Document version 2.07-29 – 11 Jan 2006

RooFit Users Manual v2.07

W. Verkerke, D. Kirkby

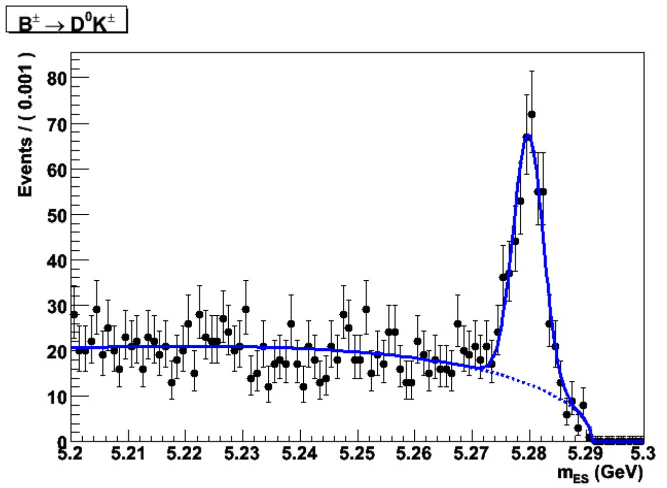
Table of Contents

What is RooFit?

The RooFit library provides a toolkit for modeling the expected distribution of events in a physics analysis. Models can be used to perform likelihood fits, produce plots, and generate "toy Monte Carlo" samples for various studies. The RooFit tools are integrated with the object-oriented and interactive ROOT graphical environment.

RooFit was originally developed for the BaBar collaboration, a particle physics experiment at the Stanford Linear Accelerator Center. This software is primarily designed as a particle physics data analysis tool, but its general nature and open architecture make it useful for other types of data analysis also.

```
// --- Observable ---
RooRealVar mes("mes","m_{ES} (GeV)",5.20,5.30) ;

// --- Build Gaussian signal PDF ---
RooRealVar sigmean("sigmean","B^{#pm} mass",5.28,5.20,5.30) ;
RooRealVar sigwidth("sigwidth","B^{#pm} width",0.0027,0.001,1.) ;
RooGaussian gauss("gauss","gaussian PDF",mes,sigmean,sigwidth) ;

// --- Build Argus background PDF ---
RooRealVar argpar("argpar","argus shape parameter",-20.0,-100.,-1.) ;
RooArgusBG argus("argus","Argus PDF",mes,RooConst(5.291),argpar) ;

// --- Construct signal+background PDF ---
RooRealVar nsig("nsig","#signal events",200,0.,10000) ;
RooRealVar nbkg("nbkg","#background events",800,0.,10000) ;
RooAddPdf sum("sum","g+a",RooArgList(gauss,argus),RooArgList(nsig,nbkg)) ;

// --- Generate a toyMC sample from composite PDF ---
RooDataSet *data = sum.generate(mes,2000) ;

// --- Perform extended ML fit of composite PDF to toy data ---
sum.fitTo(*data,Extended()) ;

// --- Plot toy data and composite PDF overlaid ---
RooPlot* mesframe = mes.frame() ;
data->plotOn(mesframe) ;
sum.plotOn(mesframe) ;
sum.plotOn(mesframe,Components(argus),LineStyle(kDashed)) ;
```



Example 1 – A non-trivial code example: Using RooFit to perform an extended unbinned maximum likelihood fit of a Gaussian signal on top of a Argus shaped background to toy Monte Carlo data sampled from the same model.

1. Installing RooFit

ROOT5

RooFit is distributed as external package with ROOT and is integrated into its `make` system from ROOT version 5.02 onward. Binary distributions of ROOT5 are shipped with pre-compiled RooFIt libraries and for those distributions you do not need to do anything special to obtain roofit. For source distributions of ROOT5, obtained either as source tarball, or from CVS you need to indicate in the makefile configuration that you wish to compile RooFit as well. To enable building of the RooFit library in ROOT5 add the `–enable-roofit` option to the `configure` command when you install ROOT.

```
unix> ./configure <platform_id> --enable-roofit
unix> make
```

If you have a full-source installation of ROOT you can also at any moment upgrade RooFit to the latest version in your existing installation of ROOT5 by downloading a new source tarball from `http://roofit.sourceforge.net/summary.php`. Go to 'File Releases' and down the latest 'RooFit (ROOT integrated)' tarball file. Untar the tarball to your ROOT installation directory and remake ROOT.

```
unix> cp roofit_vXYZ_root5kit.tar $ROOTSYS
unix> cd $ROOTSYS
unix> rm –rf roofit/
unix> tar –xvf roofit_vXYZ_root5kit.tar
unix> make
```

ROOT3 and ROOT4

RooFit also works with ROOT versions 3 and 4, but you need a full-source version of ROOT, either from a tarball or CVS and a source tarball of RooFit that you can obtain from the RooFit home page, as explained above. Download the `roofit_vXYZ_root4kit.tar` file and untar it in your ROOTSYS directory and issue a `make` command to rebuild ROOT. The `root4kit` tarball contains some extra `makefile` fragments that configure ROOT3/4 to recognize RooFit as a module.

Loading RooFit in ROOT

Once the RooFit library `is` available in `$ROOTSYS/lib`, you can use it in ROOT by executing the following commands

```
root> gSystem->Load("libRooFit") ;
root> using namespace RooFit ;
```

Be sure not to forget the second line, otherwise you will not see some of the helper functions that RooFit defines in the global namespace.

2. Getting started

In this section we will guide you through a simple exercise of building a model and fitting it to data. The aim is to familiarize you with several basic concepts and get you to a point where you can do something useful yourself quickly. In subsequent sections we will explore several aspects of RooFit in more detail

Building a model

A key concept in RooFit is that models (i.e. functions) are built in a truly object-oriented fashion. Each RooFit class has a one-to-one correspondences to a mathematical object: there is a class to express a variable, `RooRealVar`, a base class to express a function, `RooAbsReal`, a base class to express a probability density function, `RooAbsPdf`, to name a few. As even the simplest mathematical functions consists of multiple objects – i.e. the function itself and its variables – all RooFit models also consist of multiple objects. The following example illustrates this

```
RooRealVar x(“x”,”x”,-10,10) ;
RooRealVar mean(“mean”,”Mean of Gaussian”,0,-10,10) ;
RooRealVar sigma(“sigma”,”Width of Gaussian”,3,-10,10) ;

RooGaussian gauss(“gauss”,”gauss(x,mean,sigma)”,x,mean,sigma) ;
```

Example 2 – Construct a Gaussian probability density function

Each variable used in `gauss` is initialized with several properties: a name, a title, a range and optionally an initial value. Variables described by `RooRealVar` have more properties that are not visible in this example, for example an (a)symmetric error associated with the variable and a flag that specifies if the variable is constant or floating in a fit. In essence class `RooRealVar` collects all properties that are usually associated with a variable

The last line of code creates a Gaussian probability density function (PDF), as implemented in `RooGaussian`. Class `RooGaussian` is an implementation of the abstract base class `RooAbsPdf`, which described the common properties of all probability density functions. The PDF `gauss` has a name and a title, just like the variable objects, and is linked to the variables `x, mean` and `sigma` through the references passed in the constructor.

Visualizing a model

The first thing we usually want to do with a model is to see it. RooFit takes slightly more formal approach to visualization than plain ROOT. First you have to define a ‘view’, essentially an empty plot frame with one of the `RooRealVar` variables along the x-axis. Then, in OO style, you ask your model plot itself on the frame. Finally you draw the view on a ROOT `TCanvas`:

```
RooPlot* xframe = x.frame() ;
gauss.plotOn(frame) ;
frame->Draw()
```

The result of this example is shown in Figure 1. Note that in the creation of the view we do not have to specify a range, it is automatically taken from the range associated with the `RooRealVar`. It is of course possible to override this, we’ll return to this later. Note also that when `gauss` draws itself on the frame we don’t have to say that we want to plot gauss as function of `x`, this information is retrieved from the frame.

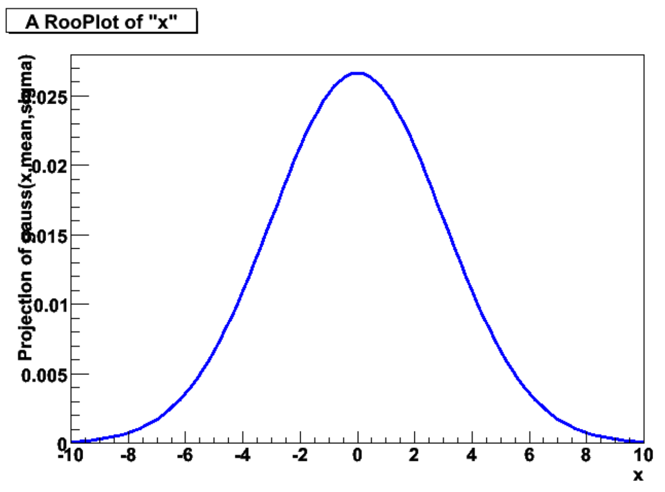


Figure 1 – Gaussian PDF

A frame can contain multiple objects (curves, histograms) to visualize. We can for example draw gauss twice with a different value of parameter sigma.

```
RooPlot* xframe = x.frame() ;
gauss.plotOn(frame) ;
sigma = 2 ;
gauss.plotOn(frame,LineColor(kRed)) ;
frame->Draw()
```

In this example we change the value of `RooRealVar` sigma after the first `plotOn()` command using the assignment operator. The color of the second curve is made red through additional `LineColor(kRed)` argument passed to `plotOn()`[1]. `LineColor` is an example of a 'named argument'. Named arguments are used throughout RooFit and provide a convenient and readable way to modify the default behavior of methods.Named arguments are covered in more detail in later sections. The output of the second code fragment in shown in Figure 2.

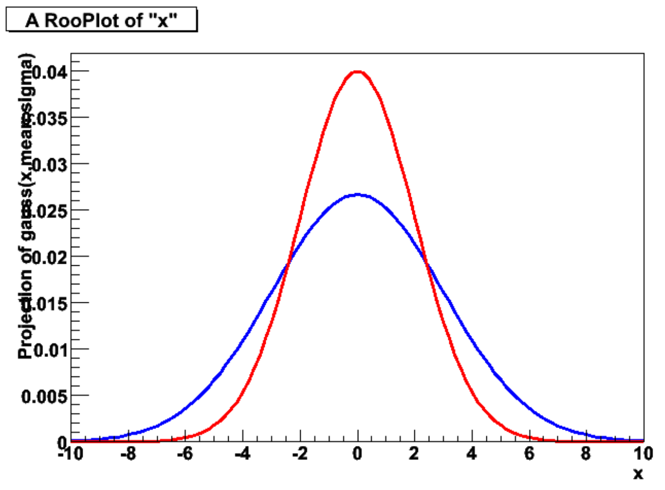


Figure 2 – Gaussian PDF with different widths

[1] If you get a ROOT error message at this point because `LineColor` is not defined, you have forgotten to include '`using namespace RooFit`' in your ROOT setup as was explained in Section 1.

The example also demonstrates that method plotOn() make a 'frozen' snapshot of the PDF: if the PDF changes shape after it has been drawn, as happens in the last code fragment, the already drawn curve will not change. Figure 2 also demonstrates that RooGaussian is always normalized to unity, regardless of the parameter values.

Importing data

Data analysis revolves around, well... data, so the next step is to import some data. Data in general comes in two flavors: unbinned data, represented in ROOT by class TTree and binned data, represented in ROOT by classes TH1,TH2 and TH3. RooFit can work with both.

Binned data (histograms)

In RooFit, binned data is represented by the RooDataHist class. You can import the contents of any ROOT histogram into a RooDataHist object

```
TH1* hh = (TH1*) gDirectory->Get("ahisto") ;
RooRealVar x("x","x",-10,10) ;
RooDataHist data("data","dataset with x",x,hh) ;
```

Example 3 – Importing data from a TTree and drawing it on a TCanvas

When you import a ROOT histogram the binning of the original histogram is imported as well. A RooDataHist always associates the histogram with a RooFit variable object of type RooRealVar. In this way it always known what kind of data is stored in the histogram.

A RooDataHist can be visualized in the same way as a function can be visualized:

```
RooPlot* xframe = x.frame() ;
data.plotOn(frame) ;
frame->Draw()
```

The result is shown in Figure 3.

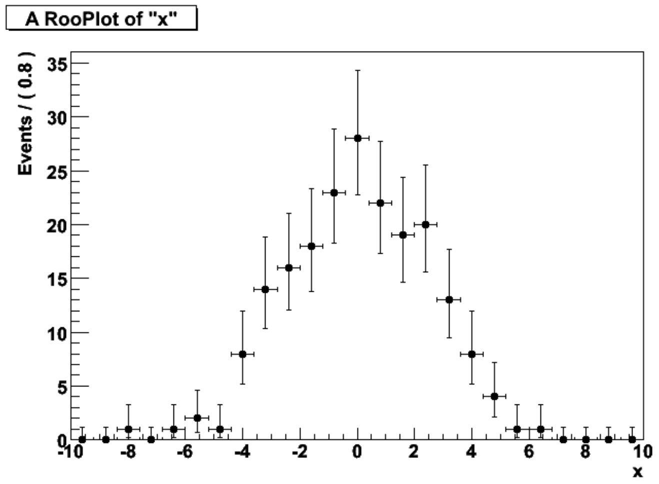


Figure 3 – Histogram visualized in RooFit

If you look closely at Figure 3 you will see that the error bars for entries at low statistics are not symmetric. This is not a mistake but a feature: at low statistics symmetric Gaussian errors of magnitude √N are only an approximation of the actual statistical uncertainty on a bin with N entries.

RooFit by default shows the 68% confidence interval for Poisson statistics[2], which is more difficult to calculate but also more accurate. Appendix C includes some basic statistics reading material that covers this and other issues.

Unbinned data (trees)

Unbinned data can be imported in RooFit much along the same lines and is store in class `RooDataSet`

```
TTree* tree = (TTree*) gDirectory->Get("atree") ;
RooRealVar x("x","x",-10,10) ;
RooDataSet data("data","dataset with x",x,tree) ;
```

In this example `tree` is assumed to have a branch named "x" as the `RooDataSet` constructor will import data from the tree branch that has the same name as the `RooRealVar` that is passed as argument.

Plotting unbinned data is similar to plotting binned data with the exception that you can now show it in any binning you like.

```
RooPlot* xframe = x.frame() ;
data.plotOn(frame,Binning(25)) ;
frame->Draw()
```

In this example we have overridden the default setting of 100 bins using the `Binning()` named argument.

Working with data

In general working with binned and unbinned data is very similar in RooFit as both class `RooDataSet` (for unbinned data) and class `RooDataHist` (for binned data) inherit from a common base class, `RooAbsData`, which defines the interface for a generic abstract data sample. With few exceptions, all RooFit methods take abstract datasets as input arguments, making it easy to use binned and unbinned data interchangeably.

The examples in this section have always dealt with one-dimensional datasets. Both `RooDataSet` and `RooDataHist` can however handle data with an arbitrary number of dimensions. In the next sections we will revisit datasets and explain how to work with multi-dimensional data.

Fitting a model to data

Fitting a model to data can be done in many ways. The most common methods are the χ^2 fit and the –log(L) fit. The default fitting method in ROOT is the χ^2 method, the default method in RooFit is the –log(L) method. We prefer the –log(L) method because it is more robust for low statistics fits and because it can also be performed on unbinned data. If you are unfamiliar with the basics of likelihood

[2] To be more precise the intervals shown are 'classic central ' intervals as described in Table I of Cousins, Am. J. Phys. 63, 398 (1995)

fitting we suggest you read through appendix C, which contains an easy introduction to the statistical theory behind χ^2 and –log(L) fitting and compares their relative advantages and disadvantages.

In practice both fitting techniques work very similar: first you construct the estimator quantity – either χ^2 or –log(L) – and then you perform the fit by finding the minimum value of the estimator with respect to all the parameters of the model. The errors on the fitted parameters are defined by the variation of the parameters that leads to a unit/half-unit increase of the χ^2/-log(L) respectively.

The standard tool in High Energy Physics to perform the minimization and error analysis since decades is MINUIT, and also RooFit delegates the minimization task to the ROOT implementation MINUIT in class `TMinuit`. RooFit is therefore more of a data modeling package rather than a fitting package.

The high-level interface to model fitting in RooFit packages all of the above in a very easy-to-use interface:

```
gauss.fitTo(data) ;
```

This command builds a –log(L) function from the `gauss` function and the given dataset, passes it to MINUIT, which minimizes it and estimate the errors on the parameters of `gauss`. The output of the `fitTo()` method produces the familiar MINUIT output on the screen:

```
**********
**   13 **MIGRAD        1000           1
**********
FIRST CALL TO USER FUNCTION AT NEW START POINT, WITH IFLAG=4.
START MIGRAD MINIMIZATION.  STRATEGY  1.  CONVERGENCE WHEN EDM .LT. 1.00e-03
FCN=25139.4 FROM MIGRAD    STATUS=INITIATE       10 CALLS          11 TOTAL
                    EDM= unknown      STRATEGY= 1      NO ERROR MATRIX
 EXT PARAMETER               CURRENT GUESS       STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  mean       -1.00000e+00   1.00000e+00   1.00000e+00  -6.53357e+01
  2  sigma        3.00000e+00   1.00000e+00   1.00000e+00  -3.60009e+01
                               ERR DEF= 0.5
MIGRAD MINIMIZATION HAS CONVERGED.
MIGRAD WILL VERIFY CONVERGENCE AND ERROR MATRIX.
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=25137.2 FROM MIGRAD    STATUS=CONVERGED      33 CALLS          34 TOTAL
                    EDM=8.3048e-07    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  mean       -9.40910e-01   3.03997e-02   3.32893e-03  -2.95416e-02
  2  sigma        3.01575e+00   2.22446e-02   2.43807e-03   5.98751e-03
                               ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 9.241e-04 -1.762e-05
-1.762e-05  4.948e-04
PARAMETER  CORRELATION COEFFICIENTS
      NO.  GLOBAL      1      2
       1  0.02606   1.000 -0.026
       2  0.02606  -0.026  1.000
**********
**   18 **HESSE        1000
**********
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=25137.2 FROM HESSE     STATUS=OK             10 CALLS          44 TOTAL
                    EDM=8.30707e-07    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                INTERNAL      INTERNAL
 NO.   NAME      VALUE            ERROR       STEP SIZE       VALUE
  1  mean       -9.40910e-01   3.04002e-02   6.65786e-04  -9.40910e-01
  2  sigma        3.01575e+00   2.22449e-02   9.75228e-05   3.01575e+00
                               ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 9.242e-04 -1.807e-05
-1.807e-05  4.948e-04
PARAMETER  CORRELATION COEFFICIENTS
      NO.  GLOBAL      1      2
       1  0.02672   1.000 -0.027
       2  0.02672  -0.027  1.000
```

The result of the fit – the new parameter values and their errors – is propagated back to the `RooRealVar` objects that represent the parameters of `gauss`, as is demonstrated in the code fragment below:

```
mean.Print() ;
RooRealVar::mean: -0.940910 +/- 0.030400

sigma.Print() ;
RooRealVar::sigma:  3.0158 +/- 0.022245
```

A subsequent drawing of `gauss` will therefore reflect the new shape of the function after the fit. We now draw both the data and the fitted function on a frame,

```
RooPlot* xframe = x.frame() ;
data.plotOn(xframe) ;
model.plotOn(xframe) ;
xframe->Draw()
```

The result of this code fragment is shown in Figure 4.

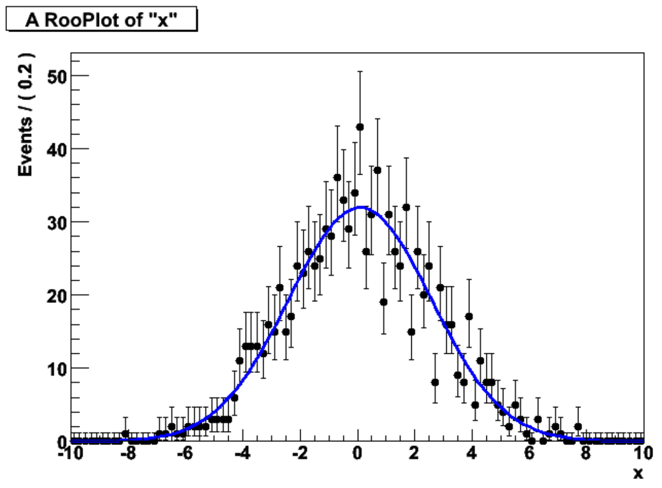


Figure 4 – Output of Example 3.

Note that the normalization of the PDF, which has an intrinsic normalization to unity by definition, is automatically adjusted to the number of events in the plot.

A powerful feature of RooFit and one of the main reasons for its inception is that the fit invocation of Example 3 works for both binned *and unbinned* data. In the latter case an unbinned maximum likelihood fit is performed. Unbinned –log(L) fits are statistically more powerful than binned fits (i.e. you will get smaller errors on averages) and avoid any arbitrariness that is introduced by a choice of bin width. These advantages are most visible when fitting small datasets and fitting multidimensional datasets.

The fitting interface to RooFit is highly customizable and easily customizable. For example, if you want fix a parameter in the fit, you just specify that as a property of the `RooRealVar` parameter object so that this

```
mean.setConstant(kTRUE) ;
gauss.fitTo(data) ;
```

Repeats the fit with parameter `mean` fixed to its present value. Similarly, you can choose to bound a floating parameter to range of allowed values:

```
   sigma.setRange(0.1,3) ;
   gauss.fitTo(data) ;
```

All such fit configuration information is automatically passed to MINUIT. Higher level aspects of MINUIT can be controlled through optional named arguments passed to the `fitTo()` command. This example enables the MINOS method to calculate asymmetric errors and changes the MINUIT verbosity level

```
   gauss.fitTo(data, Minos(kTRUE), PrintLevel(-1)) ;
```

The way the likelihood function is constructed can be influenced the same way. To restrict the likelihood (and thus the fit) to a smaller range of x values do

```
   gauss.fitTo(data, Range(-5,5)) ;
```

A subsequent plot of this fit will then only show a curve in the fitted range (Figure 5).

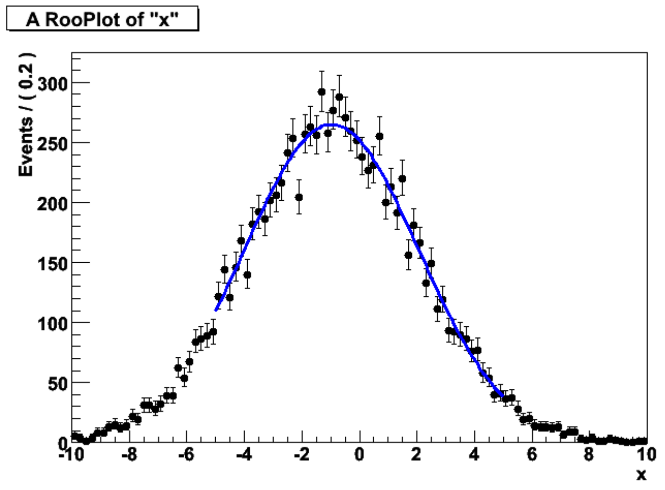


Figure 5 – Fit to a subset of the data

RooFit also supports extended maximum likelihood fits as well as χ^2 fits. These will be covered in the next sections. The complete range of fitting options as well as ways to do interactive fitting are documented in Appendix A.

Generating data from a model

RooFit is not just a fitting tool, it is a full-fledged data modeling tool. This means that you can do more with your models that just fit them to data. An important piece of other functionality is the ability to generate 'toy' Monte Carlo data from your model. Generically this is done through sampling your PDF, but smarter techniques may be used behind the scenes for certain shapes, such as a Gaussian. The most efficient technique is automatically selected for you so you don't have worry about this. In it simplest form you can generate a `RooDataSet` from a pdf as follows:

```
   RooDataSet* data = gauss.generate(x,10000) ;
```

This example create a `RooDataSet` with 10000 events with observable `x` sampled from pdf `gauss`.

Sampling datasets from your PDF is often a useful technique to study the stability of your fit, which may become a concern if your are aiming to fit a small number of events or a large sample with a small number of signal events. Sampling allows you to quickly generate similar but statistically independent datasets on which you can exercise your fit. Section 11 has more details on fit stability studies and techniques to automate them in RooFit.

Putting it all together

At this point we have guided you through various basic procedures in RooFit: defining a model and its variables, importing data, fitting the model to data and generating data from sampling the model. The following macro puts all the knowledge together into a concise exercise that demonstrates all of these abilities and can serve as starting point for your first fitting exercise in RooFit.

```
// Elementary operations on a gaussian PDF
void example(const TH1* histo=0)
{
  // Build Gaussian PDF
  RooRealVar x("x","x",-10,10) ;
  RooRealVar mean("mean","mean of gaussian",0,-100,100) ;
  RooRealVar sigma("sigma","width of gaussian",3,0.,10.) ;
  RooGaussian gauss("gauss","gaussian PDF",x,mean,sigma) ;

  RooAbsData* data = 0 ;
  if (histo) {
    // If a histogram is given import it into a RooDataHist - Binned data
    data = new RooDataHist("data","data",x,histo) ;
  } else {
    // If no histogram is given, generate some toy data - Unbinned data
    data = gauss.generate(x,10000) ;
  }

  // Fit the model to the data
  // Note here that fitTo accepts both binned and unbinned data
  gauss.fitTo(*data) ;

  // Plot PDF and toy data overlaid
  RooPlot* xframe = x.frame() ;
  data->plotOn(xframe) ;
  gauss.plotOn(xframe) ;
  xframe->Draw() ;

  // Print final value of parameters
  mean.Print() ;
  sigma.Print() ;

  // Delete the data
  delete data ;
}
```

In the next section we’ll work towards more realistic models: we will explore composite models – multiple PDFs added together – for example a PDF representing your signal and a PDF representing your background.

3. Signal and Background – Composite models

Introduction

One of the most common data analysis scenarios is that you want to determine the amount of signal and background in a given data sample through a fit. The most straightforward approach to such an analysis is to define a composite p.d.f. $M(x)$ as follows

$$M(x) = f \cdot S(x) + (1-f) \cdot B(x)$$

In this formula $M(x)$ is your fit model, $S(x)$ is your signal model, $B(x)$ is your background model and f is the fraction of your event that are part of the signal. RooFit provide a special 'addition operator' p.d.f. in class `RooAddPdf` to simplify building and using such composite p.d.f.s. A elegant property of adding p.d.f.s in this way is that M(x) does not need to be explicitly normalized to one: if both S(x) and B(x) are normalized to one then M(x) is – by construction – also normalized.

The extended likelihood formalism

Often one is not interested in the fraction of signal events in your sample but in the *number* of signal events in your sample.

$$M_E(x) = N_S \cdot S(x) + N_B \cdot B(x)$$

In this formula $M_E(x)$ is not normalized to 1 but to $N_S+N_B = N$, the number of events in the data sample. A model of this type can be fit with equal ease, but an extra piece of information has to be added to the fit (more specifically: to the likelihood function): the number of events in the data sample. With that extra piece of information the fit can relate the number of events expected by the model ($N_{exp}=N_S+N_B$), to the actual number of events in the data (N_{obs}). The technique that takes care of this extra constraint is called the *extended* maximum likelihood formalism and is described in more detail in Appendix C. All you need to know for now is that RooFit supports both forms of adding p.d.f.s.

Building composite models

Here is a simple example of a composite PDF constructed with `RooAddPdf` using fractional coefficients.

```
RooRealVar x("x","x",-10,10) ;

RooRealVar mean("mean","mean",0,-10,10) ;
RooRealVar sigma("sigma,"sigma",2,0.,10.) ;
RooGaussian sig("sig","signal p.d.f.",x,mean,sigma) ;

RooRealVar c0("c0","coefficient #0", 1.0,-1.,1.) ;
RooRealVar c1("c1","coefficient #1", 0.1,-1.,1.) ;
RooRealVar c2("c2","coefficient #2",-0.1,-1.,1.) ;
RooChebychev bkg("bkg","background p.d.f.",x,RooArgList(c0,c1,c2)) ;

RooRealVar fsig("fsig","signal fraction",0.5,0.,1.) ;

// model(x) = fsig*sig(x) + (1-fsig)*bkg(x)
RooAddPdf model("model","model",RooArgList(sig,bkg),fsig) ;
```

Example 4 – Adding two pdfs using a fraction coefficient

In this example we first construct a Gaussian p.d.f `sig` and flat background p.d.f `bkg` and then add them together with a signal fraction `fsig` in `model`.

Note the use the container class RooArgList to pass a list of objects as a single argument in a function. RooFit has two container classes: RooArgList and RooArgSet. Each can contain any number RooFit value objects, i.e. any object that derives from RooAbsArg such a RooRealVar, RooAbsPdf etc. The distinction is that a *list* is ordered, you can access the elements through a positional reference (2nd, 3rd,…), and can may contain multiple objects with the same name, while a *set* has no order but requires instead each member to have a unique name. You can read more about the properties of RooArgSet and RooArgList in Section 11.

The number of components a RooAddPdf can sum together is not restricted to 2, you can add any arbitrary number of components. Here is an example that adds three p.d.f.s with two coefficients:

```
// model2(x) = fsig*sig(x) + fbkg1*bkg1(x) + (1-fsig-fbkg)*bkg2(x)

RooAddPdf model2("model2","model2",RooArgList(sig,bkg1,bkg2),
                                   RooArgList(fsig,fbkg1)) ;
```

When you build a 'regular' p.d.f, i.e. when you fit for fractions rather than numbers of events, the number of coefficients should always be one less than the number of p.d.f.s.

The extended likelihood formalism

Here is a revision of the first example that uses the extended likelihood formalism, i.e it implements formula (2) rather than formula (1):

```
RooRealVar x("x","x",-10,10) ;

RooRealVar mean("mean","mean",0,-10,10) ;
RooRealVar sigma("sigma","sigma",2,0.,10.) ;
RooGaussian sig("sig","signal p.d.f.",x,mean,sigma) ;

RooRealVar c0("c0","coefficient #0", 1.0,-1.,1.) ;
RooRealVar c1("c1","coefficient #1", 0.1,-1.,1.) ;
RooRealVar c2("c2","coefficient #2",-0.1,-1.,1.) ;
RooChebychev bkg("bkg","background p.d.f.",x,RooArgList(c0,c1,c2)) ;

RooRealVar nsig("nsig","signal fraction",500,0.,10000.) ;
RooRealVar nbkg("nbkg","background fraction",500,0.,10000.) ;

RooAddPdf model("model","model",RooArgList(sig,bkg),RooArgList(nsig,nbkg)) ;

// shape: model(x) = nsig/(nsig+nbkg)*sig(x) + nbkg/(nsig+nbkg)*bkg(x)
// norm:  Nexpect  = nsig + nbkg

// Combined: Nexpect*model(x) = nsig*sig(*x) + nbkg*bkg(x)
```

Example 5 – Adding two pdfs using two event count coefficients

The only difference between Example 4 and Example 5 is that you supply RooAddPdf with an equal number of models and coefficients.

In practical terms the difference between the first and the second example is that in the second form the RooAbsPdf object model is capable of predicting the *expected* number of data events (i.e. nsig+nbkg) through its member function expectedEvents(), while model in the first form cannot. This extra functionality provides the information necessary to construct the extended likelihood.

NB: When you fit extended likelihood models such as Example 5 you should explicitly invoke the construction of extended likelihood term in the fitTo() operation using the Extended() option as will explained shortly in the fitting section

Compose recursively

Note that the input p.d.f.s of `RooAddPdf` do not need to be basic p.d.f.s, they can be composite p.d.f.s themselves. Take a look at this example that uses `sig` and `bkg` from Example 5 as input:

```
RooRealVar mean_bkg(“mean_bkg”,”mean”,0,-10,10) ;
RooRealVar sigma_bkg(“sigma_bkg,”sigma”,2,0.,10.) ;
RooGaussian bkg_peak(“bkg_peak”,”peaking bkg p.d.f.”,x,mean_bkg,sigma_bkg) ;

RooRealVar fpeak(“fpeak”,”peaking background fraction”,0.1,0.,1.) ;
RooRealVar fbkg(”fbkg”,”background fraction”,0.5,0.,1.) ;

RooAddPdf sigpeak(“sigpeak”,”sig + peak”,RooArgList(bkg_peak,sig),fpeak) ;
RooAddPdf model(“model”,”bkg + sigpeak”,RooArgList(bkg,sigpeak),fbkg) ;
```

Example 6 – Adding three p.d.f.s through recursive addition of two terms

The code in this example corresponds to the following formula

$$\begin{aligned} M(x) &= \left[(1-f_1)S(x) \quad + \quad f_1 B(x)\right](1-f_2) \quad + \quad f_2 B_2(x) \\ &= (1-f_1)(1-f_2)S(x) \quad + \quad f_1(1-f_2)B_1(x) \quad + \quad f_2 B_2(x) \end{aligned}$$

Technical Note

Why would you do something like this? First, it might be more intuitive to interpret the fraction parameters this way, but that depends on your specific problem. Second because your fit is more stable this way: If you add two components with a single fraction you can bound the fit to stay in the well defined region by bounding the fraction parameter between 0 and 1. If you have three components and two fraction parameters, the ‘well defined’ region is the region where the *sum* of the fraction parameters is less than one. If you only require each fraction to be inside the range (0,1) you can still end up with a solution where sum is greater than one and consequently the coefficient of the 3rd component is negative. By defining the fractions recursively, as done above, all solutions are well defined as long as each fraction is inside the range (0,1). Another solution is to use the extended likelihood formalism, which sidesteps this problem all together. Also note that if you allow any of the fractions to be exactly zero, the likelihood becomes insensitive to the parameters of the model that is multiplied by this zero fractio. If any of these parameters is floating in the fit, the fit will not converge.

Plotting composite models

The modular structure of a composite p.d.f. allows you to address the individual components. One can for example plot the individual components of a composite model on top of that model to visualize its structure.

```
RooPlot* frame = x.frame() ;
model.plotOn(frame) ;
model.plotOn(frame, Components(bkg),LineStyle(kDashed)) ;
frame->Draw() ;
```

The output of this code fragment is show in Figure 6. You can reference the components by object reference, as is done above, or by name:

```
model.plotOn(frame, Components(“bkg”),LineStyle(kDashed)) ;
```

The latter is convenient when your plotting code has no access to the component objects, for example if your model is built in a separate function that only returns the top-level `RooAddPdf` object.

If you want to draw the sum of multiple components you can do that in two ways as well:

```
model.plotOn(frame, Components(RooArgSet(bkg1,bkg2)),LineStyle(kDashed)) ;

model.plotOn(frame, Components("bkg1,bkg2"),LineStyle(kDashed)) ;
```

Note that in the latter form wildcards are allowed so that a well chosen component naming scheme allows you for example to do this:

```
model.plotOn(frame, Components("bkg*"),LineStyle(kDashed)) ;
```

If required multiple wildcard expressions can be specified in a comma separated list.

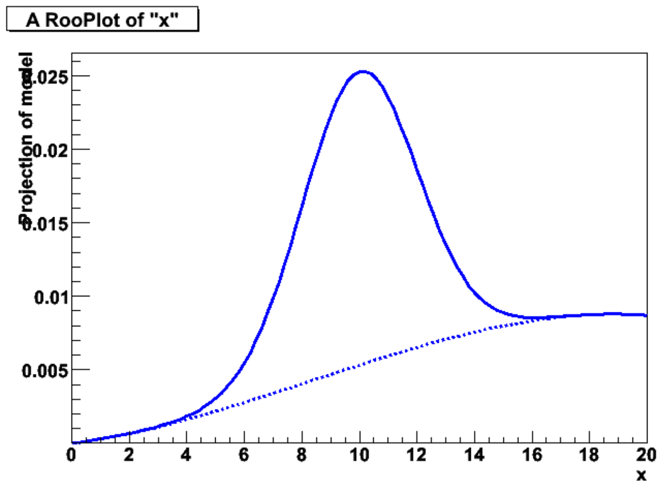


Figure 6 – Drawing of composite model and its components

Fitting composite models

Fitting composite models with fractional coefficients is no different from fitting any other model:

```
model.fitTo(data) ;
```

But fitting models with event count coefficients *is* essentially different: the 'extended likelihood term', the extra piece of the likelihood that constrains the number of events predicted by the model to be equal to the number of observed events in data must be added to the regular likelihood function for the fit to succeed. You do this with the `Extended()` named argument in `fitTo()`:

```
model.fitTo(data,Extended(kTRUE)) ;
```

If you forget to do this, no specific warning message will be issued, but the fit will not converge because there is one degree of freedom that cannot be constrained. This is one of the most common mistakes made in extended likelihood fitting.

Generating data with composite models

Just like you generate toy Monte Carlo data from a simple model you can generate toy data from a composite model:

```
// Generate 10000 events
RooDataSet* x = model.generate(x,10000) ;
```

Sampling data from a composite p.d.f. is often more efficient than sampling data from a monolithic p.d.f. with the same shape as RooFit makes effective use of the component structure of a composite p.d.f.

The extended likelihood formalism

Some extra features apply to composite models built for the extended likelihood formalism. Since these model predict a number events one can omit the requested number of events to be generated

```
RooDataSet* x = model.generate(x) ;
```

In this case the number of events predicted by the p.d.f. is generated. You can optionally request to introduce a Poisson fluctuation in the number of generated events trough the `Extended()` argument:

```
RooDataSet* x = model.generate(x, Extended(kTRUE)) ;
```

This is useful if you generate many samples as part of a study where you look at pull distributions. For pull distributions of event count parameters to be correct, a Poisson fluctuation on the total number of events generated should be present. Fit studies and pull distributions are covered in more detail in section 10.

General tools for dealing with composite objects

The logistics of creating all your p.d.f. components – and keeping tracking of them – become increasing difficult as your p.d.f. grows in complexity. This section describes some of the tools at your disposal to keep this task as easy possible.

An important feature in the design of RooFit is that all important operations – fitting, generating and plotting – can be performed through the top level p.d.f. object. This means that you can delegate the building of a complex p.d.f. to a designated function that just returns a pointer to the top level p.d.f. component, as is illustrated below

```
RooAbsPdf* buildPdf(const RooRealVar& x)
{
  RooRealVar *mean = new RooRealVar (“mean”,”mean”,0,-10,10) ;
  RooRealVar *sigma = new RooRealVar (“sigma,”sigma”,2,0.,10.) ;
  RooGaussian sig = new RooGaussian(“sig”,”signal p.d.f.”,x,*mean,*sigma) ;

  RooRealVar *c0 = new RooRealVar(“c0”,”coefficient #0”, 1.0,-1.,1.) ;
  RooRealVar *c1 = new RooRealVar(“c1”,”coefficient #1”, 0.1,-1.,1.) ;
  RooRealVar *c2 = new RooRealVar(“c2”,”coefficient #2”,-0.1,-1.,1.) ;
  RooChebychev *bkg = new RooChebychev(“bkg”,”backgroundp.d.f.”,
                                    x,RooArgList(*c0,*c1,*c2)) ;
  RooRealVar *fsig = new RooRealVar (“fsig”,”signal fraction”,0.5,0.,1.) ;
  RooAddPdf *model = new RooAddPdf(“model”,”model”,
                                 RooArgList(*sig,*bkg),*fsig);
  return model ;
}

void doTheFit()
```

```
{
  RooRealVar x("x","x",-10,10) ;
  RooAbsPdf* model = buildPdf(x) ;

  RooDataSet* data = model->generate(x,1000) ;
  model->fitTo(*data) ;

  RooPlot* frame = x.frame() ;
  data->plotOn(frame) ;
  model->plotOn(frame) ;
  model->plotOn(frame,Components("bkg")) ;
  frame->Draw() ;
}
```

Example 7 – Building your model in a separate function.
(Memory management issues ignored for the moment for clarity)

While all the big operations clearly work fine this way, it is not obvious how one would for example adjust a parameter value in Example 7 in `doTheFit()`, or print out its value after the fit, so we need some extra tools.

What are the variables of my model?

Given any composite RooFit value object, the `getVariables()` method returns you a `RooArgSet` with all parameters of your model:

```
RooArgSet* params = model->getVariables() ;
params->Print("v") ;
```

This code fragment will output

```
RooArgSet::parameters:
  1) RooRealVar::c0: "coefficient #0"
  2) RooRealVar::c1: "coefficient #1"
  3) RooRealVar::c2: "coefficient #2"
  4) RooRealVar::mean: "mean"
  5) RooRealVar::nbkg: "background fraction"
  6) RooRealVar::nsig: "signal fraction"
  7) RooRealVar::sigma: "sigma"
  8) RooRealVar::x: "x"
```

If you know the name of a variable, you can retrieve a pointer to the object through the `find()` method of `RooArgSet`:

```
RooRealVar* c0 = (RooRealVar*) params->find("c0") ;
c0->setVal(5.3) ;
```

If no object is found in the set with the given name, `find()` returns a null pointer.

Although sets can contain any RooFit value type (i.e. any class derived from `RooAbsArg`) one deals in practice usually with sets of all `RooRealVars`. Therefore class `RooArgSet` is equipped with some special member functions to simplify operations on such sets. The above example can be shortened to

```
params->setRealValue("c0",5.3) ;
```

Similarly, there also exists a member function `getRealValue()`.

What is the structure of my composite model?

In addition to manipulation of the parameters one may also wonder what the structure of a given model is. For an easy visual inspection of the tree structure of a composite object use the method `printCompactTree()`:

```
model.printCompactTree() ;
```

The output will look like this:

```
0x9a76d58 RooAddPdf::model (model)  [Auto]
  0x9a6e698 RooGaussian::sig (signal p.d.f.)  [Auto]
    0x9a190a8 RooRealVar::x (x)
    0x9a20ca0 RooRealVar::mean (mean)
    0x9a3ce10 RooRealVar::sigma (sigma)
  0x9a713c8 RooRealVar::nsig (signal fraction)
  0x9a26cb0 RooChebychev::bkg (background p.d.f.)  [Auto]
    0x9a190a8 RooRealVar::x (x)
    0x9a1c538 RooRealVar::c0 (coefficient #0)
    0x9a774d8 RooRealVar::c1 (coefficient #1)
    0x9a3b670 RooRealVar::c2 (coefficient #2)
  0x9a66c00 RooRealVar::nbkg (background fraction)
```

For each lists object you will see the pointer to the object, following by the class name and object name and finally the object title in parentheses.

A composite object tree is traversed top-down using a depth-first algorithm. With each node traversal the indentation of the printout is increased. This traversal method implies that the same object may appear more than once in this printout if it is referenced in more than one place. See e.g. the multiple reference of observable x in the example above.

Finally we mention the method `getComponents()`, which returns all the ‘branch’ nodes of a composite objects and is complementary to `getVariables()`, which returns the ‘leaf’ nodes. The example below illustrates the use of `getComponents()` to only print out the variables of model component “sig”:

```
RooArgSet* comps = model.getComponents() ;
RooAbsArg* sig = comps->find(“sig”) ;
RooArgSet* sigVars = sig->getVariables() ;
sigVars->Print() ;
```

Note that the output of most operations is of type `RooAbsArg`, the abstract value type in RooFit. Since the tree structure inspection functions are not specific to real-valued positive-definite probability density functions, we can perform all operations with these `RooAbsArg*` abstract value type pointers. The output of this example is

```
RooArgSet::parameters:
  1) RooRealVar::mean: "mean"
  2) RooRealVar::sigma: "sigma"
  3) RooRealVar::x: "x"
```

In section 10 will we go into more detail on this subject.

Putting it all together

In this section you have learned how to add basic p.d.f.s together into a composite p.d.f.s. Adding p.d.f.s can be done in one of two ways: you can add N p.d.f.s with N-1 fractions, or your can N p.d.f.s together with N event counts. The latter form involves the extended likelihood formalism and implies that you fit for the number of events in data as well as the shape of the data. Generating, fitting and plotting composite p.d.f.s is identical to generating, fitting and plotting basic p.d.f.s., except for occasional extra functionality, such as the ability to plot components of a composite p.d.f.

4. Choosing & adjusting standard p.d.f. components

We will now have a closer look at what p.d.f.s are provided with RooFit, how you can tailor them to your specific problem and how you can write a new p.d.f.s in case none of the stock p.d.f.s. have the shape you need.

What p.d.f.s are provided?

RooFit provides a library of about 20 probability density functions that can be used as building block for your model. These functions include basic functions, non-parametric functions, physics-inspired functions and specialized decay functions for B physics.

Basic functions

The most frequently used basic shapes, the Gaussian, exponential and polynomial functions are all implemented in RooFit. Their shapes are illustrated in Figure 7

Name	Functional form	Class name
Gaussian	$\exp\left(-0.5\left(\frac{x-m}{s}\right)^2\right)$	`RooGaussian(name,title,x,m,s)`
Exponential	$\exp(a \cdot x)$	`RooExponential(name,title,x,a)`
Polynomial	$1+\sum_{i=1,n} a_i x^i$	`RooPolynomial(name,title,x,alist)`
Chebychev polynomial	$1+\sum_{i=1,n} a_i T_i(x)$	`RooChebychev(name,title,x,alist)`

Table 1 – Basic functions implemented in RooFit

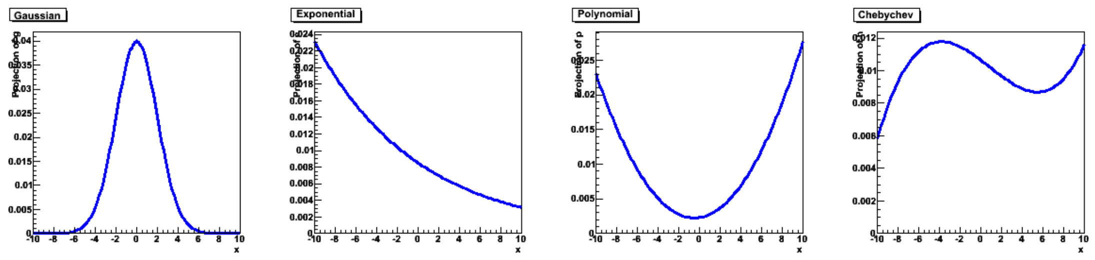


Figure 7 – Basic p.d.f shapes: Gaussian, Exponential, Polynomial and Chebychev polynomial

Note that each functional form in Table 1 has one parameter less than usual form because the degree of freedom that controls the vertical scale is eliminated by the constraint that the integral of the p.d.f. is exactly 1. The formula listed in the table are not normalized to unity for presentation clarity, but each `RooAbsPdf`-based p.d.f. is internally multiplied by the (analytical) integral of the listed expression to achieve unit normalization.

Practical Tip

We recommend the use of Chebychev polynomials over regular polynomials because of their superior stability in fits. Chebychev polynomials and regular polynomials can describe the same shapes, but a clever reorganization of power terms in Chebychev polynomials results in much lower correlations between the coefficients a_i in a fit, and thus to a more stable fit behavior. For a definition of the functions T_i and some background reading, look e.g. at `http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html`

Physics inspired functions

In addition to the basic shapes RooFit also implements a series of shapes that are commonly used to model physical 'signal' distributions.

The Landau function parameterizes energy loss in material and has no analytical form. RooFit uses the parameterized implementation in `TMath::Landau`.

The Argus function is an empirical formula to model the phase space of multi-body decays near threshold and is frequently used in B physics.

The non-relativistic Breit-Wigner shape models resonance shapes and its cousin the Voigtian – a Breit-Wigner convolved with a Gaussian --- are commonly used to describe the shape of a resonance in the present of finite detector resolution.

The Crystal ball function is a Gaussian with a tail on the low side that is traditionally used to describe the effect of radiative energy loss in an invariant mass.

The decay function differs from the exponential p.d.f in that it can also chosen to be symmetric around 0 and can be convolved analytically with a selection of resolution models.

Their shapes are illustrated in Figure 8.

Name	Functional form	Class name
Landau	`TMath::Landau(x,mean,sigma)`	`RooLandau(name,title,x,mean,sigma)`
Argus	$x\left(1-\left(\frac{x}{m}\right)^2\right)^p \cdot \exp\left(c\left(1-\left(\frac{x}{m}\right)^2\right)\right)$	`RooArgusBG(name,title,x,m,c,p)`
Breit-Wigner	$\frac{1}{(x-m)^2+\frac{1}{4}g^2}$	`RooBreigWigner(name,title,x,m,g)`
Voigtian	$\frac{1}{(x-m)^2+\frac{1}{4}g^2}\otimes\exp\left(-\frac{1}{2}\left(\frac{x}{s}\right)^2\right)$	`RooVoigtian(name,title,x,m,g,s)`
Crystal Ball	$\left.\frac{\left(\frac{n}{\lvert a\rvert}\right)^n e^{-\frac{1}{2}a^2}}{\left(\frac{n}{\lvert a\rvert}-\lvert a\rvert-x\right)^n}\right\rvert_{x<-\lvert a\rvert}, \quad \left.\exp\left(-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right)\right\rvert_{x>-\lvert a\rvert}$	`RooCBShape(name,title,x,m,s,a,n)`
Decay	$\exp(-\lvert x\rvert/\tau)\otimes R(x)$	`RooDecay(name,title,x,tau,R)`

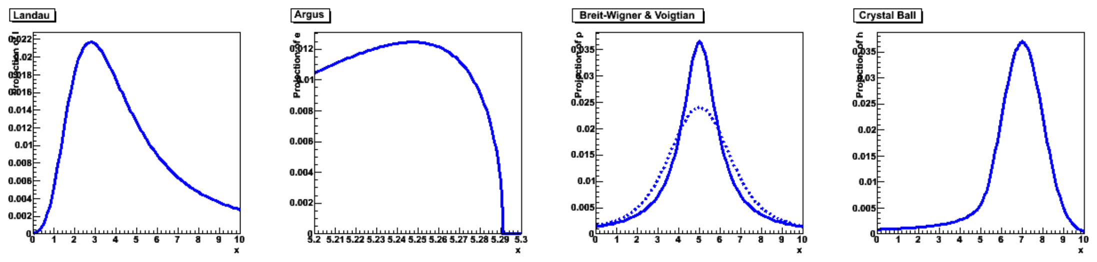


Figure 8 – Physics inspired p.d.f.s: Landau, Argus, Breit-Wigner (Voigtian) and Crystal Ball.

Non-parametric functions

RooFit offers two solutions for modeling distributions that cannot easily be parametrized.

Class `RooHistPdf` takes an input histogram in the form of a `RooDataHist` and represents its shape as a probability density. The histogram contents is explicitly scaled to obtain proper normalization. Optionally, the histogram is interpolated (up to 9th order).

Class `RooKeysPdf` is a more elaborate approach to obtain best possible continuous probability density function that aims to describe the parent distribution of an unbinned `RooDataSet`. The idea behind the KEYS algorithm, documented here[3], is that the p.d.f. is constructed as a superposition of Gaussians. Each of the events in the input data becomes a Gaussian contribution to the p.d.f. with total weight of 1/*N* centered at the *x* value of the data point. The width of the Gaussian is adjusted the local density of events: regions with a low local density of events have a Gaussian with a large width to ensure a smooth function in sparsely populated regions. Areas with a high density of events have Gaussians with a narrow width to preserve details in the structure. Read the original article for further information.

Both classes are shown in Figure 9.

Name	Functional form	Class name
Histogram	Histogram, with optional interpolation	`RooHistPdf(name,title,x,hist,intorder)`
Empirical density estimate	Superposition of many Gaussians	`RooKeysPdf(name,title,x,data)`

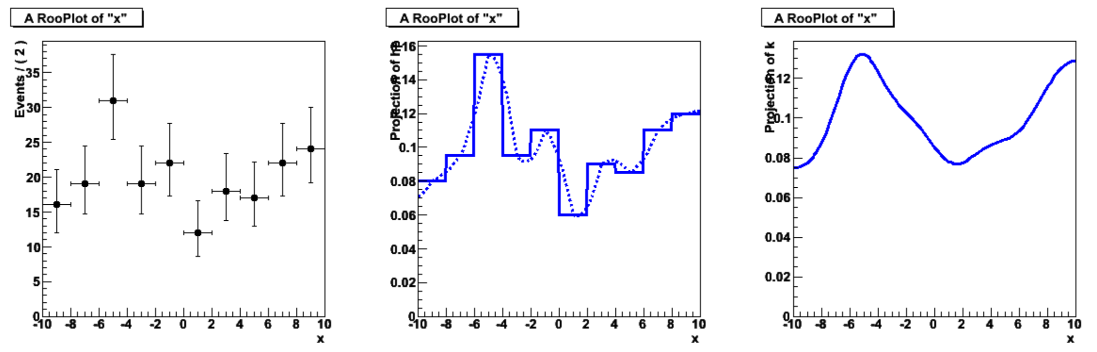


Figure 9 – Non-parametric p.d.f.s: Left: histogram of unbinned input data, Middle: Histogram-based p.d.f (2nd order interpolation), Right: KEYS p.d.f from original unbinned input data.

Specialized functions for B physics

RooFit was originally development for BaBar, the B-factory experiment at SLAC, therefore it also provides a series of specialized B physics p.d.f.s. We will list them here for completeness. A complete description is beyond the scope of this document.

Class Name	Description
`RooBMixDecay`	B decay with mixing
`RooBCPEffDecay`	B decay with CP violation parameterized as sin(2b) and |l|
`RooBCPGenDecay`	B decay with CP violation parameterized S and C
`RooNonCPEigenDecay`	B decay to non-CP eigenstates with CP violation
`RooBDecay`	Generic B decay with mixing, CP violation, CPT violation

[3] 'Kernel Estimation in High-Energy Physics', *K. Cranmer,* Comput.Phys.Commun. 136 (2001) 198-207, hep-ex/0011057

Plug and play with parameters

You are not stuck with the parameterization of the stock p.d.f.s. that we have chosen. A key feature of the design of RooFit functions and p.d.f.s. is that there is no hard-wired assumption that the parameters of a function are *variables* (i.e. a `RooRealVar`), so you can modify the parameterization of *any* existing p.d.f. by substituting a *function* for a parameter. The following example illustrates this:

```
RooRealVar x(“x”,”x”,-10,10) ;

RooRealVar mean(“mean”,”mean”,0,-10,10) ;
RooRealVar sigma(“sigma_core”,”sigma (core)”,1,0.,10.) ;
RooGaussian sig_left(“sig_left”,”signal p.d.f.”,x,mean,sigma) ;

RooRealVar shift(“shift”,”shift”,1.0) ;
RooFormulaVar mean_shifted(“mean_shifte”,”mean+shift”,RooArgSet(mean,shift));
RooGaussian sig_right(“sig_right”,”signal p.d.f.”,x,mean_shifted,sigma) ;

RooRealVar frac_left(“frac_left”,”fraction (left)”,0.7,0.,1.) ;
RooAddPdf sig(“sig”,”signal”,RooArgList(sig_left,sig_right),frac_left) ;
```

The p.d.f. `sig` is a sum of two Gaussians in which the position of one Gaussian is shifted by `shift` with respect to the other one. The mean of the second Gaussian is not specified through a `RooRealVar` parameter however, but through a `RooFormulaVar` function objects, which relates the position of the second Gaussian to that of the first Gaussian.

The function that calculates the position of the rightmost Gaussian is an object of type `RooFormulaVar`, which is a real-valued function that evaluates itself by interpreting the formula expression `mean+shift` using ROOTs `TFormula` engine.

While the functional form of the two-Gaussian p.d.f. `sig` is no different from one constructed of two ordinary Gaussian, each with their own mean, the ability to reparametrize the model like this is that one can now for example fit with a floating mean while keeping the distance between the Gaussians fixed. Figure 10 shows the `sig` p.d.f. of the above example for `mean=-3`, `mean=3` and `shift=3`,`shift=6` in red and blue respectively.

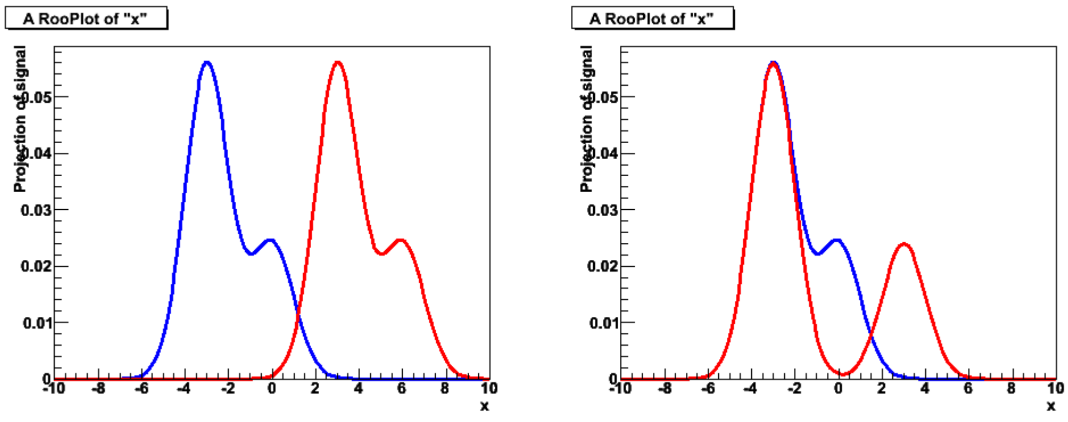


Figure 10 – left: variation of `mean` variable, right: variation of `shift` variable

Class `RooFormulaVar` can handle any C++ expression that ROOT class `TFormula` can. This includes most math operators (+,-,/,*,…), nested parentheses and some basic math and trigonometry functions like `sin`, `cos`, `log`, `abs` etc…Consult the ROOT `TFormula` documentation for a complete overview of the functionality. The names of the variables in the formula expression are those of the variables given in the `RooArgSet` as 3rd parameter in the constructor. Alternatively, you can reference the variable through positional index if you pass the variables in a `RooArgList`:

```
RooFormulaVar mean_shifted("mean_shifte","@0+@1",RooArgList(mean,shift));
```

This form is usually easier if you follow a 'factory-style' approach in your own code where you don't know (or don't care to know) the names of the variables you intend to add in code that declares the `RooFormulaVar`.

Class `RooFormulaVar` is explicitly intended for trivial transformations like the one shown above. If you need a more complex transformation you should write a compiled class. The last paragraph of this section shows how you can easily write compilable classes.

Revisiting the addition of three p.d.f.s.

In the previous section we rewrote the addition of three p.d.f.s. with two fraction coefficients as a recursive addition of two terms to be able to define two fraction parameters that each have a valid range between 0 and 1. The example below accomplishes the same functional form using a single `RooAddPdf` and a customized coefficient implemented with a `RooFormulaVar`:

```
// M(x) = (1-fp)(1-fb)·S(x) + fp(1-fb)·B1(x) + fb.B2(x)

RooFormulaVar fracPeak("fracPeak","fpeak*(1-fbkg)",RooArgSet(fpeak,fbkg)) ;
RooAddPdf model("model","bkg + sig + peak",
                RooArgList(bkg,bkg_peak,sig),RooArgList(fbkg,fracPeak)) ;
```

Which form is better – recursive `RooAddPdf` or using a `RooFormulaVar` – depends on your specific application. The point here is to demonstrate that you can achieve flexibility in your p.d.f. in more than one way.

Writing a new p.d.f. class

If none of the existing p.d.f. classes suit your needs, and no one can be customized through use of `RooFormulaVar`, you can write your own RooFit p.d.f. class. If the formula expression of your model is relatively simple, and performance is not critical you, can use `RooGenericPdf` which interprets your C++ expression, just like `RooFormulaVar`:

```
RooRealVar x("x","x",-10,10) ;
RooRealVar alpha("alpha","alpha",1.0,0.,10.) ;
RooGenericPdf g("g","sqrt(abs(alpha*x))+0.1",RooArgSet(x,alpha)) ;

RooPlot* frame = x.frame() ;
g.plotOn(frame) ;
alpha=1e-4 ;
g.plotOn(frame,LineColor(kRed)) ;
frame->Draw() ;
```

The formula expression entered into `g` is explicitly normalized through numeric integration before it is returned as the value of p.d.f `g`, so you never have to worry about normalization yourself. The automatic normalization is nicely demonstrated in Figure 11, which shows p.d.f. `g` for two values of parameter `alpha`. If your formula expression becomes more complicated than the example shown above, you should write a compiled class that implements your function.

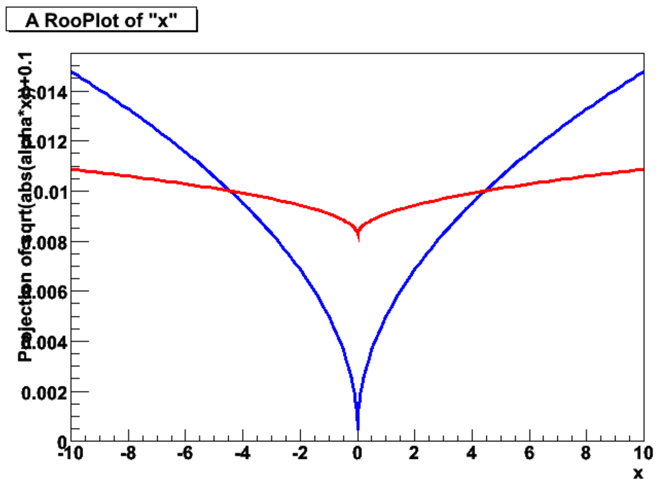


Figure 11 – Generic p.d.f `g("sqrt(abs(x*alpha))+0.1")` drawn for alpha=1 (blue) and alpha=0.0001 (red)

Writing a new p.d.f class using `RooClassFactory`

A special utility class `RooClassFactory` greatly simplifies the task of writing a RooFit p.d.f class that is an implementation of `RooAbsPdf`. The class factory writes a complete working skeleton class for you with the name you specify and with the variable names you specify. Here is an example:

```
RooClassFactory::makePdf("RooMyPdf","x,alpha") ;
```

This example invocation of `makePdf` creates two files: `RooMyPdf.cxx` and `RooMyPdf.h`. The only piece that is missing is actual function expression in terms of the variables you defined. To do so edit the file `RooMyPdf.cxx` and insert the function expression as return value of the evaluate() method of your class.

```
Double_t RooMyPdf::evaluate() const
 {
   // ENTER EXPRESSION IN TERMS OF VARIABLE ARGUMENTS HERE
   return sqrt(abs(alpha*x))+1 ;
 }
```

You are now ready to use your new class: compile the class using ROOTs ACLiC facility

```
root>.L RooMyPdf.cxx++
```

Here is the original example rewritten in terms of your new compiled class `RooMyPdf`:

```
RooRealVar x("x","x",-10,10) ;
RooRealVar alpha("alpha","alpha",1.0,0.,10.) ;
RooMyPdf g("g","compiled class g",x,alpha) ;
```

Classes that are created through `RooClassFactory` have an explicit (numeric) normalization step built in, i.e. the return value of `evaluate()` does not have to be a properly normalized expression. This is done for your convenience, but carries a (small) performance penalty.

If you know how to normalize your expression analytically, you can indicate that in your `RooClassFactory` invocation and a slightly different skeleton class is built for you that allows to implement the analytical normalization as well. You can find more details in the `RooClassFactory` HTML class documentation.

Writing a new function class using `RooClassFactory`

The code factory class `RooClassFactory` cannot only write skeleton p.d.f.s, but also skeletons for generic real-valued functions. Generic real-valued function are all classes in RooFit that inherit from `RooAbsReal`. Class `RooFormulaVar` is a good example of a generic real-valued function. Unlike p.d.f.s, `RooAbsReal` are not normalized to unity and can also take negative values.

Compilable custom real-valued functions are a good replacement for `RooFormulaVar` in cases where the formula expression is less than trivial, or in cases where performance is critical.

Creating a skeleton for a generic function object is done with the `makeFunction()` method of `RooClassFactory`

```
RooClassFactory::makeFunction("RooMyFunction","x,b") ;
```

5. Convolving a p.d.f. or function with another p.d.f.

Introduction

If you are modeling distribution of an experimental observable you are sometimes faced with a situation where you should explicitly take into account the deformation of the expected signal distributed due to the finite detector resolution. This issue becomes particularly important when the detector resolution is comparable to the structure (width) of your expected signal. The technical aspects combining the effects of detector resolution and your physics model tend to be complicated, which is why we have a separate chapter on this subject.

In general, the observed distribution is described by the convolution of your physics model *T*(*x*,*a*) and your detector response function *R*(*x*,*b*)

$$M(x,a,b) = T(x,a) \otimes R(x,b) = \int_{-\infty}^{+\infty} T(x,a) R(x-x',b)\,dx'$$

In practice the detector response function *R* is often a Gaussian, or a superposition of Gaussians. Figure 12 Illustrates the effect of a Gaussian resolution model R with three different widths on a Breit-Wigner function.

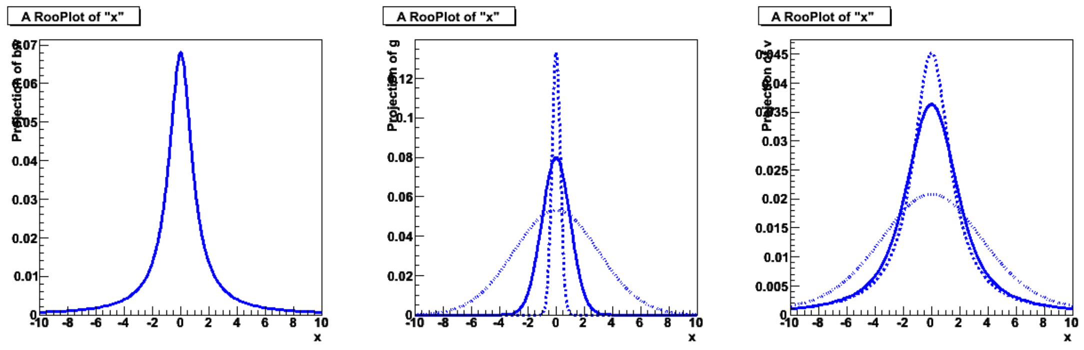


Figure 12 – left: Breit-Wigner, middle Gaussian (σ=0.3, 1, 3) right: Breit-Wigner convolved with Gaussian

You can see from Figure 12 that if R is narrow with respect to T (dotted line), the convolution T⊗R is well approximated by T. If R is wide with respect to T (dashed line), the convolution T⊗R is well approximated by R, therefore modeling your signal p.d.f. explicitly as T⊗R is usually only important if both are comparable width. This is a good thing, since calculation of integral that represents T⊗R is generically quite difficult. The normalization condition for p.d.f.s. adds one further difficulty as the final quantity acquires a double integral in the denominator.

$$M(x,a,b) = \frac{\int_{-\infty}^{\infty} T(x,a) R(x-x',b)\,dx'}{\int_{x_{\min}}^{x_{\max}} \int_{-\infty}^{\infty} T(x,a) R(x-x',b)\,dx'dx}$$

You are best off if you don't need to perform this calculation, but sometimes you just have to. In the remainder of this section we'll explain how you can deal with convolved p.d.f.s in RooFit.

Analytical versus numeric convolution

A precise and fast calculation of the convolution integral is essential as p.d.f.s are evaluated a large number of times in the course of a fit. Because of that an analytical expression for the convolution integral is therefore strongly preferred. Unfortunately this is not always possible, and a numeric calculation of the integral must sometimes be used as fallback solution.

Analytical convolution

RooFit does not find analytical expressions for convolution integrals for you, but helps you to implement them in a generic and reusable way. It does this by defining two specialize sub-classes of p.d.f.s.: convolvable p.d.f.s, which implement $T(x,a)$ and resolution models, which implement $R(x,b)$. You can combine any R and T at *runtime* into a M(x,a,b)=T(x,a) ⊗ R(x,b) so you are quite flexible in your choice of convolutions when you build your model. RooFit provides the following convolvable p.d.f.s out of the box:

Class Name	Description
`RooDecay`	Decay function: exp(-|*t*|/τ), exp(-t/τ) or exp(t/τ)
`RooBMixDecay`	B decay with mixing
`RooBCPEffDecay`	B decay with CP violation parameterized as sin(2b) and |l|
`RooBCPGenDecay`	B decay with CP violation parameterized S and C
`RooNonCPEigenDecay`	B decay to non-CP eigenstates with CP violation
`RooBDecay`	Generic B decay with possible mixing, CP violation, CPT violation

And it provides the following resolution models.

Name	Functional form	Class name
Gauss	$\exp\left(-0.5\left(\frac{x-m}{s}\right)^2\right)$	`RooGaussModel(name,title,x,m,s)`
Gauss⊗Exp	$\exp\left(-0.5\left(\frac{x-m}{s}\right)^2\right)\otimes\exp(-x/\tau)$	`RooGExpModel(name,title,x,m,s,tau)`
Truth	$\delta(x)$	`RooTruthModel(name,title,x)`
Composite	$\sum_{i=1,n-1} f_i R_i(x,\alpha)+\left(1-\sum_{i=1,n-1} f_i\right)R_n(x,\alpha)$	`RooAddModel(name,title,Rlist,flist)`

To construct an analytically convolved p.d.f. pass one of the `RooResolutionModel` implementations to the construct of a convolvable p.d.f. In the example below we construct a decay function convolved with a Gaussian resolution model:

```
RooRealVar x("x","x",-10,10) ;

RooRealVar mean("mean","mean",0) ;
RooRealVar sigma("sigma","sigma",1) ;
RooGaussModel gaussm("gaussm",x,mean,sigma) ;

RooRealVar tau("tau","lifetime",1.54) ;
RooDecay model("model","decay (x) gauss",x,tau,gaussm) ;
```

```
// --- Plot decay (x) gauss ---
RooPlot* frame = x.frame() ;
model.plotOn(frame) ;

// --- Overlay with decay (x) truth ---
RooTruthModel truthm("truthm","truth model",x) ;
RooDecay modelt("modelt","decay (x) delta",x,tau,truthm) ;
modelt.plotOn(frame,LineStyle(kDashed)) ;

frame->Draw() ;
```

Figure 133 shows the output of this example.

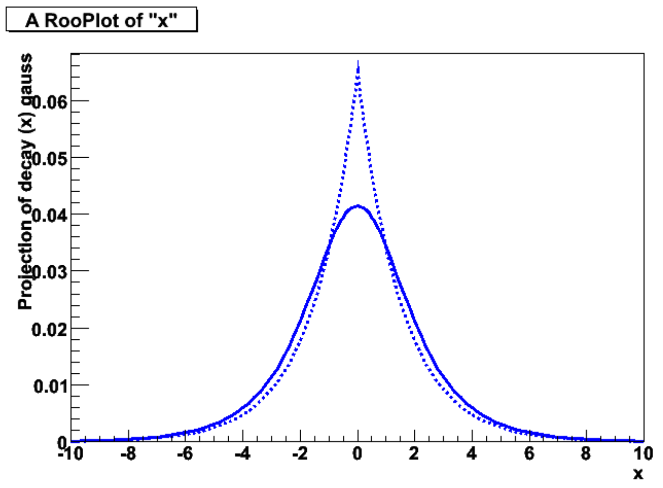


Figure 13 – Decay p.d.f convolved with Gaussian and delta function (dashed)

A realistic detector resolution is often more complex that a simple Gaussian. Class RooAddModel allows you to add multiple resolution models into a single composite resolution model that can be passed to any convolvable p.d.f. Here is an example using RooAddModel to construct a decay function convolved with a double Gaussian resolution.

```
RooRealVar x("x","x",-10,10) ;

RooRealVar mean("mean","mean",0) ;
RooRealVar sigma_core("sigma_core","sigma core",1) ;
RooGaussModel gaussm_core("gaussm_core","core gauss",x,mean,sigma_core) ;

RooRealVar sigma_tail("sigma_tail","sigma tail",5) ;
RooGaussModel gaussm_tail("gaussm_tail","tail gauss",x,mean,sigma_tail) ;

RooRealVar frac_core("frac_core","core fraction",0.9) ;
RooAddModel gaussm("gaussm","core+tail gauss",
         RooArgList(gaussm_core,gaussm_tail),frac_core) ;

RooRealVar tau("tau","lifetime",1.54) ;
RooDecay model("model","decay (x) gauss",x,tau,gaussm);
```

Class `RooAddModel` works very similar to class `RooAddPdf` with the restriction that you can only specify fraction coefficients and not event yield coefficients as the extended likelihood formalism doesn't apply to resolution models.

Technical Note

How do classes like RooDecay and RooGaussModel divide the work when it comes to performing the analytical integration? First thing to know is that RooAbsAnaConvPdf, the abstract base class for analytically convolvable p.d.f.s decomposes the p.d.f. as follows

$$M(x,a) = \sum_i c_i b_i(x)$$

In this formula $b_i(x)$ are so-called 'basis functions' and are the common language between a convolvable p.d.f and a resolution model. A resolution model like class RooGaussModel inherits from class RooResolutionModel and implements member functions that advertise if the resolution model can convolve itself with a given $b(x)$. If the resolution model can convolve itself with all of the basis functions $b_i(x)$ of the p.d.f, the return value of the convolved p.d.f. can the calculated as:

$$M(x,a) = \sum_i c_i \left[b_i(x) \otimes R(x) \right]$$

The calculation of the part in the square bracket is delegated to the resolution model object. One of the advantages of this decomposition approach is speed: if parameter model M changes that only affects coefficients c_i, the convolution integral does not need to be recalculated. Appendix C has additional technical details on the structure and inheritance of classes RooAbsAnaConvPdf and RooResolutionModel and their interaction.

Numeric convolution

If the convolution of your choice is not available in analytical form, we suggest you first try to calculate it yourself. If an analytical solution exists for your convolution there is a good chance that Wolframs Mathematica can calculate it for you. Try the free web interface on `http://integrals.wolfram.com`

Numeric integration is computationally intensive as a precision of O(10^{-6}) needs to be reached for the numeric noise not disturb MINUIT in its likelihood minimum finding. In practice this means O(100) evaluations of *R* and *T* to calculate *M* for each data point. Numeric convolution is implemented in class RooNumConvPdf. This class follows the 'operator' formalism: you specify two input p.d.f.s in its construction and its own value is the convolution of the two. Here is an example on how to use RooNumConvPdf.

```
RooRealVar x("x","x",-10,10) ;

RooRealVar meanl("meanl","mean of Landau",2) ;
RooRealVar sigmal("sigmal","sigma of Landau",1) ;
RooLandau landau("landau","landau",x,meanl,sigmal) ;

RooRealVar meang("meang","mean of Gaussian",0) ;
RooRealVar sigmag("sigmag","sigma of Gaussian",2) ;
RooGaussian gauss("gauss","gauss",x,meang,sigmag) ;

RooNumConvPdf model("model","model",x,landau,gauss) ;

RooPlot* frame = x.frame() ;
model.plotOn(frame) ;
landau.plotOn(frame,LineStyle(kDashed)) ;
frame->Draw() ;
```

Example 8 – Numeric convolution of a Landau with a Gaussian

Figure 14 show the result of Example 8.

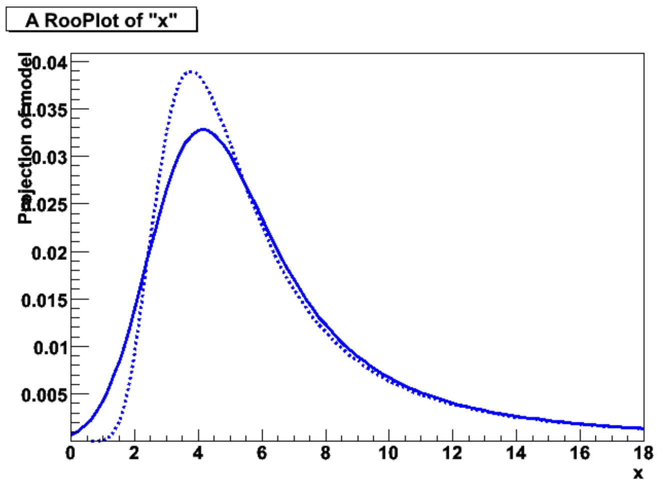


Figure 14 – Output of Example 8 – Numeric convolution of a Landau with a Gaussian, Landau convolved with a Gaussian and the original Landau (dashed line)

Configuring the numeric convolution integration

By default `RooNumConvPdf` performs the numeric convolution integral on the full domain of the convolution variable (i.e. from -∞ to +∞) using a x → 1/x transformation to calculate the integrals of the tails extending to infinity. This calculation is difficult, can suffer from stability problems and may be avoided for certain choices of resolution models. For certain resolution models, e.g. a Gaussian, you know *a priori* that the integrand of the convolution integral is effectively zero when you are far from the core of the resolution model. For such cases one can restrict the domain of the convolution integral to e.g. [$-5\sigma+\mu$,$+5\sigma+\mu$], where μ and σ are the mean and width of the Gaussian resolution model respectively. `RooNumConvPdf` offers you the option restrict the convolution domain along these lines:

```
landau.setConvolutionWindow(meang,sigmag,5)
```

The optional 3rd parameter of `setConvolutionWindow` serves as a multiplier of the width parameter and exists for solely convenience as it saves you a `RooFormulaVar`: the above example restricts the integration domain to [5*`sigmag`+`meang`,-5*`sigmag`+`meang`].

Adjusting numeric integration precision and technique.

If you are going to fit models based on numeric convolutions it is almost inevitable that you will need to fine tune the numeric integration parameters to obtain the right balance between speed and precision. You can access the numeric integration configuration object that is used for the convolution integral from member function `convIntConfig()`. You can read more about numeric integration configuration in section 11.

Numeric convolution is an intrinsically difficult problem. You should expect to spend some time tuning the integration configuration before you obtain a workable configuration (if it is at all possible).

6. Using many observables – Multidimensional models

Many data analysis problems deal with more than one observable. Devising a strategy on how to deal with all this information is a central aspect of such analysis. A common strategy is to make a preselection of your data sample using all but one of your observables. Such a preselection can anything ranging from a cut on each of the individual variables to a neural network consolidating the information of many variables into a single observable followed by a cut on that observable. A fit to the distribution of the remaining observable will then determine the number of signal and background events in your sample after preselection and determine the signal properties.

Another – more ambitious – strategy is to use many (or all) of the observables directly in a fit. This has the advantage that the information that is contained in each observable and in the correlation between the observables are optimally exploited and are exploited in a sensible and understood way. This extra power comes at the cost of some additional complications you have to deal with and come in, roughly speaking, two categories: fundamental modeling issues and practicalities. Practical issues usually revolve around your ability to manage the increased complexity of the model and how to visualize multi-dimensional models in an intuitive way. Fundamental issues include your finite ability to truly correctly understand and describe a multidimensional signal and background distribution including all possible correlations between variables.

RooFit has been designed to make working with multi-dimensional models as easy as working with one-dimensional models. Multi-dimensional models have a lot of extra functionality, but working with them is not more cumbersome than working with one-dimensional models. This enables you to design a more ambitious data analysis where you can get the most out of your data. Multidimensional modeling is not an all-or-nothing issue: you can for example combine the information of several variables into a neural network and feed the output of the network along with the remaining observables into a multi-dimensional fit. In this example approach you also let the events that are classified as less-probable by your neural net participate in your analysis and squeeze out some extra statistical power.

This section focuses on the practical aspects of building and using multi-dimensional models at any point in your analysis.

Building and using multi-dimensional models

A multi-dimensional p.d.f. is a model with more than one observable, but is in all other respects identical to the one-dimensional modes we have covered so far. The normalization condition for multi-dimensional p.d.f.s is identical to that of one-dimensional p.d.f. except that the normalization integral is now performed over all observables:

$$\int F(\vec{x};\vec{p})d\vec{x} = 1$$

Here is a very simple example of a two-dimensional p.d.f. constructed with RooFit using `RooGenericPdf`:

```
RooRealVar x(“x”,”x”,-10,10) ;
RooRealVar y(“y”,”y”,-10,10) ;

RooRealVar a(“a”,”a”,5) ;
RooRealVar b(“b”,”b”,2) ;

RooGenericPdf f(“f”,”a*x*x+b*y*y-0.3*y*y*y”,RooArgSet(x,y,a,b)) ;
```

Example 9 – A simple two-dimensional p.d.f.

The model f of this example is a ‘monolithic’ two-dimensional p.d.f, it does cannot be factorized as product of two or more simpler p.d.f.s.

```
// Generate a 2-dimensional dataset data(x,y) from gaussxy
RooDataSet* data = f.generate(RooArgSet(x,y),10000) ;

// Fit the 2-dimensional model f(x,y) to data(x,y)
f.fitTo(*data) ;

// Plot the x distribution of data(x,y) and f(x,y)
RooPlot* framex = x.frame() ;
data->plotOn(framex) ;
f.plotOn(framex) ;

// Plot the y distribution of data(x,y) and f(x,y)
RooPlot* framey = y.frame() ;
data->plotOn(framey) ;
f.plotOn(framey) ;

// Draw the x and y frames on a canvas
TCanvas *c = new TCanvas(“c”,”c”,800,400) ;
c->Divide(2) ;
c->cd(1) ; framex->Draw() ;
c->cd(2) ; framey->Draw() ;
```

Example 10 – A two-dimensional p.d.f. constructed with RooProdPdf

There are several points worth noting about this example. First, generating data works *exactly* the same for multidimensional p.d.f.s as for one-dimensional datasets, simply supply a RooArgSet of observables instead of single observable as the first argument of generate. Second, fitting also works exactly the same. Lastly, plotting is essentially the same, but have more options now. In the example we created two one-dimensional views: a view in x (framex) and a view in y (framey). Once each view is defined all goes automatic: the two-dimensional dataset data(x,y) plots the appropriate observable in each frame and the two-dimensional p.d.f. fxy(x,y) plots the appropriate projection on the frame. The output of Example 10 is shown in Figure 16.

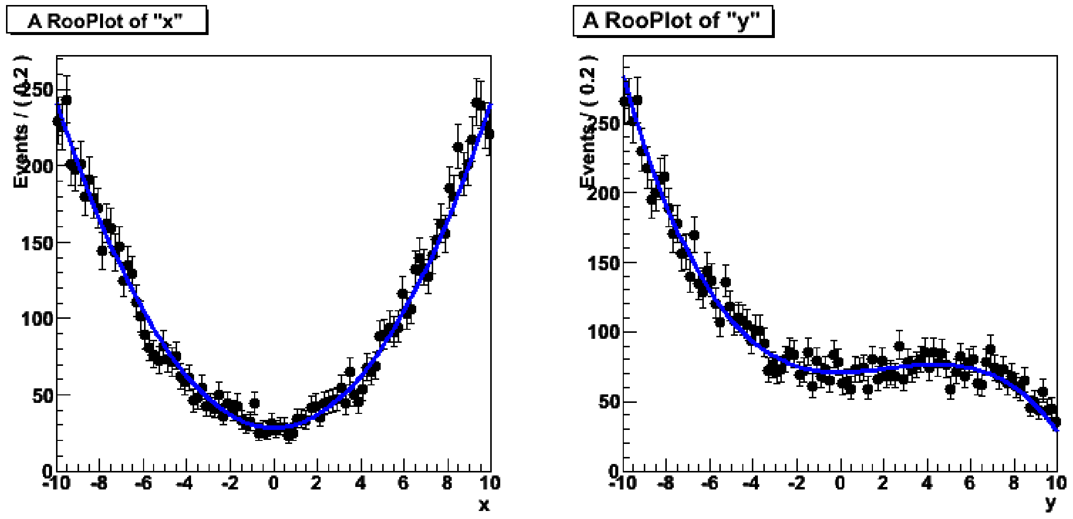


Figure 15 – The x and y projection of p.d.f. f from Example 9.

A bit more on plotting multi-dimensional p.d.f.s

The fact that the two plots of Example 10 come out as expected is not entirely trivial and reflect some bookkeeping that RooFit does for you in the background. Plotting the data is easy: to obtain the x distribution of `data(x,y)` you should ignore the y values and fill a histogram with the x values. Plotting a p.d.f. involves a bit more thinking: the x distribution of `gaussxy(x,y)` is different for each value of y, so we cannot simply plot `gauss(x,y)` as function of x for a given value of y: we should plot something that matches the x distribution of the data for the given distribution of y values that are in the data. To obtain that shape you need to *integrate* the p.d.f. over y:

$$F_x(x;\vec{p}) = \int F(x,\vec{y};\vec{p})d\vec{y}$$

A nice feature of RooFit is that you almost never need to worry about performing such integrals as RooFit keeps track of all the ‘projected’ observables in any plot that you make. For example, when we plotted data on `xframe` (‘`data->plotOn(framex)`’) not only a histogram representing the distribution in x of data was added to `xframe`, but also a list of all observables that were stored in data, in this case `(x,y)`. The subsequent call `f.plotOn(framex)` retrieves this complete list of data observables and compares it to the list of model observables and concludes that both `fxy` and data have a common observable – y – in addition to the plotted x observable. Therefore the `RooAbsPdf::plotOn()` call automatically integrates `fxy` over y before adding it to `framex` to ensure that both data and model represent the same ‘view’. Any such transformation in plotting is always announced:

```
RooAbsReal::plotOn(fxy) plot on x integrates over variables (y)
RooAbsReal::plotOn(fxy) plot on y integrates over variables (x)
```

The integrals involved in the creation of p.d.f. projections can be quite cumbersome, e.g. for Example 9 they are

$$f_x(x,a,b) = \frac{\int f(x,y,a,b)dy}{\iint f(x,y,a,b)dxdy} \quad ; \quad f_y(y,a,b) = \frac{\int f(x,y,a,b)dx}{\iint f(x,y,a,b)dxdy}$$

where *f*(*x*,*y*,*a*,*b*) is the unnormalized expression that was entered in the `RooGenericPdf` constructor.

Constructing multi-dimensional p.d.f.s through multiplication

Although it is straightforward to define monolithic multi-dimensional p.d.f.s such as `f`, they are actually not very common in practice. In many real-life situations you deal with observables that are (nearly) uncorrelated and you construct a multi-dimensional model for such cases by simply multiplying a number of one-dimensional p.d.f.s:

$$F(x,y;p,q) = f(x;p)\cdot g(y;q)$$

The tradeoff between a monolithic p.d.f. and a factorizing product p.d.f is a classic tradeoff between performance and simplicity one side and maximum flexibility and accuracy on the other side. Product p.d.f.s are very elegant in use: if the input p.d.f.s *f*(*x*;*p*) and *g*(*y*,*q*) are both properly normalized than *F*(*x*,*y*;*q*,*p*) is automatically normalized too. The interpretation is also straightforward: *f*(*x*;*p*) defines the distribution of the model in *x* and *f*(*y*,*q*) defines the distribution of the model in *y*.

The biggest drawback of the product construction is that you cannot introduce correlations between the observables because the product terms are by construction uncorrelated. They can however be introduced in an elegant way through a variant of the product construction: the conditional product. We will come back to this in the next section.

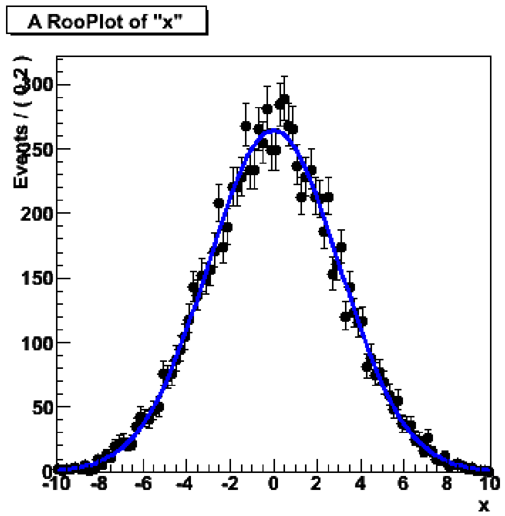
Class RooProdPdf

In RooFit the construction of any kind of product p.d.f. is done through class `RooProdPdf`. Here is a simple example:

```
RooRealVar x("x","x",-10,10) ;
RooRealVar meanx("meanx","meanx",0,-10,10) ;
RooRealVar sigmax("sigmax","sigmax",3,0.,10.) ;
RooGaussian gaussx("gaussx","gaussx",x,meanx,sigmax) ;

RooRealVar y("y","y",-10,10) ;
RooRealVar meany("meany","meany",0,-10,10) ;
RooRealVar sigmay("sigmay","sigmay",2,0.,10.) ;
RooGaussian gaussy("gaussy","gaussy",y,meany,sigmay) ;

RooProdPdf gaussxy("gaussxy","gaussxy",RooArgSet(gaussx,gaussy)) ;

RooDataSet* data = gaussxy.generate(RooArgSet(x,y),10000) ;
gaussxy.fitTo(*data) ;

RooPlot* framex = x.frame() ;
data->plotOn(framex) ;
gaussxy.plotOn(framex) ;

RooPlot* framey = y.frame() ;
data->plotOn(framey) ;
gaussxy.plotOn(framey) ;
```

Example 11 – A 2-dimensional p.d.f. constructed as the product of two one-dimensional p.d.f.s

The product p.d.f. `gaussxy` can be used for fitting and generating in exactly the same way as the monolithic p.d.f. `f` of Example 9. Note that `RooProdPdf` can multiply *any* number of components, in this example we multiply two one-dimensional p.d.f.s, but you can equally well multiply e.g. 7 one-dimensional p.d.f.s or 2 five-dimensional p.d.f.s



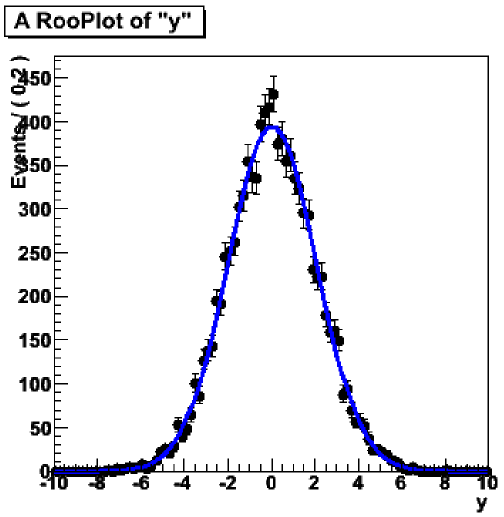


Figure 16 – Output from Example 11

Projection integrals over generic multi-dimensional p.d.f.s such as `f` are by default created through the `createIntegral()` method of that p.d.f. and are calculated analytically or numerically depending on the availability of analytical integrals, as advertised by the p.d.f. For multi-dimensional p.d.f.s that are defined as a product of factorizing terms, i.e. `RooProdPdf` objects, the integral calculation is automatically factorized as well and often results in a significant simplification of the calculation. For example the integration of gaussxy over y *is* trivial:

$$F_x(x;\vec{p}) = \int f(x;\vec{p})g(\vec{y};\vec{p})d\vec{y} = f(x;\vec{p})\int g(\vec{y};\vec{p})d\vec{y} = f(x;\vec{p})\cdot 1$$

and comes out to the intuitively expected answer: *f*(*x*,*p*).

Two-dimensional views

You can also make two-dimensional plots of multi-dimensional p.d.f.s, but he interface to do this is more rudimentary as two-dimensional views lends themselves less to manipulation and layering. It is difficult for example to overlay a two-dimensional view of data and a model and judge by eye if they agree.

In RooFit you can create 2 or 3 dimensional view of datasets and model represented as ROOT TH2 or TH3 objects. The code below creates a two-dimensional histogram of the data and p.d.f. of Example 10 and shows them side by side:

```
    TH2* hd = data->createHistogram("hd",x,Binning(20),YVar(y,Binning(20)));
    TH2* hf = gaussxy.createHistogram("hf",x,Binning(40),YVar(y,Binning(40))) ;

    TCanvas *c = new TCanvas("c","c",800,400) ;
    c->Divide(2) ;
    c->cd(1) ; hdata->Draw("lego") ;
    c->cd(2) ; hpdf->Draw("surf") ;
```

Example 12 – Generating two-dimensional plots of data and p.d.f.

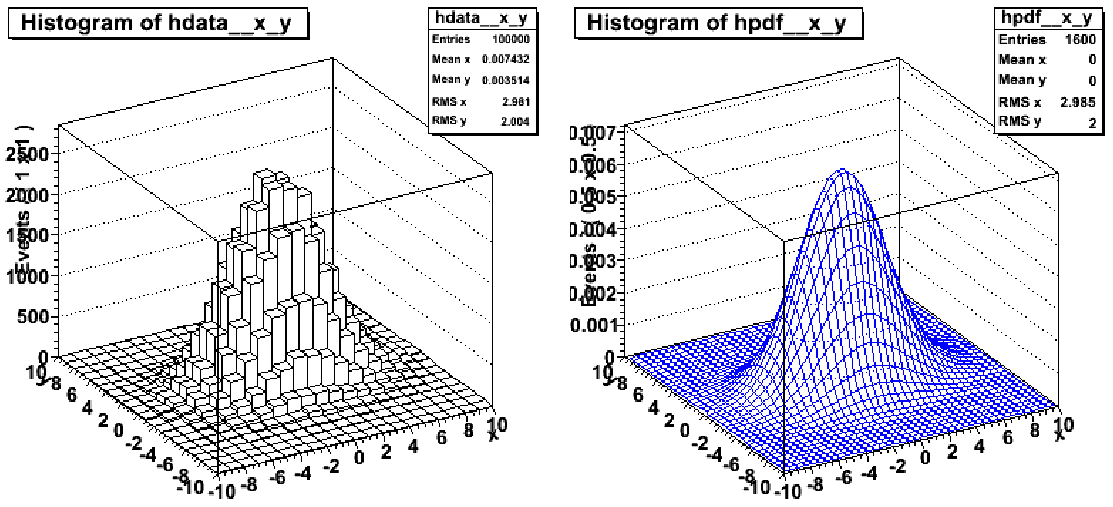


Figure 17 – Output of Example 12

The createHistogram() method of both RooAbsReal and RooAbsData can generate 1,2 and 3 dimensional ROOT histograms depending on the arguments. The option y and z variable are specified through the YVar() and ZVar() named arguments. The binning in each variables can be specified through the Binning() named argument. You can also restrict the range to be histogrammed in each dimension through a Range() named argument. Appendix A documents all options of the createHistogram() method.

Showing your multi-dimensional signal – slices and projections

Most of the new topics you encounter when going from one-dimensional to multi-dimensional models are in the area of visualization. Instead of a single 'view' of a model, you have multiple views: one for each observable. Alternatively you can make 2- or 3-dimensional views of models, but as mentioned before they are less easy to interpret. In this section we explore other ways to look at you N-dimensional model through 'slice' views: e.g. show the distribution of x in the 'signal region' of y. We will illustrate the concept using this simple two-dimensional model:

```
//--- Observables ---
RooRealVar x("x","x",-10,10) ;
RooRealVar y("y","y",-10,10) ;

//--- Signal p.d.f. ---
RooRealVar meanx("meanx","meanx",0,-10,10) ;
RooRealVar sigmax("sigmax","sigmax",3,0.,10.) ;
RooGaussian gaussx("gaussx","gaussx",x,meanx,sigmax) ;

RooRealVar meany("meany","meany",0,-10,10) ;
RooRealVar sigmay("sigmay","sigmay",2,0.,10.) ;
RooGaussian gaussy("gaussy","gaussy",y,meany,sigmay) ;

RooProdPdf sig("sig","gaussx*gaussy",RooArgSet(gaussx,gaussy)) ;

//--- Background p.d.f. ---
RooPolynomial flatx("flatx","flatx",x) ;
RooPolynomial flaty("flaty","flaty",y) ;

RooProdPdf bkg("bkg","flatx*flaty",RooArgSet(flatx,flaty)) ;

//--- Composite model ---
RooRealVar nsig("nsig","nsig",1000,0,10000) ;
RooRealVar nbkg("nbkg","nbkg",10000,0,1000000) ;
RooAddPdf model("model","sig+bkg",RooArgList(sig,bkg),RooArgList(nsig,nbkg));
```

Example 13 – A composite two-dimensional p.d.f.

We have a 2-dimensional signal and background, the signal is Gaussian in both observables, the background is flat in both observables. If we look at the two-dimensional distribution the signal is nicely visible, but as you can see in Figure 18 the one-dimensional projection and x and y do not do justice to the signal:

```
RooDataSet* data = model.generate(RooArgSet(x,y),10000) ;

TH2* hmodel2d =
   model.createHistogram("hmodel2d",x,Binning(40),YVar(y,Binning(40))) ;

RooPlot* xframe = x.frame() ;
data->plotOn(xframe) ;
model.plotOn(xframe) ;

RooPlot* yframe = y.frame() ;
data->plotOn(yframe) ;
model.plotOn(yframe) ;

TCanvas* c = new TCanvas("c","c",1200,400) ;
c->Divide(3) ;
c->cd(1) ; hmodel2d->Draw() ;
c->cd(2) ; xframe->Draw() ;
c->cd(3) ; yframe->Draw() ;
```

Example 14 – Plotting a 2-dimensional composite model

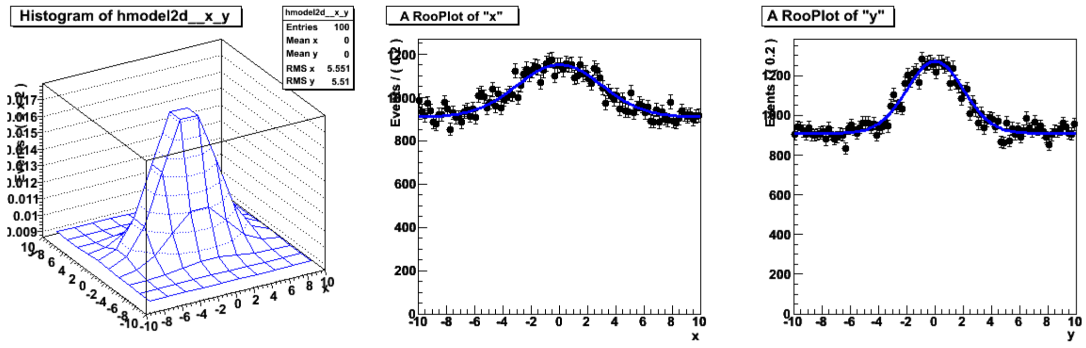


Figure 18 – Output of Example 14

The reason is of course that when you make a projection in either x or y of model you indiscriminately include the regions with a lot of signal – around (0,0) – as well as regions where there is only background. The result is that the nice peak in the 2-dimensional plot is watered down a lot.

You could choose to show the 2-D plot, but that has several disadvantages: it is difficult to overlay data and model for example. Another approach is to show only a *slice* of the data, i.e. when you show the projection in x, you for example only include data in the range -3<y<3.

Show the data with this requirement on y is conceptually straightforward: you only include data points in the histogram that meet your selection criteria in y. The projection of the model with the same requirement is less trivial, but still conceptually easy: the integration limits of the projection integral over y show now be adjusted from the full range to the range that we have selected, i.e.

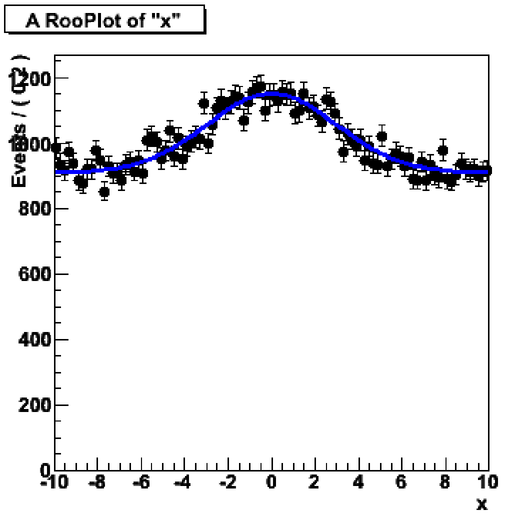
$$F_x^{a<y<b}(x) = \int_a^b F(x,y)dy$$

Here is how you do it:

```
// Define a range in y named “selection”
y.setRange(“selection”,-2,2) ;

RooPlot* xframe2 = x.frame() ;

// Plot data with requirement y named “selection”
data->plotOn(frame,CutRange(“selection”)) ;

// Plot model with requirement on y named “selection”
model.plotOn(frame,ProjectionRange(“selection”)) ;
model.plotOn(frame,ProjectionRange(“selection”),
              Components(“bkg”),LineStyle(kDashed)) ;
```

Example 15 – Projecting a slice of a two-dimensional p.d.f.

Result: a much more pronounced signal is visible in the slice projection Figure 19 (right) than in the ordinary ‘full’ projection Figure 19 (left). Lets now look a bit better at the code fragment that produced this plot. The use of ranges in plotting is always a two-step process: first you define a range with a name in one (or more) variables and then you use that range by referencing it by its name.
Though it requires you to write one extra line of code this approach has two advantages: 1) you define the requirement in a single location, eliminating the possibility of multiple inconsistent copies and 2) it allows you to refer to ranges in more than one variable through a single name. Also note that we have combined multiples options – to modify the projection range, the line style and the component selection – into a single call.



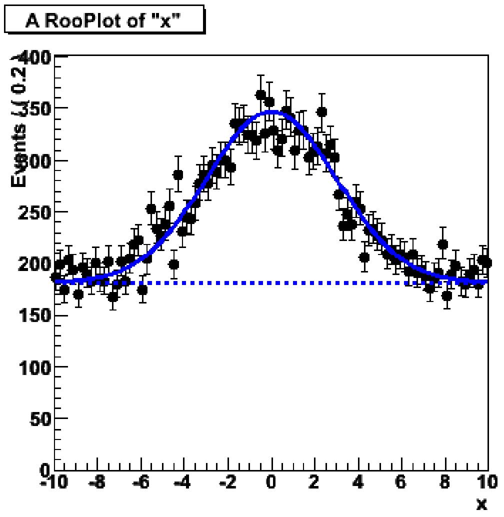


Figure 19 – Regular projection of p.d.f. from Example 13 (left) versus slice projection of same p.d.f (right, |y|<2)

A convenient feature of `CutRange()` and `ProjectionRange()` is that they automatically ignore any requirements imposed on the plot variable itself, which allow you to effectively use the 'named range ' concepts ability to refer to multiple range by the same name. This is illustrated in the following example:

```
// Define a range in y named “selection”
x.setRange(“selection”,-3,3) ;
y.setRange(“selection”,-2,2) ;

// Make plot of data and model in x with “selection” cut on y
RooPlot* xframe2 = x.frame() ;
data->plotOn(xframe2,CutRange(“selection”)) ;
model.plotOn(xframe2,ProjectionRange(“selection”)) ;

// Make plot of data and model in y with “selection” cut on x
RooPlot* yframe2 = y.frame() ;
data->plotOn(yframe2,CutRange(“selection”)) ;
model.plotOn(yframe2,ProjectionRange(“selection”)) ;
```

Dealing with more than 2 dimensions

The technique of projecting slices is easily generalized to p.d.f.s with more than 2 dimensions. Take as an example a three dimensional p.d.f. of the same form as the p.d.f. defined in Example 13, just add a `gaussz` and a `flatz`. A projection plot in any dimension now integrates out 2 dimensions and a slice projection plot can, at your choice, make a slice cut in either or both observables that are projected out. You can choose the range that you wish to select in each observable by calling `setRange()` for each observable:

```
// --- Construct three-dimensional p.d.f. ---
RooRealVar z(“z”,”z”,-10,10) ;
RooRealVar meany(“meanz”,”meanz”,0,-10,10) ;
RooRealVar sigmay(“sigmaz”,”sigmaz”,1,0.,10.) ;
RooGaussian gaussz(“gaussz”,”gaussz”,z,meanz,sigmaz) ;
RooPolynomial flatz(“flatz”,”flatz”,z) ;
```

```
   RooProdPdf sig3("sig3","gx*gy*gz",RooArgSet(gaussx,gaussy,gaussz)) ;
   RooProdPdf bkg3("bkg3","fx*fy*fz",RooArgSet(flatx,flaty,flatz)) ;
   RooAddPdf model3("m3","s3+b3",RooArgList(sig3,bkg3),RooArgList(nsig,nbkg));

   RooDataSet* data3 = model3.generate(RooArgSet(x,y,z),100000) ;

   // --- Make straight and slice projection on z ---
   x.setRange("selection",-3,3) ;
   y.setRange("selection",-2,2) ;

   RooPlot* zframe = z.frame() ;
   data3->plotOn(zframe) ;
   model3.plotOn(zframe) ;
   model3.plotOn(zframe,Components("bkg3"),LineStyle(kDashed)) ;

   RooPlot* zframe2 = z.frame() ;
   data3->plotOn(zframe2) ;
   model3.plotOn(zframe2,ProjectionRange("selection")) ;
   model3.plotOn(zframe2,ProjectionRange("selection"),
                 Components("bkg3"),LineStyle(kDashed)) ;
```

Example 16 – Projecting a 2-dimensional slice of a 3-dimensional model

The output is shown in Figure 20.

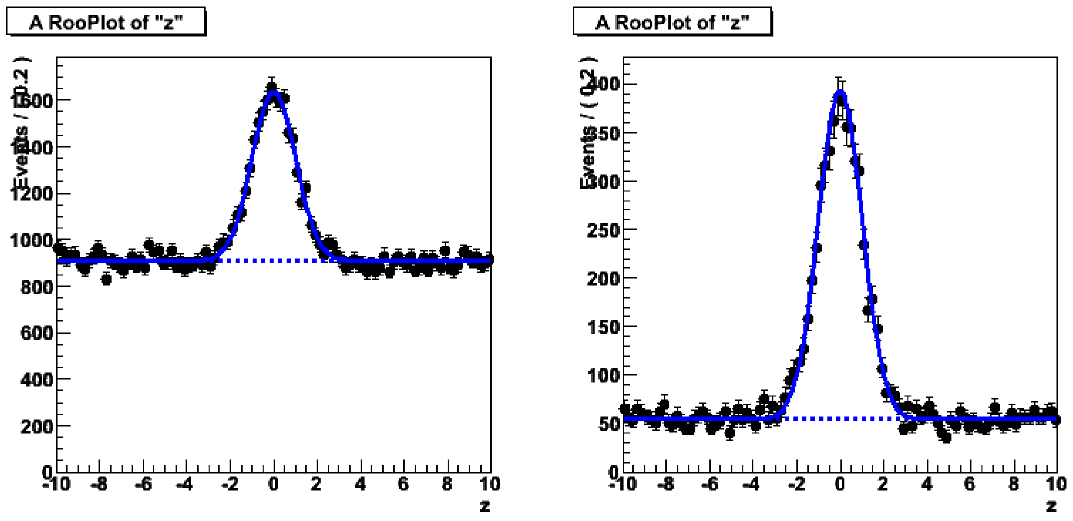


Figure 20 – Projection of 3-dimensional model of Example 16 on *z* (left) and projection of 2-dimensional slice "selection" in (*x,y*) on *z* axis.

Plotting multiple or discontinuous ranges

RooFit ranges are 'simple' ranges: each range is defined two parameters: the lower bound and the higher bound. Sometimes though there is good use for discontinuous ranges, for example a lower and higher sideband around a signal region. You effectively construct such discontinues ranges by simultaneously specifying multiple ranges in any command that accepts ranges e.g.

```
   x.setRange("sb_lo",-6,-3) ;
   x.setRange("sb_hi", 3, 6) ;

   RooPlot* xframe2 = x.frame() ;
   data->plotOn(xframe2,CutRange("sb_lo,sb_hi")) ;
   model.plotOn(xframe2,ProjectionRange("sb_lo,sb_hi")) ;
```

Note that when the ranges are active in more than one observable, ‘multiplication’ amongst observables takes precedence over ‘addition’ of ranges, i.e. given a range A and a range B defined in both x and y, the definition of (A,B) is `(A(x) && A(y)) || (B(x) && B(y))`.

Selecting arbitrarily shaped regions for plotting

The ‘range’ techniques projects a slice, box or (hyper-)cube shaped region on a one-dimensional plot, but you can also project differently shaped regions. Such projection integrals can in general not be calculated analytically and have to be performed with Monte Carlo-style methods. Before we get into the technical details on how to do that, we will first look a bit better in why you may want to that.

Generally speaking the goal of a ‘range’ plot is usually to visualize your capability to separate signal from background by showing the data distribution in one dimension but exploiting the discriminating information from all dimensions. In the standard range plot this is accomplished by cutting around the signal region in all dimensions that are projected out. When you try to generalize this concept two questions arise:

- how to you find the cuts that leads to the ‘best’ plot, and
- what is the optimal shape of the region to select
  (e.g. a (hyper)ellipsoid is probably better than a (hyper)cube).

The question of what is best somewhat subjective as it revolves around the issue of how to present your data, but generally involves finding a way that fairly represents the signal/background separation that you achieve in a multi-dimensional model in a one-dimensional plot. One of the standard approaches that address both of these issues is the so-called ‘likelihood-ratio plot’.

The likelihood ratio plot

The idea behind this method is that likelihood of the signal and background component of your model – or rather ratio of these likelihoods – contains all the information you have on signal/background discrimination. A ‘constant-likelihood surface’ is the optimal cut shape under the assumption that your model is correct. A nice feature of this method is that it reduces the definition of your multi-dimensional signal region to a definition with a single parameter: the likelihood ratio value on the cut surface. This parameter controls the purity versus efficiency balance in your plot. The likelihood ratio plot is not as easily produced as a hyper-cube slice plot because it involves numeric methods.

In practice, a likelihood ratio plot is constructed as follows: for a composite model *M(x,y,z) = fS(x,y,z) + (1-f)B(x,y,z)* and a dataset *D(x,y,z)* you

- Formulate a likelihood ratio R(x,y,z) = S(x,y,z)/B(x,y,z)
- Plot the subset of the data D(x,y,z) for which R(x,y,z,) > R_{cut}
- Project model M using a Monte Carlo technique for the region of phase space in which it predicts R(x,y,z)>R_{cut}

Most of the practical difficulties arise the calculation of the latter, as the over the region defined by R(x,y,z)>R_{cut} can in all but exceptional cases *not* be performed analytically. In the remainder of this section we will explain how to perform these steps in RooFit.

Formulating the likelihood

We illustrate the formulation of the likelihood ratio in RooFit using the model of Example 16 as a starting point:

```
// ---  Create (x,y) projection of signal and background p.d.f.s ---
RooAbsReal* sigProj = sig3.createProjection(RooArgSet(x,y),z) ;
RooAbsReal* bkgProj = bkg3.createProjection(RooArgSet(x,y),z) ;

// --- Construct log10 of ratio of S(x,y) and B(x,y) ---
RooFormulaVar llRatio_func("llRatio","log10(Lsig/Lbkg)","log10(@0/@1)",
                           RooArgList(*sigProj,*bkgProj)) ;
```

In this example we do not use the straight signal and background likelihoods as input for the ratio, but rather their integrals over z:

$$S(y,z) = \int S(x,y,z)dx \quad ; \quad B(y,z) = \int B(x,y,z)dx$$

You do this because you do *not* want use the information in the z observable in the likelihood ratio if you are plotting the distribution of z: that would be equivalent to showing a distribution on *z* after a cut on *z*. The creation of the projection integrals that take out the information in *z* is facilitated by the `createProjection()` method of `RooAbsReal`. Its arguments are the observables over which the resulting integrals should be normalized, (*x,y*) in the above example, and the observables which should be integrated out, *z* in the above example. The actual ratio is calculated by a `RooFormulaVar`, which in the example above, also takes the *10log* of the ratio as this compresses long tails in the distribution, and this thus easier to work with in practice.

Plotting the data with a likelihood ratio cut
Now we are ready to visualize the data. First we calculate the value of R for each data point and plots its distribution.

```
// --- Calculate llRatio for each point in data3 and add as column to data3 -
RooRealVar* llRatio = data3->addColumn(llRatio_func) ;

// --- Plot distribution of llRatio values in data3 ---
RooPlot* lframe = llRatio->frame(Range(-10,2)) ;
data3->plotOn(lframe) ;
```

The first step is accomplished by the `addColumn()` function of `RooAbsData`. This function takes one or more `RooAbsReal` function objects, evaluates their outcome for each data point and adds a column to the dataset with the name of the function object that stores the calculated value of that function. Now the values of `llRatio` can be simply treated as an observable and we can plot its distribution the usual way with one caveat: we have never associated a default range with `llRatio`, so give one explicitly here using the `Range()` argument. Here we have chosen a range bracketing 0 for `llRatio` as the interesting region of signal-to-background ratios typically hovers around unity. You can also let RooFit do range determination for you using the `AutoRange()` or `AutoSymRange()` options:

```
RooPlot* frame = llRatio->frame(AutoRange(*data3)) ;
RooPlot* frame = llRatio->frame(AutoSymRange(*data3)) ;
```

The `AutoRange()` option chooses a range for you that fits all data in a given dataset with some extra margin of each side. The `AutoSymRange()` option does the same, but it additionally adjusts the margins such that the mean value of the distribution is by construction in the center of the range.

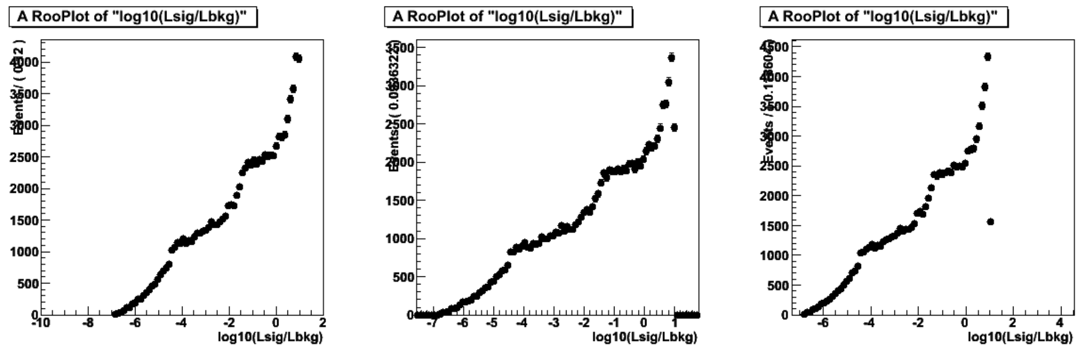


Figure 21 – Distribution of S(x,y)/B(x,y) likelihood ratio, plotted in frames created with `Range(-2,10)`, `AutoRange()` and `AutoSymRange()` respectively. (Data is identical all plots)

Figure 21 shows the distribution of `llRatio` as generated by the example code. Based on the distribution of `llRatio` choose a cut 0. Note that this cut value is arbitrary to a certain extent and you should experiment here to achieve the result you like the best. Now we select the subset of events in data3, for which `llRatio` is greater than zero and plot its distribution in z

```
// --- Plot distribution of z values after cut S(x,y)/B(x,y) ratio ---
RooDataSet* selData3 = data3->reduce("llRatio>0") ;

RooPlot* frame = z.frame() ;
selData3->plotOn(frame) ;
```

For the selection step we use the `reduce()` method of `RooAbsData`, which takes any Boolean formula expression in terms of the dataset observables. Figure 22 shows the output of this code, as well as the output of a variation where we have required a cut value of -1 and +1 instead.

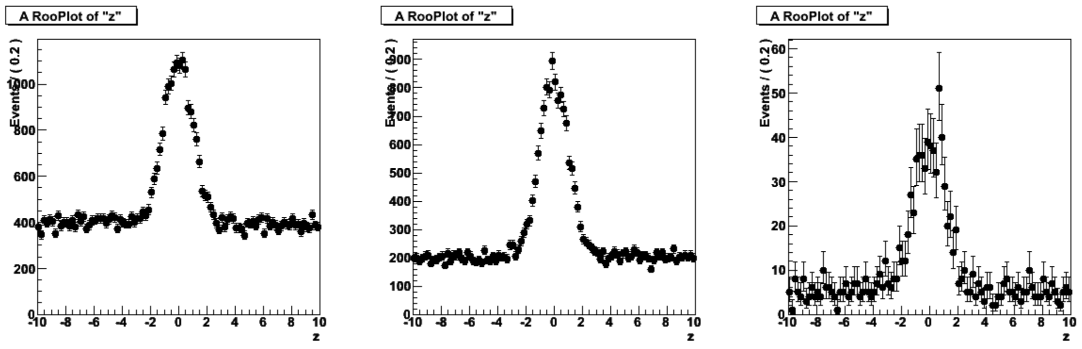


Figure 22 – Distribution of events in *z* after `llRatio` cut at -1, 0 and +1 respectively

Plotting the p.d.f. projection with a likelihood ratio cut
To complete the plots of Figure 22 we should add a projection of the model with an identical requirement, which revolves around calculating the integral

$$\int_{R(x,y)>R_{cut}} M(x,y,z)dxdy$$

As this integral can not be calculated analytically in all but exceptional cases we follow a Monte Carlo inspired numerical approach. The Monte Carlo theorem says that you can approximate any integral by a sum of values randomly sample from the distribution you are trying to integrate, i.e.

$$\lim_{n\to\infty}\left[\frac{1}{n}\sum_{i=1,n} M(\vec{x}_i)\right]_{x_i\in R} = \int_R M(\vec{x})d\vec{x}$$

with the values x_i randomly sample from M, such as is done for example by `M.generate()`. This theorem is valid for *any* region R to be integrated and provides an easy way to approximate the integral that you need to calculate for the model projection with a likelihood ratio cut. Here is the complete code for the projection of model3 with a likelihood ratio cut:

```
// --- Generate toy MC events from model3, calculate llRatio for each
//                            toy event and add it as column to toyData ---
RooDataSet* toyModelData = model3.generate(RooArgSet(x,y,z),10000) ;
toyModelData->addColumn(llRatio_func) ;

// --- Select subset of toy events that S/B ratio cut ---
RooDataSet* modelProjData = toyModelData->reduce("llRatio>0") ;

// --- Draw model projection over z data in frame ---
model3.plotOn(frame,ProjWData(*modelProjData)) ;
model3.plotOn(frame,ProjWData(*modelProjData),Components("bkg3"),
              LineStyle(kDashed)) ;
```

This code example recycles the `addColumn()` and `reduce()` techniques that we used earlier to project the data, but differs in the final plotting call. The `ProjWData()` argument to `RooAbsPdf:plotOn()` instructs RooFit to perform the projection integral explicitly through a Monte Carlo approximation using the given dataset, rather than using the default numerical or analytical integration techniques. By giving a dataset with a preselection on `llRatio` we effectively perform the integral over the region defined by `llRatio>0` and thus construct a p.d.f. projection that is consistent with the data. The final result is shown in Figure 23 with a likelihood ratio cut at -1, 0 and +1 respectively.

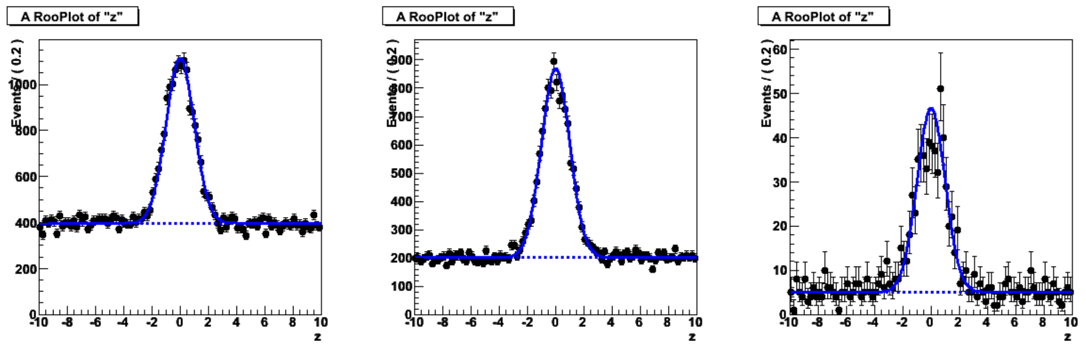


Figure 23 – Distribution of events in z after `llRatio` cut of -1, 0 and +1 respectively overlaid with p.d.f. projection with identical requirement.

Likelihood ratio plot – Putting it all together

Here is the complete code to make a likelihood ratio plot starting from the model defined in Example 16.

```
// ---  Create (x,y) projection of signal and background p.d.f.s ---
RooAbsReal* sigProj = sig3.createProjection(RooArgSet(x,y),z) ;
RooAbsReal* bkgProj = bkg3.createProjection(RooArgSet(x,y),z) ;

// --- Construct log10 of ratio of S(x,y) and B(x,y) ---
RooFormulaVar llRatio_func("llRatio","log10(Lsig/Lbkg)","log10(@0/@1)",
                           RooArgList(*sigProj,*bkgProj)) ;
```

```
// --- Calculate llRatio for each point in data3 and add as column to data3 -
RooRealVar* llRatio = data3->addColumn(llRatio_func) ;

// --- Plot distribution of llRatio values in data3 ---
RooPlot* lframe = llRatio->frame(Range(-10,2)) ;
data3->plotOn(lframe) ;

// --- Plot distribution of z values after cut S(x,y)/B(x,y) ratio ---
RooPlot* frame = z.frame() ;
RooDataSet* selData3 = data3->reduce("llRatio>0") ;
selData3->plotOn(frame) ;

// --- Generate toy MC events from model3, calculate llRatio for each
//                            toy event and add it as column to toyData ---
RooDataSet* toyData = model3.generate(RooArgSet(x,y,z),10000) ;
toyData->addColumn(llRatio_func) ;

// --- Select subset of toy events that S/B ratio cut ---
RooDataSet* projData = toyData->reduce("llRatio>0") ;

// --- Draw model projection over z data in frame ---
model3.plotOn(frame,ProjWData(*projData)) ;
model3.plotOn(frame,ProjWData(*projData),Components("bkg3"),
              LineStyle(kDashed)) ;

TCanvas* c = new TCanvas("c","c",800,400) ;
c->Divide(2) ;
c->cd(1) ; lframe->Draw() ;
c->cd(2) ; frame->Draw() ;
```

Example 17 – Complete code to construct a likelihood ratio plot

7. Correlations in multi-dimensional models

We now take the building of multi-dimensional models one step further by introducing explicit correlations between observables. A multi-dimensional model includes correlations if it can not be factorized into a product of one-dimensional p.d.f.s. Dealing with correlations is a central issue in many multivariate analyses. Many standard techniques, for example a neural network, find and exploit correlations between observables automatically for you, but sometimes you may want to explicitly deal with correlations yourself: if you are a priori aware of these correlations and know how to describe them.

In the introduction of the preceding section we looked at a 2-dimensional model with correlations through the construction of a 2-dimensional `RooGenericPdf` based on a non-factorizable formula involving observables x and y:

```
RooGenericPdf f("f","a*x*x+b*y*y-0.3*y*y*y",RooArgSet(x,y,a,b)) ;
```

While this approach explicitly describes the distribution in x, y and the correlation between x and y, it is not so easy to disentangle the three. If you want to modify the functional form such that the distribution in y changes, but the distribution in x and the correlation between x and y are preserved, it is not obvious how you should do that. The concept of a *conditional* probability density function makes it easier to achieve such a conceptual separation. Before we get into the details of conditional p.d.f.s, we introduce a realistic example of an analysis with known correlations that will serve as illustration and make clear why you want to be able to achieve this type of factorization in your model description.

Example analysis with a known correlation between observables

Suppose we want to measure the lifetime of the decay of a particle in a generic particle physics experiment. The first step in this measurement is to collect a data sample with observed decays. Each decay is described by a decay time, which is derived from a flight length measurement between the production vertex of the particle and the decay vertex of the particle. For an ideal detector the distribution of observed decay times is an exponential distribution with an exponent that is the inverse of the lifetime τ of the particle:

$$F_I(t) = \exp(-t/\tau)$$

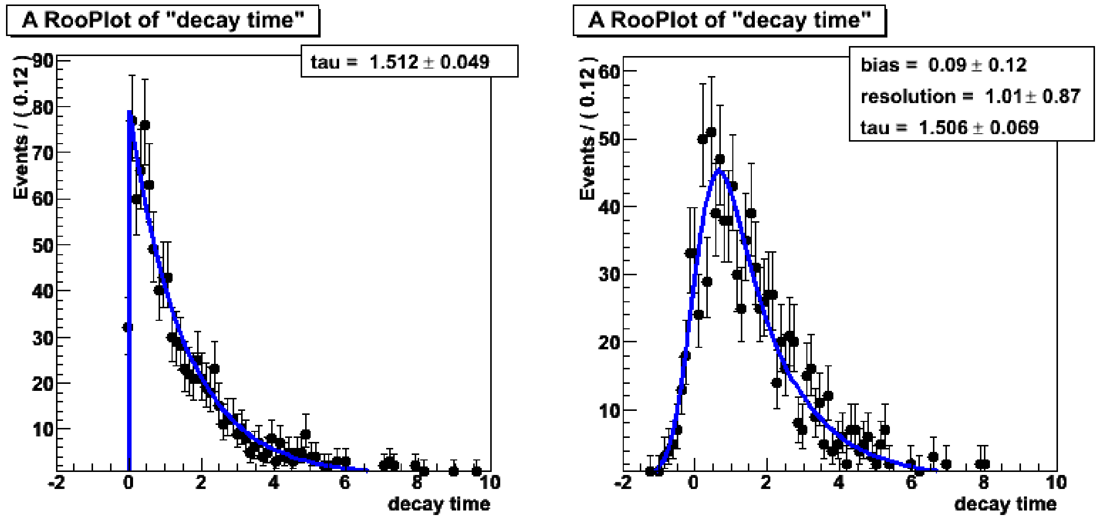


Figure 24 – Distribution of decay times measure with ideal detector (left) and realistic detector(right)

A real-life detector has a finite experimental resolution on each measurement *t* of the decay time. We adjust our model to incorporate a Gaussian measurement uncertainty on each t by convolving F_I with a Gaussian:

$$F_R(t) = \exp(-t/\tau) \otimes G(t,\mu,\sigma) \equiv \int \mathrm{d}t' \exp(-t'/\tau)\, G(t-t',\mu,\sigma)$$

In this expression G denotes a Gaussian with mean μ and width σ. The width σ expresses the experimental resolution on each measurement of t and the mean μ parameterizes the average bias in that measurement. We assume the latter to be zero for the sake of this examples simplicity. Figure 24 shows the ideal and realistic model F_I and F_R fit to a sample of toy Monte Carlo events. You can see from the magnitude of error on the fitted value of τ that the finite *t* resolution of the realistic model reduces the precision of the measurement of τ.

Introducing a second correlated observable

Each measurement of a decay time t in our example is the result of a measurement of the distance between two decay vertices that are each calculated from the intersection of a number of charged particle trajectories. These vertex positions have uncertainties associated to them that are derived from the uncertainties on the participating charged particle trajectories and can be used to assigned an experimental error *dt* to each measurement *t*. This means that the detector resolution on *t* is not really a fixed value, but rather varies from event to event.

Our example of a decay time measurement has not been randomly chosen: it represents a large class of measurements where an observable x is accompanied by an error estimate *dx* that can be treated as a second correlated observable in the model that describes the experimental results.

We modify the model such that each event is characterized by a pair of values (*t,dt*) rather than a single number *t* and thereby we acknowledge that certain events – those with small *dt* – carry more information then others, and use this information to achieve a better measurement of τ with the same data. Here is the enhanced p.d.f:

$$F_E(t,dt) = \exp(-t/\tau) \otimes G(t,\mu,\mathbf{dt})$$

It is easy to see that this small modification – replacing the resolution estimate σ by the per-event error dt – accomplishes what you want. Imagine to events A and B with identical observed decay times $t_A=t_B=t$ and uncertainties that differ by a factor of two $dt_A= dt_B/2$, the contribution of event A to the total likelihood will differ from the contribution of event B because exponential shape of the model for event A is convolved with a Gaussian that is twice as small as that for event B. A refit of the data sample of Figure 24 to this enhanced model reflects the enhanced statistical power of this model, by reducing the measurement error of τ from 0.067 to 0.060, a 10% improvement of the measurement performed on the same data that is equivalent to having 20% more data available[4].

There is one major caveat in the enhance model F_E: *it assumes that the error estimates dt are correct*. If these estimates are too small on average in reality, the error on the physics parameter τ will be too small as well. As this is highly undesirable, you should verify the correctness of the errors dt by looking at pull distributions, i.e. comparing the spread of the measured values (the external error) to the distribution of the given errors (the internal error). Fortunately this check an be trivially incorporated in the model F_E through the following modification:

$$F_E(t,dt) = \exp(-t/\tau) \otimes G(t,\mu,\mathbf{s}\cdot\mathbf{dt})$$

Now the model doesn't make any *absolute* interpretation of the errors dt, it just assumes that the true uncertainty of each t measurement scales linearly with the provided error. The parameter s serves as a global scale factor applied to the per-event errors dt. If you fit this model to the data and the uncertainty estimates dt turn out the be correct on average you will find that σ=1. If the error estimates

[4] The actual gain depends on the spread of the per-event errors. The chosen example is typical for BaBar experimental data.

are too high or too low on average, this is apparent from a mismatch in the distribution of values and errors in the data and the fit will steer σ to a value smaller or greater than 1. Effectively one could interpret G as a fit to the pull distribution associated with the vertexing procedure. Thanks to this built-in correction of the per-event errors the improved model F_E has gained an important quality: the error on the physics parameter τ is to first order *independent* of the correctness of the error estimates dt. (A second order dependency comes in when the pull distribution of the dt errors cannot be accurately described by a Gaussian. Also this can be mitigated, for example by replacing G by a sum of two or more Gaussians of different width and mean).

In summary, incorporating the errors *dt* on each decay *t* in your model in the form F_E gives you enhanced statistical sensitivity to τ, it gives you an estimate of the correctness of the provided errors, and *dt* cancels to first order the effects that arise from *dt* estimates that are too small or too large on average.

Some practical caveats

The enhanced decay time model is a great example of a p.d.f with an explicit correlation between observables, but it has some practical caveats that have not been discussed yet and that should be addressed before one can use it in practice: If you were to fit F_E as written above to data you would effective use it as a two-dimensional model predicting the distribution of t, the correlation between t and *dt*, *and the distribution of dt,* as there is nothing in the description of F_E that warrants a special treatment of *dt*. In the description of the example we've conveniently left in the middle what the distribution of *dt* is, but once you start actually using your model with data this becomes acutely relevant as your model must be able to describe the data's *dt* distribution.

Lets examine our example a bit further: the prediction of F_E for the distribution of dt is obtained by integrating $F_E(t,dt)$ over *t*. You cannot do this analytically, but RooFit can do it numerically, and the result is a more or less flat distribution in *dt* as shown in Figure 25. (NB: The slight drop-off in Figure 25 towards high values of *dt* is caused by the finite range of t in the definition of F_E)

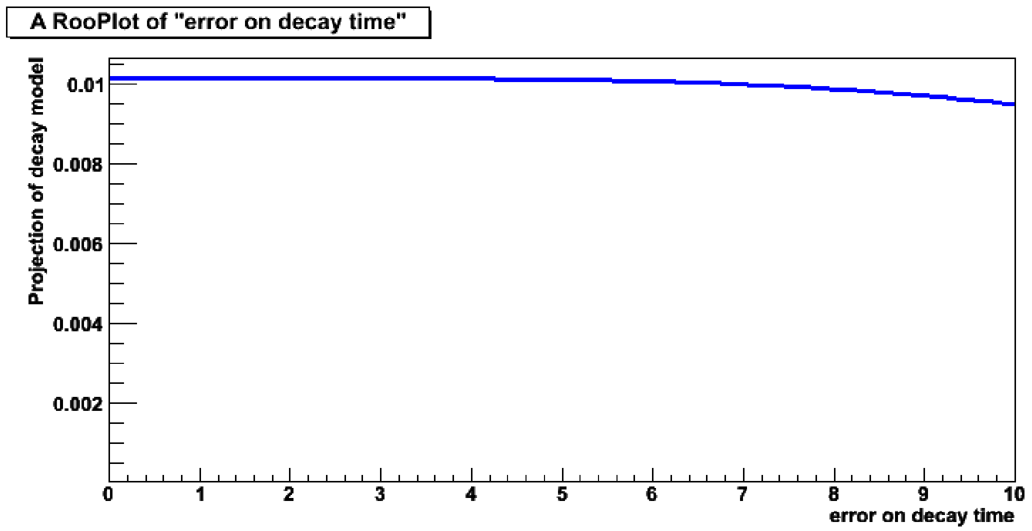


Figure 25 – Prediction of F_E for the distribution of the per-event error dt

Your actual distribution of *dt* in data is likely to be very different, so fitting F_E to data would result in a bad fit. Even worse, you don't have any knobs to turn to modify the shape of the *dt* distribution predicted by F_E without altering its behavior in t and in the correlation between t and *dt*. So here were are back at the opening question of the chapter '*If you want to modify the functional form such that the distribution in y changes, but the distribution in x and the correlation between x and y are preserved, it is not obvious how you should do that.*' We have now seen *why* you want that, next we will talk about *how* you do it. The key to this lies in the concept of conditional p.d.f.s

Conditional probability density functions

The premise behind a *conditional* probability density function F(x|y) is that it describes the distribution of a set of observables x *given* the values of a set of other observable y. In effect F(x|y) describes the distribution of x, the correlation between x and y, but not the distribution of y. Conditional p.d.f.s differs in only one respect from ordinary p.d.f.s: the normalization condition. Whereas a regular two-dimensional p.d.f F(x,y) meets the normalization condition

$$\int F(x,y)dxdy \equiv 1$$

A conditional p.d.f. F(x|y) meets the normalization condition

$$\int F(x,y)dx \equiv 1 \text{ for all values of } y \qquad (1)$$

A conditional p.d.f F(x|y) has no predictive power in y, it just takes the distribution of y as a given and predicts the distribution of x for that value of y. This is precisely the way we want to use our enhanced life time model F_E: rather than fitting $F_E(t,dt)$ to data we want to fit $F_E(t|dt)$ to data. How can we do this in RooFit? In RooFit you can perform both fits, regular and conditional, from the *same* `RooAbsPdf` object: each RooFit p.d.f is always constructed from a function expression that is explicitly normalized by dividing that expression by its integral

$$F(x,y)=\frac{f(x,y)}{\int f(x,y)dxdy}, \quad F(x\,|\,y)=\frac{f(x,y)}{\int f(x,y)dx}$$

The only issue is that you have to indicate in the use context that you want a `RooAbsPdf` to represent the conditional form that than the regular form. The next section will explain how to do that.

Using conditional p.d.f.s for fitting, plotting and generating

We first code the enhanced life time p.d.f. of the opening section to be able to concretely illustrate the various uses of conditional p.d.f.s.:

```
// Observables
RooRealVar t("t","decay time",0,20) ;
RooRealVar dt("dt","error on decay time",0,1) ;

// Gaussian resolution model Gauss(t,0,s*dt) ;
RooRealVar s("s","resolution",3,0,20) ;
RooGaussModel res("res","det. resol.",t,RooConst(0),s,dt) ;

// NB: Convenient special ctor of RooGaussModel with 4 arguments
// defines gaussian width as product of 3rd and 4th argument and saves you
// the effort of a separate RooFormulaVar object

// Decay (x) res model
RooRealVar tau("tau","lifetime",1.5,0,20) ;
RooDecay decay("decay","decay model",t,tau,res,RooDecay::SingleSided) ;
```

As was just explained, the definition of decay – like any other p.d.f – has not, and needs not, to have any clauses relating to possible use as a conditional p.d.f

Given a dataset `data` with observables `t` and `dt` we now explicitly fit `decay` as conditional p.d.f using the `ConditionalObservables()` directive in the `fitTo()` command:

```
// Fit decay(t|dt) as conditional p.d.f. to data(t,dt)
decay.fitTo(data,ConditionalObservables(dt)) ;
```

The effect of the `ConditionalObservables()` argument is that the likelihood function that it is constructed using `decay` in its conditional form:

$$NLL_{reg}(\tau,\sigma) = -\sum_{D} \log\left(decay(t_i, dt_i; \tau, \sigma)\right)$$
$$NLL_{cond}(\tau,\sigma) = -\sum_{D} \log\left(decay(t_i; \tau, \sigma \mid dt_i)\right)$$

It is an instructive exercise to compare the output of a regular `fitTo()` operation with the `Verbose()` argument with that of a `fitTo()` operation with both the `Verbose()` and the `Conditional-Observables()` arguments: you will see that the absolute value of the likelihood printed by the `Verbose()` option is very different. This is a direct consequence of the different normalization conditions illustrated in Eq. 1. You will also notice that the performance differs: in the fit to the conditional form of `decay` the normalization integral needs to be evaluated for every event, as it has a different value for each event. The normalization term of `decay` when used as a regular fit on the other hand depends only on parameters, and is thus only evaluated when MINUIT changes those parameter values, a much less frequent occurrence.

The visualization of conditional models is conceptually more complicated as the shape of a conditional p.d.f. is partly dictated by the data it is aiming to describe. Lets look concretely at our example analysis: there are two data plots you can make: the distribution of t and the distribution of *dt*. Since a conditional p.d.f. *decay(t|dt)* has by construction no predictive power in *dt* we cannot plot it as function of *dt*. That leaves us with a plot of the distribution of t that can be overlaid with a matching projection of decay over observable *dt*. As integration makes no sense here, again because *decay(t|dt)* has no predictive power in *dt*, we project out *dt* by summing the distribution of *decay(t|dt)* over each *dt* value in the data, i.e.

$$\frac{1}{n}\sum_{i=1,n} \frac{d(t \mid dt_i)}{\int d(t \mid dt_i)dt}$$

This technique is identical to the Monte Carlo integration technique described in the preceding chapter, it is only applied with different data: for Monte Carlo integration we summed over the values of a simulated dataset sampled from the p.d.f. itself, whereas here we sum over the values of the experimental data:

```
RooPlot* frame = t.frame() ;
data.plotOn(frame) ;
decay.plotOn(frame,ProjWData(dt,data)) ;
```

In the `ProjWData()` we specify the dataset with *dt* values over which should be summed – here we use the actual `data` – and we also specify that we only wish to override the projection method for the observable `dt`. The latter is not strictly necessary in this example as `dt` is the only observable to be projected in `data`, but it is good practice to do spell that out explicitly. In cases where data contains additional observables that you still want to be projected out through integration, this specification is essential. Figure 26 shows the *t* and *dt* distributions of our example data and the properly projected t distribution of the conditional p.d.f *decay(t|dt)*. *Note that a projection of a conditional p.d.f. is always a ‘hybrid’ object: it is not a pure model prediction, but a conditional prediction tailored to the data it is being compared to.*

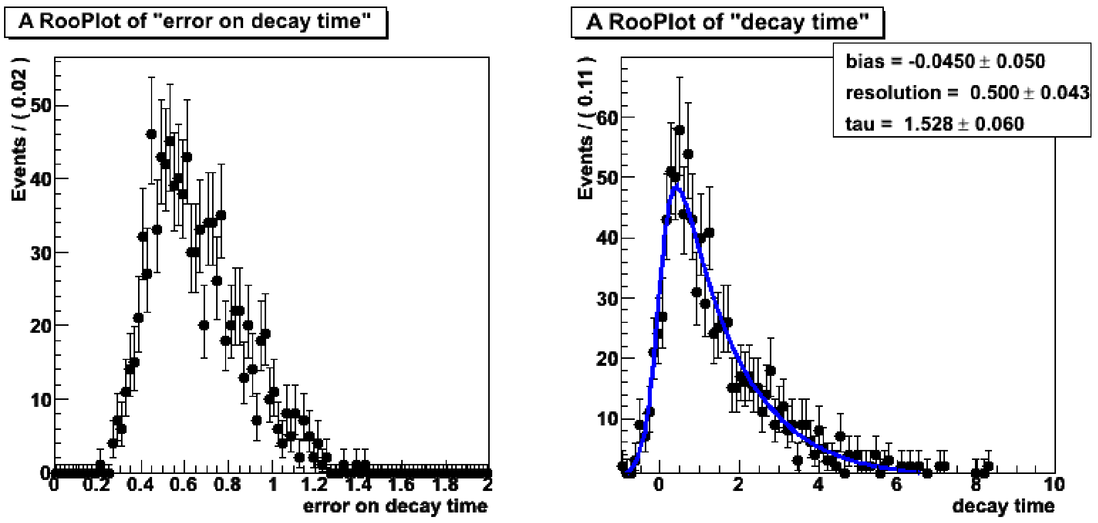


Figure 26 – Distribution of decay time errors (left) and distribution of decay times, overlaid with conditional p.d.f. decay(t|dt) projected with dt values of the data (right).

Generating events using p.d.f. in conditional form requires external input on the conditional variables, just like in plotting conditional p.d.f.s. The simplest way to do this is to pass the generator an existing set of *dt* values and ask it to generate the corresponding t values according to the model

```
RooDataSet* toyData = decay.generate(t,ProtoData(data)) ;
```

The result of this operation is a two-dimensional dataset with values of *t* and *dt*. The *dt* values are identical to those of the input dataset passed through the `ProtoData` argument, the *t* values and their correlation with dt are generated from the decay p.d.f. You do not need to specify a number of events to be generated when you use `ProtoData()`, the number of events in the prototype dataset is the implicit default, but you still can change this through an explicit `NumEvents()` argument. Beware aware though that when you require more events to be generated then are available in `data`, certain data entries will be used more than once.

If you wish to describe the distribution in *dt* with a p.d.f. rather than with a collection of values, the event generation becomes a two-step process: First you sample a *dt* distribution from a regular p.d.f, then you sample the t distribution from a conditional p.d.f.

```
RooAbsPdf* dtModel ;
RooDataSet* dtData = dtModel->generate(dt,1000) ;
RooDataSet* allData = decay.generate(t,ProtoData(*dtData)) ;
```

Multiplying conditional p.d.f.s with regular p.d.f.s.

The ability to use any p.d.f. in conditional form in RooFit unlocks essential new ways to use a p.d.f in describing certain classes of problems, but their direct use is less elegant because information on the distribution of the conditional observables needs to be externally supplied in many operations. We will now look at another way to use conditional p.d.f.s that mitigates these practical problems: conditional products. The essence of the idea is that the final ‘high level’ p.d.f. is a p.d.f that can be used in regular mode even though it internally contains a conditional p.d.f. We can achieve this idea through a simple multiplication step: we multiply a conditional p.d.f.s with a supplemental p.d.f.s that describes the conditional observables to form a full p.d.f. For the initial example of this chapter this amounts to defining a new p.d.f as follows

F(t,dt) = decay(t|dt)·vtx(dt)

It easy to convince yourself that F(t,dt) is regular p.d.f by explicitly proving that F is normalized over both *t* and *dt*, i.e. Int F(t,dt) dt ddt = 1

$$\iint decay(t\mid dt)vtx(dt)dtddt = \int\left(\int decay(t\mid dt)dt\right)vtx(dt)ddt = \int 1vtx(dt)ddt = 1$$

While *F(t,dt)* is now a regular p.d.f in all respects we have retained the advantage of the conditional p.d.f: we have separated the description of the 'physics' part of *F*, the decay model and its correlation with the vertex error from the description of the 'empirical' part of *F*, the description of the distribution of vertex errors. This leaves us with the task of describing the distribution of *dt* one way or another, but this could – worst case – be done with a non-parametric p.d.f such as a `RooHistPdf` or a `RooKeysPdf`. Here is the reworked example of the introduction that use an – arbitrarily chosen – bifurcated Gaussian as a toy model for the distribution of vertex errors

```
// Bifurcated Gaussian p.d.g. as model for per-event vertex errors
RooRealVar m("m","mean of dt",0.5,0,1) ;
RooRealVar sl("sl","low-side sigma of dt gauss",0.1,0.,1.) ;
RooRealVar sr("sr","higth-side sigma of dt gauss",1,0.,10.) ;
RooBifurGauss vtx("vtx","vtx error dist",dt,m,sl,sr) ;

// Full model: product of conditional decay model with vtx toy model
RooProdPdf F("F","decay(t|dt)*vtx(dt)",Conditional(decay,t),vtx) ;
```

The `Conditional()` modifier in the constructor instruct `RooProdPdf` to interpret decay as a conditional p.d.f that only describes the observable t. Any other observable referenced in decay – in this case *dt* – is treated as a conditional observable. The `RooProdPdf` constructor is in fact the *only* place in which we will make this declaration of conditional use.

The net result is a p.d.f. *F* that has all the necessary information to describe the distribution of *t* and *dt* and we therefore can proceed as usual and work with *F* to generate events, fit it to data and plot it:

```
// Generate events from F(t,dt)
RooDataSet* data = F.generate(RooArgSet(t,dt),10000) ;

// Fit F to data
F.fitTo(*data) ;

// Plot t and dt distributions of data with F overlaid
RooPlot* tframe = t.frame() ;
data->plotOn(tframe) ;
F.plotOn(tframe) ;

RooPlot* dtframe = dt.frame() ;
data->plotOn(dtframe) ;
F.plotOn(dtframe) ;
```

The data samples shown in Figure 26 were in fact produced with this conditional product p.d.f. *F*.

It is instructive to understand what happens behind scenes in *F* when you plot, fit or generate conditional product p.d.f.s.

- Fitting is straightforward as the probability for each event is simply defined by `F(d,dt) = decay(t|dt)vtx(dt)`, which we already proved is properly normalized.

- Generating events from F is again a two-step process, as was the case for standalone use of conditional p.d.f.s, except that F now has all the information it needs to complete both generation step in a single command: first a value of `dt` is generated from `vtx`, then a value of `t` is generated by decay given the value of `dt`.

- Finally, for plotting the projection of `F` over `dt` is calculated as a standard numeric integral rather than as an implicit Monte Carlo approximation

$$F_t(t) = \int decay(t \mid dt) vtx(dt) ddt$$

Even though above projection integral can only be evaluated numerically, its evaluation is faster than a Monte Carlo approximation as specialized one-dimensional integration techniques such as the adaptive Gauss-Kronrod rule converge to a controllable precision in O(30) function evaluations.

8. Discrete variables

This section is scheduled for the next version of the manual

Representing discrete information – class RooCategory

Representing selection criteria – real-to-discrete functions

Tabulating discrete information

9. Multiple datasets and simultaneous fitting

This section is scheduled for the next version of the manual

The problem

The solution

Automating the solution

10. Organizational tools – Setting up a complex analysis

This section is scheduled for the next version of the manual

Using sets and list to manage user configuration

Automated function building and customization

11. Common issues, pitfalls and their solutions

This section is scheduled for the next version of the manual

Integrating PDFs

How to use `createIntegral()`

Tuning numeric integration parameters and methods

How to use `defaultIntegratorConfig()` etc…

Using weighted data

Describe ways to use weighted data and problems that may arise in likelihood fits

Adding penalty terms to a likelihood or $\chi2$

How to use `RooFormulaVar` to add penalty terms to a likelihood or $\chi2$

Interactive fitting

How to use class `RooMinuit` to do an interactive fit session

Blinding parameters

How to blind parameters in your fit

Rearranging the contents of a `RooPlot`, adding arrows, boxes etc…

How to rearrange the contents of a `RooPlot` and beautify it

Merging and concatenating, reducing data

How to merge and concatenate datasets

Fit instability due to strongly correlated parameters

Effects of bounding parameters

Observables and parameters

Distinguishing parameters from observables

There are tools to distinguish the ‘parameter’ variables of a p.d.f. from the ‘observable’ variables of a p.d.f. Lets first start with the definition of these terms:

- An *observable* is a variable of a p.d.f. that also occurs in the data. A p.d.f. is also normalized to unity with respect to all its observables.
- A *parameter* is any remaining variable of a p.d.f.s.

RooFit p.d.f.s have no intrinsic or static designation which variable is a observable and which variable is a parameter. This designation always arises dynamically in the context of data and is a fundamental design consideration. You can read more about this in appendix C. A consequence is that any routine that identifies a variable as either parameter or observable needs to be passed a dataset to be able to make the distinction:

```
RooAbsData* data ; // A sample dataset containing ‘x’
RooArgSet* params = model.getParameters(data) ;
RooArgSet* observables = model.getObservables(data) ;
params->Print(“1”) ;
observables->Print(“1”) ;

RooArgSet::parameters:
  1) RooRealVar::c0: "coefficient #0"
  2) RooRealVar::c1: "coefficient #1"
  3) RooRealVar::c2: "coefficient #2"
  4) RooRealVar::mean: "mean"
  5) RooRealVar::nbkg: "background fraction"
  6) RooRealVar::nsig: "signal fraction"
  7) RooRealVar::sigma: "sigma"

RooArgSet::dependents:
  1) RooRealVar::x: "x"
```

Appendix A – Quick reference guide

This appendix summarizes the most core *named argument* methods of RooFit for plotting, fitting and data manipulation. The named argument formalism chief advantage is that it is a flexible and self-documenting way to call methods that have a highly variable functionality. Here is the list of methods that is documented in this section

Plotting

Make a plot frame – `RooAbsRealLValue::frame()`

Usage example: **`RooPlot* frame = x.frame(…)`**

Create a new `RooPlot` on the heap with a drawing frame initialized for this object, but no plot contents. Use `x.frame()` as the first argument to the `y.plotOn(...)` method, for example. The caller is responsible for deleting the returned object.

This function supports the following optional named arguments

`Range(double lo, double hi)`	Restrict plot frame to the specified range
`Range(const char* name)`	Restrict plot frame to range with the specified name

`Bins(Int_t nbins)`	Set default binning for datasets to specified number of bins
`AutoRange(const RooAbsData& data, double margin=0.1)`	Choose plot range such that all points in given data set fit inside the range with given fractional margin.
`AutoSymRange(const RooAbsData data, double margin=0.1)`	Choose plot range such that all points in given data set fit inside the range *and* such that center of range coincides with mean of distribution in given dataset.
`Name(const char* name)`	Give specified name to `RooPlot` object
`Title(const char* title)`	Give specified title to `RooPlot` object

Some examples:

```
// Create frame with name "foo" and title "bar"
x.frame(Name("foo"),Title("bar")) ;

// Create frame with range (-10,10) and default binning of 25 bins
x.frame(Range(-10,10),Bins(25)) ;

// Create frame with range that fits all events in data with 10% margin that
// is centered around mean of data
x.frame(AutoSymRange(data)) ;
```

Draw a PDF on a frame – `RooAbsPdf::plotOn()`

Usage example: **`RooPlot* frame = pdf.plotOn(frame,…) ;`**

Plots (projects) the PDF on a specified frame. If a PDF is plotted in an empty frame, it will show a unit normalized curve in the frame variable, taken at the present value of other observables defined for this PDF.

If a PDF is plotted in a frame in which a dataset has already been plotted, it will show a projected curve integrated over all variables that were present in the shown dataset except for the one on the x-axis. The normalization of the curve will also be adjusted to the event count of the plotted dataset. An informational message will be printed for each projection step that is performed

This function takes the following named arguments

	Projection control
`Slice(const RooArgSet& set)`	Override default projection behavior by omitting observables listed in set from the projection, resulting a 'slice' plot. Slicing is usually only sensible in discrete observables
`Project(const RooArgSet& set)`	Override default projection behavior by projecting over observables given in set and complete ignoring the default projection behavior. Advanced use only.

`ProjWData(const RooAbsData& d)`	Override default projection *technique* (integration by default). For observables present in given dataset projection of PDF is achieved by constructing a Monte-Carlo summation of the curve for all observable values in given set. Consult manual sections 'Selecting arbitrarily shaped regions for plotting' (p.42) and 'Using conditional p.d.f.s for fitting, plotting and generating' (p.50) for details
`ProjWData(const RooArgSet& s, const RooAbsData& d)`	As above but only consider subset 's' of observables in dataset 'd' for projection through data averaging
`ProjectionRange(const char* rn)`	Override default range of projection integrals to a different range specified by given range name. This technique allows you to project a finite width slice in a real-valued observable

Miscellaneous content control

`Normalization(Double_t scale, ScaleType code)`	Adjust normalization by given scale factor. Interpretation of number depends on code: `Relative`: relative adjustment factor, `NumEvent`: scale to match given number of events.
`Name(const chat* name)`	Give curve specified name in frame. Useful if curve is to be referenced later
`Asymmetry(const RooCategory& c)`	Show the asymmetry of the PDF in given two-state category $(A^{+}-A^{-}) / (A^{+}+A^{-})$ rather than the PDF projection. Category must have two states with indices -1 and +1 or three states with indices -1,0 and +1.
`ShiftToZero(Bool_t flag)`	Shift entire curve such that lowest visible point is at exactly zero. Mostly useful when plotting -log(L) or χ^2 distributions
`AddTo(const char* name,Double_t wgtSelf, double_t wgtOther)`	Add constructed projection to already existing curve with given name and relative weight factors

Plotting control

`LineStyle(Int_t style)`	Select line style by ROOT line style code, default is solid
`LineColor(Int_t color)`	Select line color by ROOT color code, default is blue
`LineWidth(Int_t width)`	Select line with in pixels, default is 3
`FillStyle(Int_t style)`	Select fill style, default is not filled. If a filled style is selected, also use `VLines()` to add vertical downward lines at end of curve to ensure proper closure
`FillColor(Int_t color)`	Select fill color by ROOT color code
`Range(const char* name)`	Only draw curve in range defined by given name
`Range(double lo, double hi)`	Only draw curve in specified range
`VLines()`	Add vertical lines to y=0 at end points of curve
`Precision(Double_t eps)`	-- Control precision of drawn curve w.r.t to scale of plot, default is 1e-3. Higher precision will result in more and more densely spaced curve points

`Invisble(Bool_t flag)`	Add curve to frame, but do not display. Useful in combination `AddTo()`

Draw parameters of a PDF on a frame – `RooAbsPdf::paramOn()`

Usage example: `pdf.paramOn(frame,…)`

Add a box with parameter values (and errors) to the specified frame

The following named arguments are supported

`Parameters(const RooArgSet& param)`	Only the specified subset of parameters will be shown. By default all non-constant parameters are shown
`ShowConstant(Bool_t flag)`	Also display constant parameters
`Format(const char* optStr)`	Classic parameter formatting options, provided for backward compatibility
`Format(const char* what,...)`	Parameter formatting options, details are given below
`Label(const chat* label)`	Add header line with given label to parameter box
`Layout(Double_t xmin, Double_t xmax, Double_t ymax)`	Specify relative position of left, right side and top of box. Vertical size of box is calculated automatically from number lines in box

The `Format(const char* what,...)` has the following structure

`const char* what`	Controls what is shown. "N" adds name, "E" adds error, "A" shows asymmetric error, "U" shows unit, "H" hides the value
`FixedPrecision(int n)`	Controls precision, set fixed number of digits
`AutoPrecision(int n)`	Controls precision. Number of shown digits is calculated from error + n specified additional digits (1 is sensible default)

Example use: `pdf.paramOn(frame,Label("fit result"),Format("NEU",AutoPrecision(1)));`

Draw data on a frame – `RooAbsData::plotOn()`

Usage example: `data.plotOn(frame,…)`

Plots the dataset on the specified frame. By default an unbinned dataset will use the default binning of the target frame. A binned dataset will by default retain its intrinsic binning.

The following optional named arguments can be used to modify the default behavior

	Data representation options
`Asymmetry(const RooCategory& c)`	Show the asymmetry of the data in given two-state category $(A^+-A^-) / (A^++A^-)$. Category must have two states with

	indices -1 and +1 or three states with indices -1, 0 and +1.
`ErrorType(RooAbsData::EType)`	Select the type of error drawn: `Poisson` (default) draws asymmetric Poisson confidence intervals. `SumW2` draws symmetric sum-of-weights error
`Binning(double xlo, double xhi, int nbins)`	Use specified binning to draw dataset
`Binning(const RooAbsBinning&)`	Use specified binning to draw dataset
`Binning(const char* name)`	Use binning with specified name to draw dataset
`RefreshNorm(Bool_t flag)`	Force refreshing for PDF normalization information in frame. If set, any subsequent PDF will normalize to this dataset, even if it is not the first one added to the frame. By default only the 1st dataset added to a frame will update the normalization information

Histogram drawing options

`DrawOption(const char* opt)`	Select ROOT draw option for resulting `TGraph` object
`LineStyle(Int_t style)`	Select line style by ROOT line style code, default is solid
`LineColor(Int_t color)`	Select line color by ROOT color code, default is black
`LineWidth(Int_t width)`	Select line with in pixels, default is 3
`MarkerStyle(Int_t style)`	Select the ROOT marker style, default is 21
`MarkerColor(Int_t color)`	Select the ROOT marker color, default is black
`MarkerSize(Double_t size)`	Select the ROOT marker size
`XErrorSize(Double_t frac)`	Select size of X error bar as fraction of the bin width, default is 1

Misc. other options

`Name(const chat* name)`	Give curve specified name in frame. Useful if curve is to be referenced later
`Invisble(Bool_t flag)`	Add curve to frame, but do not display. Useful in combination `AddTo()`
`AddTo(const char* name, Double_t wgtSelf, Double_t wgtOther)`	Add constructed histogram to already existing histogram with given name and relative weight factors

Draw data statistics on a frame – `RooAbsData::statOn()`

Usage example: `data.statOn(frame,…)`

Add a box with statistics information to the specified frame. By default a box with the event count, mean and RMS of the plotted variable is added.

The following optional named arguments are accepted

`What(const char* whatstr)`	Controls what is printed: "N" = count, "M" is mean, "R" is RMS.
`Format(const char* optStr)`	Classic parameter formatting options, provided for backward compatibility
`Format(const char* what,...)`	Parameter formatting options, details given below
`Label(const chat* label)`	Add header label to parameter box
`Layout(Double_t xmin, Double_t xmax, Double_t ymax)`	Specify relative position of left, right side of box and top of box. Vertical size of the box is calculated automatically from number lines in box
`Cut(const char* expression)`	Apply given cut expression to data when calculating statistics.
`CutRange(const char* rangeName)`	Only consider events within given range when calculating statistics. Multiple `CutRange()` argument may be specified to combine ranges

The `Format(const char* what,...)` has the following structure

`const char* what`	Controls what is shown. "N" adds name, "E" adds error, A" shows asymmetric error, "U" shows unit, "H" hides the value
`FixedPrecision(int n)`	Controls precision, set fixed number of digits
`AutoPrecision(int n)`	Controls precision. Number of shown digits is calculated from error + n specified additional digits (1 is sensible default)
`VerbatimName(Bool_t flag)`	Put variable name in a `\verb+     +` clause.

Fill a 2D or 3D root histogram from a dataset – `RooAbsData::createHistogram()`

Usage example: `TH1* hist = data.createHistogram(name,xvar,…)`

Create and fill a ROOT histogram TH1,TH2 or TH3 with the values of this dataset.

This function accepts the following arguments

`const char* name`	Name of the ROOT histogram
`const RooAbsRealLValue& xvar`	Observable to be mapped on x axis of ROOT histogram

`Binning(const char* name)`	Apply binning with given name to x axis of histogram
`Binning(RooAbsBinning& binning)`	Apply specified binning to x axis of histogram
`Binning(double lo, double hi, int nbins)`	Apply specified binning to x axis of histogram
`YVar(const RooAbsRealLValue& var,...)`	Observable to be mapped on y axis of ROOT histogram
`ZVar(const RooAbsRealLValue& var,...)`	Observable to be mapped on z axis of ROOT histogram

The `YVar()` and `ZVar()` arguments can be supplied with optional `Binning()` arguments to control the binning of the Y and Z axes, e.g.

```
createHistogram("histo",x,Binning(-1,1,20),
                YVar(y,Binning(-1,1,30)), ZVar(z,Binning("zbinning")))
```

The caller takes ownership of the returned histogram

Fill a 2D or 3D root histogram from a PDF – `RooAbsReal::createHistogram()`

Usage example: `TH1* hist = pdf.createHistogram(name,xvar,…)`

Create and fill a ROOT histogram `TH1`, `TH2` or `TH3` with the values of this function.

This function accepts the following arguments

`const char* name`	Name of the ROOT histogram
`const RooAbsRealLValue& xvar`	Observable to be mapped on x axis of ROOT histogram
`Binning(const char* name)`	Apply binning with given name to x axis of histogram
`Binning(RooAbsBinning& binning)`	Apply specified binning to x axis of histogram
`Binning(double lo, double hi, int nbins)`	Apply specified binning to x axis of histogram
`ConditionalObservables(const RooArgSet& set)`	Do not normalized PDF over following observables when projecting PDF into histogram
`YVar(const RooAbsRealLValue& var,...)`	Observable to be mapped on y axis of ROOT histogram
`ZVar(const RooAbsRealLValue& var,...)`	Observable to be mapped on z axis of ROOT histogram

The `YVar()` and `ZVar()` arguments can be supplied with optional `Binning()` arguments to control the binning of the Y and Z axes, e.g.

```
 createHistogram("histo",x,Binning(-1,1,20),
                 YVar(y,Binning(-1,1,30)), ZVar(z,Binning("zbinning")))
```

The caller takes ownership of the returned histogram.

Fitting and generating

Fit a PDF to data – `RooAbsPdf::fitTo()`

Usage example: `pdf.fitTo(data,…)`

Fit PDF to given dataset. If dataset is unbinned, an unbinned maximum likelihood is performed. If the dataset is binned, a binned maximum likelihood is performed. By default the fit is executed through the MINUIT commands MIGRAD, HESSE and MINOS in succession.

The following named arguments are supported

Options to control construction of -log(L)

`ConditionalObservables(const RooArgSet& set)`	Do not normalize PDF over listed observables
`Extended(Bool_t flag)`	Add extended likelihood term, off by default
`Range(const char* name)`	Fit only data inside range with given name
`Range(Double_t lo, Double_t hi)`	Fit only data inside given range. A range named "fit" is created on the fly on all observables.
`NumCPU(int num)`	Parallelize NLL calculation on num CPUs
`Optimize(Bool_t flag)`	Activate constant term optimization (on by default)
`SplitRange(Bool_t flag)`	Use separate fit ranges in a simultaneous fit. Actual range name for each subsample is assumed to by `rangeName_{indexState}` where `indexState` is the state of the master index category of the simultaneous fit

Options to control flow of fit procedure

`InitialHesse(Bool_t flag)`	Flag controls if HESSE before MIGRAD as well, off by default
`Hesse(Bool_t flag)`	Flag controls if HESSE is run after MIGRAD, on by default
`Minos(Bool_t flag)`	Flag controls if MINOS is run after HESSE, on by default
`Minos(const RooArgSet& set)`	Only run MINOS on given subset of arguments
`Save(Bool_t flag)`	Flag controls if `RooFitResult` object is produced and returned, off by default
`Strategy(Int_t flag)`	Set MINUIT strategy (0 through 2, default is 1)

`FitOptions(const char* optStr)`	Steer fit with classic options string (for backward compatibility). Use of this option excludes use of any of the new style steering options

Options to control informational output

`Verbose(Bool_t flag)`	Flag controls if verbose output is printed (NLL, parameter changes during fit
`Timer(Bool_t flag)`	Time CPU and wall clock consumption of fit steps, off by default
`PrintLevel(Int_t level)`	Set MINUIT print level (1 through 3, default is 1). At 1 all RooFit informational messages are suppressed as well.

Print fit results as a LaTeX table – `RooAbsCollection::printLatex()`

Usage example: `paramList.printLatex(…) ;`

Output content of collection as LaTex table. By default a table with two columns is created: the left column contains the name of each variable, the right column the value.

The following optional named arguments can be used to modify the default behavior

`Columns(Int_t ncol)`	Fold table into multiple columns, i.e. ncol=3 will result in 3 x 2 = 6 total columns
`Sibling(const RooAbsCollection& other)`	Define sibling list. The sibling list is assumed to have objects with the same name in the same order. If this is not the case warnings will be printed. If a single sibling list is specified, 3 columns will be output: the (common) name, the value of this list and the value in the sibling list. Multiple sibling lists can be specified by repeating the `Sibling()` command.
`Format(const char* str)`	Classic format string, provided for backward compatibility
`Format(...)`	Formatting arguments, details are given below
`OutputFile(const char* fname)`	Send output to file with given name rather than standard output

The Format(const char* what,...) has the following structure

`const char* what`	Controls what is shown. "N" adds name, "E" adds error, "A" shows asymmetric error, "U" shows unit, "H" hides the value
`FixedPrecision(int n)`	Controls precision, set fixed number of digits
`AutoPrecision(int n)`	Controls precision. Number of shown digits is calculated from error + n specified additional digits (1 is sensible default)

VerbatimName(Bool_t flag)	Put variable name in a `\verb+ +` clause.

Example use:

```
list.printLatex(Columns(2), Format("NEU",AutoPrecision(1),VerbatimName()) ) ;
```

Generate toy Monte Carlo datasets – `RooAbsPdf::generate()`

Usage example: `RooDataSet* data = pdf.generate(x,…) ;`

Generate a new dataset containing the specified variables with events sampled from our distribution. Generate the specified number of events or expectedEvents() if not specified.

Any variables of this PDF that are not in whatVars will use their current values and be treated as fixed parameters. Returns zero in case of an error. The caller takes ownership of the returned dataset.

The following named arguments are supported

Verbose(Bool_t flag)	Print informational messages during event generation
NumEvent(int nevt)	Generate specified number of events
Extended()	The actual number of events generated will be sampled from a Poisson distribution with mu=nevt. For use with extended maximum likelihood fits
ProtoData(const RooDataSet& data, Bool_t randOrder)	Use specified dataset as prototype dataset. If randOrder is set to true the order of the events in the dataset will be read in a random order the order of the events in the dataset will be read in a random order number of events in the prototype dataset

If `ProtoData()` is used, the specified existing dataset as a prototype: the new dataset will contain the same number of events as the prototype (unless otherwise specified), and any prototype variables not in `whatVars` will be copied into the new dataset for each generated event and also used to set our PDF parameters.

The user can specify a number of events to generate that will override the default. The result is a copy of the prototype dataset with only variables in `whatVars` randomized. Variables in `whatVars` that are not in the prototype will be added as new columns to the generated dataset.

Create integrals of functions– `RooAbsReal::createIntegral()`

Usage example: `RooAbsReal* intOfFunc = func.createIntegral(x,…) ;`

Create an object that represents the integral of the function over one or more observables listed in iset

The actual integration calculation is only performed when the return object is evaluated. The name of the integral object is automatically constructed from the name of the input function, the variables it integrates and the range integrates over

The following named arguments are accepted

NormSet(const RooArgSet&)	Specify normalization set, mostly useful when working with PDFS

`NumIntConfig(const RooNumIntConfig&)`	Use given configuration for any numeric integration, if necessary
`Range(const char* name)`	Integrate only over given range. Multiple ranges may be specified by passing multiple Range() arguments

Automate fit studies – class `RooMCStudy`

Usage example: `RooMCStudy mgr(model,observables,…) ;`

Construct Monte Carlo Study Manager. This class automates generating data from a given PDF, fitting the PDF to that data and accumulating the fit statistics.

The constructor accepts the following arguments

`const RooAbsPdf& model`	The PDF to be studied
`const RooArgSet& observables`	The variables of the PDF to be considered the observables
`FitModel(const RooAbsPdf&)`	The PDF for fitting, if it is different from the PDF for generating
`ConditionalObservables(const RooArgSet& set)`	The set of observables that the PDF should *not* be normalized over
`Binned(Bool_t flag)`	Bin the dataset before fitting it. Speeds up fitting of large data samples
`FitOptions(const char*)`	Classic fit options, provided for backward compatibility
`FitOptions(....)`	Options to be used for fitting. All named arguments inside `FitOptions()`are passed to `RooAbsPdf::fitTo();`
`Verbose(Bool_t flag)`	Activate informational messages in event generation phase
`Extended(Bool_t flag)`	Determine number of events for each sample anew from a Poisson distribution
`ProtoData(const RooDataSet&, Bool_t randOrder)`	Prototype data for the event generation. If the `randOrder` flag is set, the order of the dataset will be re-randomized for each generation cycle to protect against systematic biases if the number of generated events does not exactly match the number of events in the prototype dataset at the cost of reduced precision with mu equal to the specified number of events

The `plotParam()` method plots the distribution of the fitted value of the given parameter on a newly created frame. This function accepts the following optional arguments

`FrameRange(double lo, double hi)`	Set range of frame to given specification
`FrameBins(int bins)`	Set default number of bins of frame to given number
`Frame(...)`	Pass supplied named arguments to

	`RooAbsRealLValue::frame()` function. See `frame()` function for list of allowed arguments

If no frame specifications are given, the `AutoRange()` feature will be used to set the range. Any other named argument is passed to the `RooAbsData::plotOn()` call. See that function for allowed options

The `plotPull()` method plots the distribution of pull values for the specified parameter on a newly created frame. If asymmetric errors are calculated in the fit (by MINOS) those will be used in the pull calculation This function accepts the following optional arguments

`FrameRange(double lo, double hi)`	Set range of frame to given specification
`FrameBins(int bins)`	Set default number of bins of frame to given number
`Frame(...)`	Pass supplied named arguments to `RooAbsRealLValue::frame()` function. See `frame()` function for list of allowed arguments
`FitGauss(Bool_t flag)`	Add a Gaussian fit to the frame

If no frame specifications are given, the `AutoSymRange()` feature will be used to set the range
Any other named argument is passed to the `RooAbsData::plotOn()` call. See that function for allowed options

Data manipulation

Reduce a dataset – `RooAbsData::reduce()`

Usage example: `RooAbsData* reducedData = data.reduce(…) ;`

Create a reduced copy of this dataset. The caller takes ownership of the returned dataset

The following optional named arguments are accepted

`SelectVars(const RooArgSet& vars)`	Only retain the listed observables in the output dataset
`Cut(const char* expression)`	Only retain event surviving the given cut expression
`Cut(const RooFormulaVar& expr)`	Only retain event surviving the given cut formula
`CutRange(const char* name)`	Only retain events inside range with given name. Multiple `CutRange` arguments may be given to select multiple ranges
`EventRange(int lo, int hi)`	Only retain events with given sequential event numbers
`Name(const char* name)`	Give specified name to output dataset
`Title(const char* name)`	Give specified title to output dataset

Appendix B – Selected statistical issues

This section is scheduled for the next version of the manual

Appendix C – RooFit class structure

This section is scheduled for the next version of the manual

General philosophy

Generic value objects – RooAbsArg

Real valued objects

Discrete valued objects

Datasets

Collections