFunction) has six parameters:

\[fitFunction = background(x, \text{par}) + lorenzianPeak(x, \&\text{par}[3])\]

- \[par[0]\] = \[a_1\]
- \[par[1]\] = \[a_2\]
- \[par[2]\] = \[a_3\]
- \[par[3]\] = \[A_p\]
- \[par[4]\] = \[G\]
- \[par[5]\] = \[m\]

This script creates a histogram and fits the combination of the two functions. First we define the two functions and the combination function:

```c
// Quadratic background function
Double_t background(Double_t *x, Double_t *par) {
    return par[0] + par[1]*x[0] + par[2]*x[0]*x[0];
}

// Lorentzian Peak function
Double_t lorentzianPeak(Double_t *x, Double_t *par) {
    return (0.5*par[0]*par[1]/TMath::Pi()) / TMath::Max(1.e-10,
                (x[0]-par[2])*(x[0]-par[2])+ .25*par[1]*par[1]);
}

// Sum of background and peak function
Double_t fitFunction(Double_t *x, Double_t *par) {
    return background(x,par) + lorentzianPeak(x,&par[3]);
}

void FittingDemo() {
    // bevington exercise by P. Malzacher, modified by R. Brun
    const int nBins = 60;
    Stat_t data[nBins] = { 6, 1,10,12, 6,13,23,22,15,21,
                           23,26,36,25,27,35,40,44,66,81,
                           75,57,48,45,46,41,35,36,53,32,
                           40,37,38,31,36,44,42,37,32,32,
                           43,44,35,33,33,39,29,41,32,44,
                           26,39,29,35,32,21,21,15,25,15};

    TH1F *histo = new TH1F("example_9_1",
                             "Lorentzian Peak on Quadratic Background",60,0,3);

    for(int i=0; i < nBins; i++) {
        // we use these methods to explicitly set the content
        // and error instead of using the fill method.
        histo->SetBinContent(i+1,data[i]);
        histo->SetBinError(i+1,TMath::Sqrt(data[i]));
    }
    // continued...
```
// create a TF1 with the range from 0 to 3 and 6 parameters
TF1 *fitFcn = new TF1("fitFcn",fitFunction,0,3,6);

// first try without starting values for the parameters
// this defaults to 1 for each param.
histo->Fit("fitFcn");
// this results in an ok fit for the polynomial function however
// the non-linear part (lorenzian) does not respond well
// second try: set start values for some parameters
fitFcn->SetParameter(4,0.2);  // width
fitFcn->SetParameter(5,1);    // peak

histo->Fit("fitFcn","V+");

// improve the picture:
TF1 *backFcn = new TF1("backFcn",background,0,3,3);
backFcn->SetLineColor(3);
TF1 *signalFcn = new TF1("signalFcn",lorentzianPeak,0,3,3);
signalFcn->SetLineColor(4);
Double_t par[6];
// writes the fit results into the par array
fitFcn->GetParameters(par);

backFcn->SetParameters(par);
backFcn->Draw("same");
signalFcn->SetParameters(&par[3]);
signalFcn->Draw("same");
}

This is the result:

For another example see: http://root.cern.ch/root/html/examples/backsig.C.html
Associated Function

One or more objects (typically a TF1*) can be added to the list of functions (fFunctions) associated to each histogram. A call to TH1::Fit adds the fitted function to this list. Given a histogram h, one can retrieve the associated function with:

```c
TF1 *myfunc = h->GetFunction("myfunc");
```

Access to the Fit Parameters and Results

If the histogram (or graph) is made persistent, the list of associated functions is also persistent. Retrieve a pointer to the function with the TH1::GetFunction() method. Then you can retrieve the fit parameters from the function (TF1) with calls such as:

```c
root[] TF1 *fit = hist->GetFunction(function_name);
root[] Double_t chi2 = fit->GetChisquare();
// value of the first parameter
root[] Double_t p1 = fit->GetParameter(0);
// error of the first parameter
root[] Double_t e1 = fit->GetParError(0);
```

Associated Errors

By default, for each bin, the sum of weights is computed at fill time. One can also call TH1::Sumw2 to force the storage and computation of the sum of the square of weights per bin. If Sumw2 has been called, the error per bin is computed as the \( \sqrt{\text{sum of squares of weights}} \), otherwise the error is set equal to the \( \sqrt{\text{bin content}} \).

To return the error for a given bin number, do:

```c
Double_t error = h->GetBinError(bin);
```

Fit Statistics

You can change the statistics box to display the fit parameters with the TStyle::SetOptFit(mode) method. This mode has four digits.

```c
mode = pcev (default = 0111)
```

- \( v = 1 \) print name/values of parameters
- \( e = 1 \) print errors (if \( e=1, v \) must be 1)
- \( c = 1 \) print Chi-square/number of degrees of freedom
- \( p = 1 \) print probability

For example:

```c
gStyle->SetOptFit(1011);
```

This prints the fit probability, parameter names/values, and errors.
The Minimization Package

This package was originally written in FORTRAN by Fred James and part of PACKLIB (patch D506). It has been converted to a C++ class by Rene Brun. The current implementation in C++ is a straightforward conversion of the original FORTRAN version. The main changes are:

- The variables in the various Minuit labeled common blocks have been changed to the TMinuit class data members
- The internal arrays with a maximum dimension depending on the maximum number of parameters are now data members’ arrays with a dynamic dimension such that one can fit very large problems by simply initializing the TMinuit constructor with the maximum number of parameters
- The include file Minuit.h has been commented as much as possible using existing comments in the code or the printed documentation
- The original Minuit subroutines are now member functions
- Constructors and destructor have been added
- Instead of passing the FCN function in the argument list, the addresses of this function is stored as pointer in the data members of the class. This is by far more elegant and flexible in an interactive environment. The member function SetFCN can be used to define this pointer
- The ROOT static function Printf is provided to replace all format statements and to print on currently defined output file
- The derived class TMinuitOld contains obsolete routines from the FORTRAN based version
- The functions SetObjectFit/GetObjectFit can be used inside the FCN function to set/get a referenced object instead of using global variables
- By default fGraphicsMode is true. When calling the Minuit functions such as mncont, mnscan, or any Minuit command invoking mnplot, TMinuit::mnplot() produces a TGraph object pointed by fPlot. One can retrieve this object with TMinuit::GetPlot(). For example:

```c
h->Fit("gaus");
gMinuit->Command("SCAn 1");
TGraph *gr = (TGraph*)gMinuit->GetPlot();
gr->SetMarkerStyle(21);
gr->Draw("alp");
```

- To set Minuit in no graphics mode, call

```c
gMinuit->SetGraphicsMode(kFALSE);
```

Basic Concepts of Minuit

The Minuit package acts on a multi parameter FORTRAN function to which one must give the generic name FCN. In the ROOT implementation, the function FCN is defined via the Minuit SetFCN member function when a HistogramFit command is invoked. The value of FCN will in general depend on one or more variable parameters.

To take a simple example, in case of ROOT histograms (classes TH1C, TH1S, TH1F, TH1D) the Fit function defines the Minuit fitting function as being HIFitChisquare or HIFitLikelihood depending on the options selected. HIFitChisquare calculates the chi-square between the user fitting function (Gaussian, polynomial, user defined, etc) and the data for given values of the parameters. It is the task of Minuit to find those values of the parameters which give the lowest value of chi-square.
The Transformation of Limited Parameters

For variable parameters with limits, Minuit uses the following transformation:

\[ P_{\text{int}} = \arcsin \left( 2 \left( \frac{P_{\text{ext}} - a}{b-a} \right) - 1 \right) \]

\[ P_{\text{ext}} = a + \left( \frac{b - a}{2} \right) \left( \sin P_{\text{int}} + 1 \right) \]

so that the internal value \( P_{\text{int}} \) can take on any value, while the external value \( P_{\text{ext}} \) can take on values only between the lower limit \( a \) and the ext upper limit \( b \). Since the transformation is necessarily non-linear, it would transform a nice linear problem into a nasty non-linear one, which is the reason why limits should be avoided if not necessary. In addition, the transformation does require some computer time, so it slows down the computation a little bit, and more importantly, it introduces additional numerical inaccuracy into the problem in addition to what is introduced in the numerical calculation of the FCN value. The effects of non-linearity and numerical round off both become more important as the external value gets closer to one of the limits (expressed as the distance to nearest limit divided by distance between limits). The user must therefore be aware of the fact that, for example, if he puts limits of \((0, 10^{10})\) on a parameter, then the values 0.0 and 1.0 will be indistinguishable to the accuracy of most machines.

The transformation also affects the parameter error matrix, of course, so Minuit does a transformation of the error matrix (and the "parabolic" parameter errors) when there are parameter limits. Users should however realize that the transformation is only a linear approximation, and that it cannot give a meaningful result if one or more parameters is very close to a limit, where \( \frac{\partial P_{\text{ext}}}{\partial P_{\text{int}}} \neq 0 \). Therefore, it is recommended that:

- Limits on variable parameters should be used only when needed in order to prevent the parameter from taking on unphysical values
- When a satisfactory minimum has been found using limits, the limits should then be removed if possible, in order to perform or re-perform the error analysis without limits

How to Get the Right Answer from Minuit

Minuit offers the user a choice of several minimization algorithms. The MIGRAD algorithm is in general the best minimized for nearly all functions. It is a variable-metric method with inexact line search, a stable metric updating scheme, and checks for positive-definiteness. Its main weakness is that it depends heavily on knowledge of the first derivatives, and fails miserably if they are very inaccurate.

If parameter limits are needed, in spite of the side effects, then the user should be aware of the following techniques to alleviate problems caused by limits:

**Getting the Right Minimum with limits**

If MIGRAD converges normally to a point where no parameter is near one of its limits, then the existence of limits has probably not prevented Minuit from finding the right minimum. On the other hand, if one or more parameters is near its limit at the minimum, this may be because the true minimum is indeed at a limit, or it may be because the minimized has become "blocked" at a limit. This may normally happen only if the parameter is so close to a limit (internal value at an odd multiple of \( \pi/(2) \)) that Minuit prints a warning to this effect when it prints the parameter values. The minimized can become blocked at a limit, because at a limit the derivative seen by the minimized \( \frac{\partial F}{\partial P_{\text{int}}} \) is zero no matter what the real derivative \( \frac{\partial F}{\partial P_{\text{int}}} \) is.

\[ \left( \frac{\partial F}{\partial P_{\text{int}}} \right) = \left( \frac{\partial F}{\partial P_{\text{ext}}} \right) \left( \frac{\partial P_{\text{ext}}}{\partial P_{\text{int}}} \right) = 0 \]

**Getting the Right Parameter Errors with Limits**

In the best case, where the minimum is far from any limits, Minuit will correctly transform the error matrix, and the parameter errors it reports should be accurate and very close to those you would have got without limits. In other cases (which should be more common,
since otherwise you wouldn't need limits), the very meaning of parameter errors becomes problematic. Mathematically, since the limit is an absolute constraint on the parameter, a parameter at its limit has no error, at least in one direction. The error matrix, which can assign only symmetric errors, then becomes essentially meaningless.

**Interpretation of Parameter Errors**

There are two kinds of problems that can arise: the reliability of Minuit's error estimates, and their statistical interpretation, assuming they are accurate.

**Statistical Interpretation**

For discussion of basic concepts, such as the meaning of the elements of the error matrix, or setting of exact confidence levels see the articles:


**Reliability of Minuit Error Estimates**

Minuit always carries around its own current estimates of the parameter errors, which it will print out on request, no matter how accurate they are at any given point in the execution. For example, at initialization, these estimates are just the starting step sizes as specified by the user. After a HESSE step, the errors are usually quite accurate, unless there has been a problem. Minuit, when it prints out error values, also gives some indication of how reliable it thinks they are. For example, those marked CURRENT GUESS ERROR are only working values not to be believed, and APPROXIMATE ERROR means that they have been calculated but there is reason to believe that they may not be accurate.

If no mitigating adjective is given, then at least Minuit believes the errors are accurate, although there is always a small chance that Minuit has been fooled. Some visible signs that Minuit may have been fooled are:

- Warning messages produced during the minimization or error analysis
- Failure to find new minimum
- Value of EDM too big (estimated Distance to Minimum)
- Correlation coefficients exactly equal to zero, unless some parameters are known to be uncorrelated with the others
- Correlation coefficients very close to one (greater than 0.99). This indicates both an exceptionally difficult problem, and one which has been badly parameterized so that individual errors are not very meaningful because they are so highly correlated
- Parameter at limit. This condition, signaled by a Minuit warning message, may make both the function minimum and parameter errors unreliable. See the discussion above ‘Getting the right parameter errors with limits’

The best way to be absolutely sure of the errors, is to use "independent" calculations and compare them, or compare the calculated errors with a picture of the function. Theoretically, the covariance matrix for a "physical" function must be positive-definite at the minimum, although it may not be so for all points far away from the minimum, even for a well-determined physical problem. Therefore, if MIGRAD reports that it has found a non-positive-definite covariance matrix, this may be a sign of one or more of the following:

**A non-physical region**

On its way to the minimum, MIGRAD may have traversed a region which has unphysical behavior, which is of course not a serious problem as long as it recovers and leaves such a region.

**An underdetermined problem**

If the matrix is not positive-definite even at the minimum, this may mean that the solution is not well-defined, for example that there are more unknowns than there are data points, or
that the parameterization of the fit contains a linear dependence. If this is the case, then Minuit (or any other program) cannot solve your problem uniquely. The error matrix will necessarily be largely meaningless, so the user must remove the under determinedness by reformulating the parameterization. Minuit cannot do this itself.

**Numerical inaccuracies**

It is possible that the apparent lack of positive-definiteness is in fact only due to excessive round off errors in numerical calculations in the user function or not enough precision. This is unlikely in general, but becomes more likely if the number of free parameters is very large, or if the parameters are badly scaled (not all of the same order of magnitude), and correlations are also large. In any case, whether the non-positive-definiteness is real or only numerical is largely irrelevant, since in both cases the error matrix will be unreliable and the minimum suspicious.

**An ill-posed problem**

For questions of parameter dependence, see the discussion above on positive-definiteness. Possible other mathematical problems are the following:

**Excessive numerical round off**

Be especially careful of exponential and factorial functions which get big very quickly and lose accuracy.

**Starting too far from the solution**

The function may have unphysical local minima, especially at infinity in some variables.

---

**FUMILI Minimization Package**

FUMILI is used to minimize Chi-square function or to search maximum of likelihood function. Experimentally measured values \( F_i \) are fitted with theoretical functions \( f_i (\vec{x}, \vec{\theta}) \), where \( \vec{x} \) are coordinates, and \( \vec{\theta} \) - vector of parameters. For better convergence Chi-square function has to be the following form

\[
\frac{\chi^2}{2} = \frac{1}{2} \sum_{i=1}^{n} \left( \frac{f_i(\vec{x}, \vec{\theta}) - F_i}{\sigma_i} \right)^2
\]

where \( \sigma_i \) are errors of measured function. The minimum condition is:

\[
\frac{\partial \chi^2}{\partial \theta_i} = \sum_{j=1}^{n} \frac{1}{\sigma_j^2} \frac{\partial f_j}{\partial \theta_i} \left[ f_j(\vec{x}, \vec{\theta}) - F_j \right] = 0, \quad i = 1 \ldots m
\]

where \( m \) is the quantity of parameters. Expanding left part of this equation over parameter increments and retaining only linear terms one gets

\[
\left( \frac{\partial \chi^2}{\partial \theta_i} \right)_{\vec{\theta} = \vec{\theta}_0} + \sum_k \left( \frac{\partial^2 \chi^2}{\partial \theta_i \partial \theta_k} \right)_{\vec{\theta} = \vec{\theta}_0} \cdot (\theta_k - \theta_0^k) = 0
\]

here \( \theta_0 \) is some initial value of parameters. In general case:

\[
\frac{\partial^2 \chi^2}{\partial \theta_i \partial \theta_k} = \sum_{j=1}^{n} \frac{1}{\sigma_j^2} \frac{\partial f_j}{\partial \theta_i} \frac{\partial f_j}{\partial \theta_k} + \sum_{j=1}^{n} \frac{(f_j - F_j)}{\sigma_j^2} \frac{\partial^2 f_j}{\partial \theta_i \partial \theta_k}
\]
In FUMILI algorithm for second derivatives of Chi-square approximate expression is used when last term in previous equation is discarded. It is often done, not always wittingly, and sometimes causes troubles, for example, if user wants to limit parameters with positive values by writing down $\theta^2_i$ instead of $\theta_i$. FUMILI will fail if one tries minimize $\chi^2 = g^2(\theta)$ where $g$ is arbitrary function.

Approximate value is:

$$\frac{\partial^2 \chi^2}{\partial \theta_i \partial \theta_k} \approx Z_{ik} = \sum_{j=1}^{n} \frac{1}{\sigma_j^2} \frac{\partial f_j}{\partial \theta_i} \frac{\partial f_j}{\partial \theta_k}$$

Then the equations for parameter increments are

$$\left(\frac{\partial \chi^2}{\partial \theta_i}\right)_{\theta = \theta^0} + \sum_k Z_{ik} \cdot (\theta_k - \theta^0_k) = 0, \quad i = 1 \ldots m$$

Remarkable feature of algorithm is the technique for step restriction. For an initial value of parameter $\theta^0$ a parallelepiped $P_0$ is built with the center at $\theta^0$ and axes parallel to coordinate axes $\theta_i$. The lengths of parallelepiped sides along $i$-axis is $2b_i$, where $b_i$ is such a value that the functions $f_j(\theta)$ are quasi-linear all over the parallelepiped.

FUMILI takes into account simple linear inequalities in the form:

$$\theta^\text{min}_i \leq \theta_i \leq \theta^\text{max}_i$$

They form parallelepiped $P$ ($P_0$ may be deformed by $P$). Very similar step formulae are used in FUMILI for negative logarithm of the likelihood function with the same idea - linearization of functional argument.
Neural Networks

Introduction

Neural Networks are more and more used in various fields for data analysis and classification, both for research and commercial institutions. Some randomly chosen examples are image analysis, financial movements predictions and analysis, or sales forecast and product shipping optimization. In particles physics neural networks are mainly used for classification tasks (signal over background discrimination). A vast majority of commonly used neural networks are multilayer perceptrons. This implementation of multilayer perceptrons is inspired from the MLPfit package, which remains one of the fastest tool for neural networks studies.

The MLP

The multilayer perceptron is a simple feed-forward network with the following structure:

![MLP Diagram](image)

It is made of neurons characterized by a bias and weighted links in between - let's call those links synapses. The input neurons receive the inputs, normalize them and forward them to the first hidden layer. Each neuron in any subsequent layer first computes a linear combination of the outputs of the previous layer. The output of the neuron is then function of that combination with $f$ being linear for output neurons or a sigmoid for hidden layers.

Such a structure is very useful because of two theorems:

1. A linear combination of sigmoids can approximate any continuous function.
2. Trained with output=1 for the signal and 0 for the background, the approximated function of inputs $x$ is the probability of signal, knowing $x$.

Learning Methods

The aim of all learning methods is to minimize the total error on a set of weighted examples. The error is defined as the sum in quadrate, divided by two, of the error on each individual output neuron. In all methods implemented in this library, one needs to compute the first derivative of that error with respect to the weights. Exploiting the well-known properties of the derivative, one can express this derivative as the product of the local partial derivative by the weighted sum of the outputs derivatives (for a neuron) or as the product of the input value with the local partial derivative of the output neuron (for a synapse). This computation is called "back-propagation of the errors". Six learning methods are implemented.
Stochastic Minimization

This is the most trivial learning method. This is the Robbins-Monro stochastic approximation applied to multilayer perceptrons. The weights are updated after each example according to the formula:

\[ w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \]

with

\[ \Delta w_{ij}(t) = -\eta(\partial L / \partial w_{ij} + \delta) + \epsilon \Delta w_{ij}(t-1) \]

The parameters for this method are \( \text{Eta}, \text{EtaDecay}, \text{Delta} \) and \( \text{Epsilon} \).

Steepest Descent With Fixed Step Size (Batch Learning)

It is the same as the stochastic minimization, but the weights are updated after considering all the examples, with the total derivative \( \frac{dE}{dw} \). The parameters for this method are \( \text{Eta}, \text{EtaDecay}, \text{Delta} \) and \( \text{Epsilon} \).

Steepest Descent Algorithm

Weights are set to the minimum along the line defined by the gradient. The only parameter for this method is \( \text{Tau} \).

Lower \( \text{Tau} \) = higher precision = slower search.

A value \( \text{Tau}=3 \) seems reasonable.

Conjugate Gradients With the Polak-Ribiere Updating Formula

Weights are set to the minimum along the line defined by the conjugate gradient. Parameters are \( \text{Tau} \) and \( \text{Reset} \), which defines the epochs where the direction is reset to the steepest descent.

Conjugate Gradients With the Fletcher-Reeves Updating Formula

Weights are set to the minimum along the line defined by the conjugate gradient. Parameters are \( \text{Tau} \) and \( \text{Reset} \), which defines the epochs where the direction is reset to the steepest descent.

The Broyden, Fletcher, Goldfarb, Shanno (BFGS) Method

It implies the computation of a \( N \times N \) matrix, but seems more powerful at least for less than 300 weights. Parameters are \( \text{Tau} \) and \( \text{Reset} \), which defines the epochs where the direction is reset to the steepest descent.

Using the Network

Neural network are build from a set of "samples". A sample is a set of values defining the inputs and the corresponding output that the network should ideally provide. In ROOT this is a \text{TTree} entry. The first thing to be decided is the network layout. This layout is described in a string where the layers are separated by semicolons. The input/output layers are defined by giving the expression for each neuron, separated by commas. Hidden layers are just described by the number of neurons. In addition, output layer formulas can be preceded by '@' (e.g. "@out") if one wants to also normalize the output (by default, only input neurons are normalized). Expressions are evaluated as for \text{TTree}::\text{Draw}().

// a simple network: 2 inputs, 8 hidden and 1 normalized output neuron
TMultiLayerPerceptron network("r,z:10:@Br",tree);

Input and outputs are taken from the \text{TTree} associated with the network. This \text{TTree} can be given as argument of the constructor or defined later with
TMultiLayerPerceptron::setData(). Events can also be weighted. The weight
expression must be given in the constructor or set later with
TMultiLayerPerceptron::setWeight().

Two datasets must be defined before learning the network: a training dataset which is used
when minimizing the error, and a test dataset which will avoid bias. Those two datasets can
be build aside and then given to the network, or can be build from a standard expression. By
default, half of the events are put in both datasets.

```c++
// a more complex 4:2:1 network
// the ptsumf branch is used as weigh; default event lists are explicit
TMultiLayerPerceptron network("m,pt,acol,acopl:8:type","pt",
    "Entry$%2","Entry$/2");
```

The learning method is defined by TMultiLayerPerceptron::setLearningMethod().
Learning methods are:

```
TMultiLayerPerceptron::kStochastic,
TMultiLayerPerceptron::kBatch,
TMultiLayerPerceptron::kSteepestDescent,
TMultiLayerPerceptron::kRibierePolak,
TMultiLayerPerceptron::kFletcherReeves,
TMultiLayerPerceptron::kBFGS // default
```

One starts the training with TMultiLayerPerceptron::Train(Int_t nepoch,
Option_t* options). The first argument is the number of epochs while option is a string
that can contain: "text" (simple text output), "graph" (evaluating graphical training curves),
"update=X" (step for the text/graph output update) or "+" (will skip the randomization and
start from the previous values). All combinations are available.

```
network.Train(1000,"text,graph,update=10"); //full output every 10 epochs
network.Train(100,"text,+"); //100 more epochs
```

The weights can be saved to a file (DumpWeights) and then reloaded (LoadWeights) to a
new compatible network. The output can also be evaluated (Evaluate) for a given output
neuron and an array of double input parameters or the network can be exported (Export)
as a standalone code. Up to now this is only as a C++ or PYTHON class, but other
languages could be implemented.

**Examples**

An example of how to use TMultiLayerPerceptron is the macro mlpHiggs.C in
$ROOTSYS/tutorials. Using some standard simulated information that could have been
obtained at LEP, a neural network is build which can make the difference between WW events
and events containing a Higgs boson. Starting with a TFile containing two TTrees: one for
the signal, the other for the background, a simple script is used:

```c++
void mlpHiggs(Int_t ntrain=100)
{
    if (!gROOT->GetClass("TMultiLayerPerceptron"))
        gSystem->Load("libMLP");

    // prepare inputs - the 2 trees are merged into one, and a "type"
    // branch, equal to 1 for the signal and 0 for the background is added
    TFile *mlpHiggs = new TFile("mlpHiggs.root");
    TTree *signal = (TTree *) mlpHiggs->Get("sig_filtered");
    TTree *background = (TTree *) mlpHiggs->Get("bg_filtered");
    TTree *simu = new TTree("MonteCarlo", "Filtered Monte Carlo Events");
    ...
```

Since the input is a TTree and we are starting from two different TTrees (with different
names), they are first merged into one, and a "type" branch is added, that says whether
there is a signal or a background event. Those irrelevant details are skipped here.
TMultiLayerPerceptron *mlp = new TMultiLayerPerceptron("msumf,ptsumf,
acolin, acopl:8:type",
"ptsumf",simu,"Entry$%2","Entry$/2");
mlp->Train(ntrain, "text,graph,update=10");

The neural network is instantiated and trained. "ptsumf" is used as a weight, and the
standard event lists are explicit. The network that is then build has four input neurons, eight
additional ones in the only hidden layer and one single output neuron.

// Use the NN to plot the results for each sample
TH1F *bg = new TH1F("bgH","NN output",50,-.5,1.5);
TH1F *sig = new TH1F("sigh","NN output",50,-.5,1.5);
bg->SetDirectory(0);
sig->SetDirectory(0);
Double_t params[4];

for (i = 0; i < background->GetEntries(); i++) {
    background->GetEntry(i);
    params[0] = msumf;
    params[1] = ptsumf;
    params[2] = acolin;
    params[3] = acopl;
    bg->Fill(mlp->Evaluate(0, params));
}

for (i = 0; i < signal->GetEntries(); i++) {
    signal->GetEntry(i);
    params[0] = msumf;
    params[1] = ptsumf;
    params[2] = acolin;
    params[3] = acopl;
    sig->Fill(mlp->Evaluate(0, params));
}

TCanvas *cv = new TCanvas("NNout_cv","Neural net output");
bg->SetFillStyle(3008);
bg->SetFillColor(kBlue);
sig->SetFillStyle(3003);
sig->SetFillColor(kRed);
bg->SetStats(0);
sig->SetStats(0);
bg->Draw();
sig->Draw("same");

TLegend *legend = new TLegend(.75,.80,.95,.95);
legend->AddEntry(bg,"Background(WW)");
legend->AddEntry(sig,"Signal(Higgs)");
legend->Draw();

The neural net output is then used to display the final difference between background and
signal events. This plot is shown next.
As it can be seen, this is a quite efficient technique. As mentioned earlier, neural networks are also used for fitting function. For some application with a cylindrical symmetry, a magnetic field simulation gives as output the angular component of the potential vector \( A \), as well as the radial and \( z \) components of the \( B \) field.

One wants to fit those distributions with a function in order to plug them into the Geant simulation code. Polynomial fits could be tried, but it seems difficult to reach the desired precision over the full range. One could also use a spline interpolation between known points. In all cases, the resulting field would not be \( C \)-infinite.

An example of output (for Br) is shown. First, the initial function that can be seen as the target. Then, the resulting (normalized) neural net output. In order to ease the learning, the "normalize output" was used here. The initial amplitude can be recovered by multiplying by the original RMS and then shifting by the original mean.