RooFit

Wouter VERKERKE
Nikhef, Amsterdam

1 Introduction and Overview

One of the central challenges in performing a physics analysis is to accurately model the distributions of observable quantities $\vec{x}$ in terms of the physical parameters of interest $\vec{p}$ as well as other parameters $\vec{q}$ needed to describe detector effects such as resolution and efficiency. The resulting model consists of a "probability density function" (PDF)

$$F(\vec{x}\,;\vec{p},\vec{q})$$

that is normalized over the allowed range of the observables $\vec{x}$ with respect to the parameters $\vec{p}$ and $\vec{q}$. Experience in the BaBar experiment has demonstrated that the development of a suitable model, together with the tools needed to exploit it, is a frequent bottleneck of a physics analysis. For example, some analyses initially used binned fits to small samples to avoid the cost of developing an unbinned fit from scratch.

To address this problem, a general-purpose toolkit for physics analysis modeling was started in 1999. This project fills a gap in the particle physicists' tool kit that had not previously been addressed. The `RooFit` toolkit is integrated and distributed with the ROOT[1] analysis environment. Detailed documentation on the RooFit package can be found in the Users Manual[2] and various tutorials macros linked from the RooFit web home [3].

2 Object-oriented data modeling

To keep the distance between a physicists' mathematical description of a data model and its implementation as small as possible, the `RooFit` interface is styled after the language of mathematics. The object-oriented ROOT environment is ideally suited for this approach : each mathematical object is represented by a C++ software object. Table 1 illustrates the correspondence between some basic mathematical concepts and `RooFit` classes.

Concept	Math Symbol	`RooFit` class name
Variable	x, p	`RooRealVar`
Function	$f(\vec{x})$	`RooAbsReal`
PDF	$F(\vec{x};\vec{p},\vec{q})$	`RooAbsPdf`
Space point	$\vec{x}$	`RooArgSet`
Integral	$\int_{\vec{x}_{min}}^{\vec{x}_{max}} f(\vec{x})d\vec{x}$	`RooRealIntegal`
Derivative	dF/dx	`RooDerivative`
-log(Likelihood)	$-\sum_{data}\log(F(x_i,\vec{p})$	`RooNLLVar`
List of space points	$\vec{x}_k$	`RooAbsData`

Tab. 1: Correspondence between mathematical concepts and `RooFit` classes.

For example, a Gaussian probability density function with its variables is created as follows :

```
RooWorkspace w("w") ;
w.factory("Gaussian::g(x[-10,10],mean[0],sigma[3])") ;
```

The token `Gaussian::g(x[-10,10],mean[0],sigma[3])` is *not* interpreted as a mathematical expression itself, but as an expression to construct a series of objects that together represent a Gaussian probability density function and its parameters and observables. This distinction is important, as the 'constructor' approach allows to access and modify all created objects in the workspace at a later time for tuning and tailoring. It also allows construct more elaborate expression built on previously created objects in the workspace.

In terms of object created, the example expression above is equivalent to constructing the following objects 'by hand' :

```
RooRealVar x("x","x",-10,10) ;
RooRealVar m("m","mean",0) ;
RooRealVar s("s","sigma",3) ;
RooGaussian g("g","gauss(x,m,s)",x,m,s) ;
```

Each object, either created by hand or by the factory, has a name, and a title. The name serves as unique identifier of each object, the title can hold a more elaborate description of each object and only serves documentation purposes. All objects can be inspected with the universally implemented `Print()` method, which supports several verbosity levels.

In its default terse mode, output is limited to one line, in 'tree' mode, one line is printed per component object.

```
ROOT> g.Print() ;
RooGaussian::g[ x=x mean=m sigma=s ] = 1

ROOT> x.Print() ;
RooRealVar::x = 0  L(-10 - 10)

ROOT> g.Print("t") ;
0xbb4870 RooGaussian::g = 1 [Auto]
  0xba23c0/V- RooRealVar::x = 0
  0xba5620/V- RooRealVar::m = 0
  0xba5ff0/V- RooRealVar::s = 3
```

Object that represent variables, such as `RooRealVar` in the example above, store in addition to the value of that variable a series of associated properties, such as the allowed range, a binning specification and their role in fits (constant vs. floating), which serve as default values in many situations. Objects created by the factory can be inspected, and if necessary modified, in the same way by prefixing the name of the object with `w::`.

```
ROOT> w::m=-2 ;
ROOT> w::g.Print("t") ;
...
```

Function objects are linked to their ingredients : the function object `g` *always* reflects the values of its input variables `x`,`m` and `s`. The absence of any explicit invocation of calculation methods allows for true symbolic manipulation in mathematical style.

3 Creating basic probability density functions

We will now explore the basics of the workspace for maintaining analysis projects, describe what basic p.d.f shapes are available, how new ones can be created and how existing ones can be tailored.

3.1 Workspace basics

A workspace is a container class that is intended to contain all data modeling components of an analysis project : data, (probability density) functions, parameters and observables. An empty workspace is created as follows

```
RooWorkspace w("w",kTRUE) ;
```

If the second argument of the constructor is `kTRUE`, the workspace will also create a C++ namespace in the CINT interpreter with the same name as the workspace, and populate that namespace with references to the contents of the workspace. This feature allows to access all workspace contents through the namespace in a type-safe way. There are two ways to populate a workspace with contents. The first one is the `import()` method, which imports a copy of an existing RooFit object, and all its subsidiaries into the workspace. The other way is to use the factory to create objects directly in the workspace :

```
w.factory("Gaussian::g(x[-10,10],mean[0],sigma[3])") ;
```

The syntax rules for factory construction of basic p.d.f.s and variables are as follows

name refers to an existing object with that name in the workspace

name[value] creates an initially constant `RooRealVar`

name[min,max] creates a `RooRealVar` with allowed values in the range [min,max].

name[val,min,max] is similar with an explicitly provided initial value

classname : :objname(var,var,...) creates an object of class `classname` with instance name `objname` with the given arguments. The `Roo` prefix on the class name may be dropped for brevity. The order and interpretation of the provided arguments is defined by the 3rd through n-th constructor arguments of the class. An automatic name and title are inserted as 1st and 2nd constructor arguments.

classname(var,var,...) creates an object of class classname with automatically chosen instance name. This is mostly useful for shorthand construction of intermediate functions objects in complex expression.

{a,b,c} refers to a `RooArgSet` or `RooArgList` of given arguments (dependent on context)

The syntax elements defined above can be used recursively, e.g. a variable can be defined in the place where it is expected as input argument for the definition of a p.d.f. (as was already done the Gaussian example above). The examples below illustrate some other uses of the factory syntax.

```
// Create second Gaussian using previously created x and mean variables
w.factory("Gaussian::g2(x,mean,sigma2[5])") ;

// Create a Chebychev polynomial (requires list argument)
w.factory("Chebychev::ch(x,{a0[-1,1],a1[-1,1],a2[-1,1]})") ;
```

3.2 Standard and generic functions

The RooFit distribution comes with a collection of standard p.d.f. shapes. The most frequently used ones include

- `RooGaussian(x,mean,sigma)` - Gaussian p.d.f.
- `RooExponential(x,alpha)` - Exponential decay p.d.f.
- `RooPolynomial(x,{a0,a1,a2...})` - Polynomial p.d.f.
- `RooChebychev(x,{t0,t1,t2,...})` - Chebychev polynomial p.d.f.
- `RooBreitWigner(x,mean,width)` - Non-relativistic Breit-Wigner p.d.f.
- `RooLandau(x,mean,width)` - Landau p.d.f.
- `RooCBShape(x,x0,sigma,alpha,n)` - Crystal ball p.d.f.

- `RooPoisson(x,mu)` - Poisson p.d.f.
- `RooArgusBG` - Argus phase space p.d.f.
- `RooHistPdf` - Histogram-based shape p.d.f. (with optional interpolation)
- `RooKeysPdf` - Kernel estimation p.d.f.

All of these p.d.f. can be adjusted in terms of reparameterized input arguments, as well as through addition, multiplication and convolution as will be shown in the next sections. It is also possible to define a new p.d.f. class on the fly from a formula expression that is explicitly normalized through (numeric) integration. The factory provides two interfaces : `EXPR` and `CEXPR`.

```
w.factory(  "EXPR::mypdf('exp(x*y+a)-b*x',x[0,10],y[0,10],a[3],b[5])") ;
w.factory("CEXPR::mypdf2('exp(x*y+a)-b*x',x[0,10],y[0,10],a[3],b[5])") ;
```

The difference is that the `EXPR` interface creates an interpreted `RooGenericPdf` that is based on a ROOT `TFormula` interpreter engine, while the `CEXPR` interface create a custom `RooFit` p.d.f class that is compiled, linked and instantiated on the spot. The trade off is a difference in initialization versus execution speed : interpreted p.d.f. are created nearly instantly, but are slower in evaluation, whereas custom compiled p.d.f. need a few seconds to instantiate, but are as fast as built-in p.d.f. in evaluation.

Along similar lines, generic *functions* can be defined with the `expr` and `cexpr` interfaces. Such functions are most useful to define on-the-fly transformations of p.d.f. parameters and observables.

```
w.factory("expr::mean('a0*y+a1',y[-10,10],a0[1.03],a1[2.7,0,10])") ;
w.factory("Gaussian::g(x[-10,10],mean,s[3])") ;
```

Here we have defined a two-dimensional p.d.f. $G(x, \mu = (a_0 \cdot y + a_1), \sigma)$ by tailoring the built-in `RooGaussian` p.d.f. class. Such customization operations are generically possible on *all* RooFit p.d.f. classes, including user-defined classes, and do not require explicit handling of such scenarios in the p.d.f. class itself. You can equivalently write the same p.d.f. in a single line of factory code as follows

```
w.factory("Gaussian::g(x[-10,10],expr('a0*y+a1',y[-10,10],a0[1.03],a1[2.7,0,10]),s[3])") ;
```

4 Fitting, Plotting and Toy Generation from p.d.f.s

Once a p.d.f. is created (in a workspace or outside) we can use it for fitting, plotting and toy event generation. To plot a p.d.f, first a plot frame must be created that represents a view of a given observable. To plot a Gaussian p.d.f. versus the observable x, one does

```
// Make Gaussian p.d.f.
w.factory("Gaussian::g("x[-10,10],m[-10,10],s[3,0.1,10])") ;

// Make plot frame in x
RooPlot* frame = w::x.frame() ;

// Plot p.d.f. g on x
w::g.plotOn(frame) ;

// Draw frame on canvas
frame->Draw() ;
```

The resulting plot is shown in Figure 1. To fit this p.d.f. to data, the data needs to be first available in the form of a RooFit binned or unbinned dataset. Unbinned data can be imported from a ROOT TTree as follows

```
RooDataSet data("data","my data",w::x,myTree) ;
```

where `myTree` is a ROOT `TTree` with a branch named `x`. Binned data can be imported from a ROOT `TH1` as follows

```
RooDataHist data("data","my data",w::x,myHist) ;
```

Both data types inherit from a common abstract data type `RooAbsData`, that is accepted by `RooAbsPdf::fitTo()`. An (un)binned maximum likelihood fit of the p.d.f. `g` to data (of either type) is then performed as

```
w::g.fitTo(data) ;
```

The effect of the `fitTo()` operation is that the parameter objects of `g` have their values changed to the fitted values and that errors are associated with these values :

```
w::m.Print() ;
RooRealVar::m = 0.08544 +/- 0.0987391  L(-10 - 10)

w::s.Print() ;
RooRealVar::s = 3.10016 +/- 0.0723132  L(0.1 - 10)
```

A subsequent plot of the p.d.f. will reflect these new values

```
RooPlot* frame = w::x.frame() ;
data.plotOn(frame) ;
w::g.plotOn(frame) ;

// Add text box to frame with parameter values
w::g.paramOn(frame,data) ;

frame->Draw() ;
```

The resulting plot is shown in Figure 2. Optionally, the `fitTo()` method can return a `RooFitResult` object which provides additional information on the fit, such as the covariance and correlation matrix between the parameters, the MINUIT status code, the covariance matrix quality code, the estimated distance to the minimum etc. To request that information, add the `Save()` option to `fitTo()`

```
ROOT> RooFitResult* r = w::g.fitTo(data,Save()) ;
ROOT> r->Print() ;

 RooFitResult: minimized FCN value: 2542.02, estimated distance to minimum: 4.12535e-07
               covariance matrix quality: Full, accurate covariance matrix

   Floating Parameter    FinalValue +/-  Error
 --------------------  --------------------------
                   m    8.5440e-02 +/-  9.87e-02
                   s    3.1002e+00 +/-  7.23e-02

ROOT> r->correlationMatrix().Print() ;

2x2 matrix is as follows

     |      0    |      1    |
--------------------------------
   0 |          1    0.002938
   1 |   0.002938           1
```

With help of this additional information on the correlations between the parameters it is also possible to visualize the uncertainty on the p.d.f. shape in the plot

```
// Plot p.d.f. g on x
w::g.plotOn(frame,VisualizeError(*r)) ;
```

The resulting plot is shown in Figure 3, where a smaller data sample was used to enlarge the error for illustration clarity.

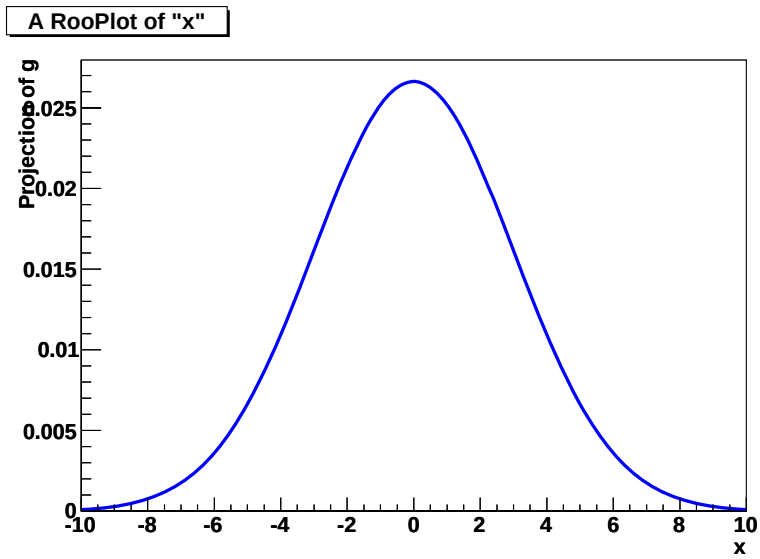

Fig. 1: A Unit normalized Gaussian p.d.f projected on a `RooPlot`

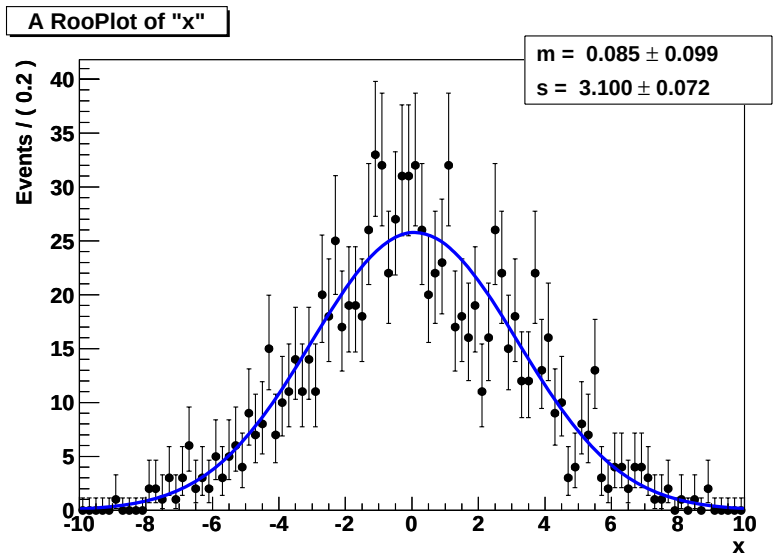

Fig. 2: Gaussian p.d.f. projected over a dataset. The normalization is adjusted to the event count of the dataset

5 Adding shapes into composite models

Most realistic data description models are sum of multiple components, e.g. signal and background. Mathematically the sum of two probability density functions is also a normalized probability density function as along as the coefficients add up to 1, e.g.

$$S(x) = f \cdot F(x) + (1 - f) \cdot G(x),$$

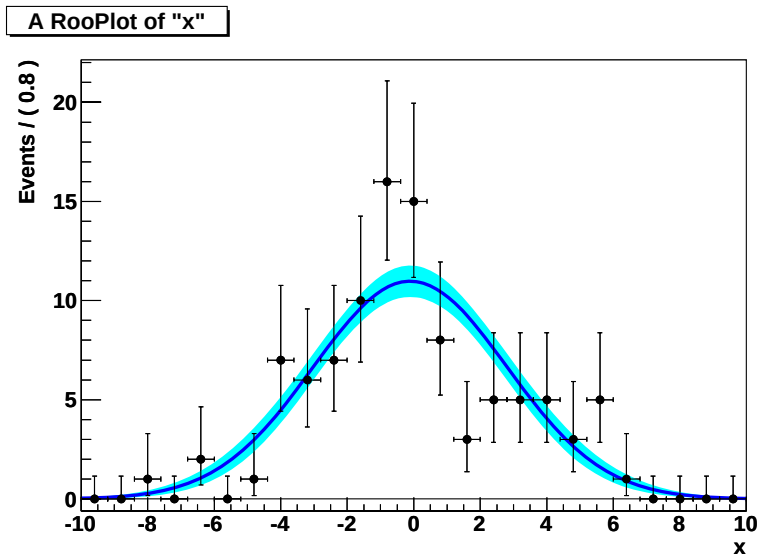

Fig. 3: Visualization of p.d.f. uncertainty propagated from the covariance matrix of a fit to the data

or generically for N components :

$$S(x) = c_0 \cdot F_0(x) + c_1 \cdot F_1(x) + ... + c_{n-1} \cdot F_{n-1}(x) + \left(1 - \sum_{i=0}^{n-1} c_i\right) F_n(x) \tag{1}$$

In RooFit such summed p.d.f.s are represented by class `RooAddPdf`, which are created by the factory operator `SUM()`

```
// Build two Gaussian p.d.f.s. and an ARGUS shape
w->factory("Gaussian::gauss1(x[0,10],mean1[2],sigma[1])") ;
w->factory("Gaussian::gauss2(x,mean2[3],sigma)") ;
w->factory("ArgusBG::argus(x,x0[9.0],kappa[-1]") ;

// Now add them together using the SUM operator
w->factory("SUM::model(g1frac[0.5]*gauss1, g2frac[0.1]*gauss2, argus") ;
```

The components of such summed p.d.f.s can be addressed individually in plotting, e.g.

```
RooPlot* frame = w::x.frame() ;
w::model.plotOn(frame) ;

// Plot argus component only
w::model.plotOn(frame,Components(w::argus),LineStyle(kDashed)) ;
```

Components in plotting can be specified with an component object reference, as done above, a `RooArgSet()` of component references to specify multiple components, or with a string with the component name or names (separated by commas). Figure 4 shows the plot that results from the above example.

5.1 Recursive fractions

If a composite p.d.f. consists of more than two components, there is no simple way in the formalism of Eq. 1 to constrain the fraction parameters such that the sum of the fractions is always less than one. If the sum of these coefficients becomes larger than one, the remainder coefficient will be assigned a negative fraction. As long as the summed p.d.f. is greater than zero everywhere, this is not ill-defined, but may pose some problems in the interpretation. An alternative is to write the sum of p.d.f.s recursively :

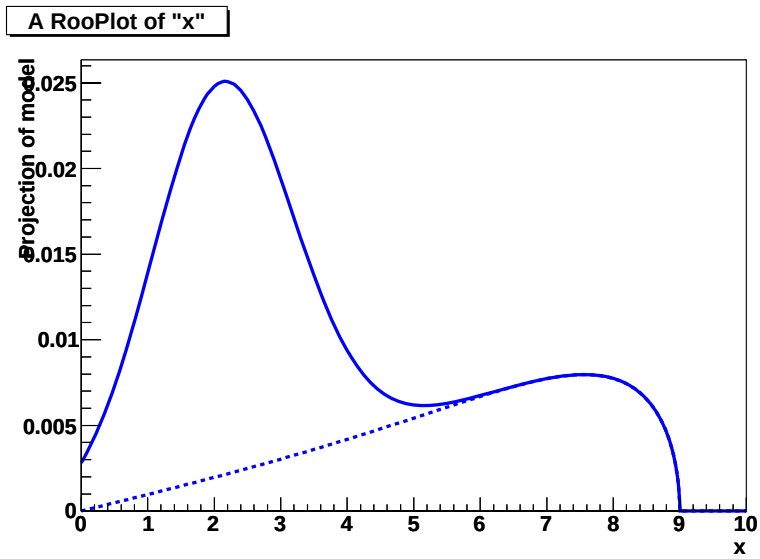

Fig. 4: Visualization of components of a composite p.d.f.

$$
\begin{aligned}
S_2(x) &= f \cdot F_1(x) + (1-f) \cdot F_2(x) \\
S_3(x) &= f_1 \cdot F_1(x) + (1-f_1) \cdot (f_2 \cdot F_2(x) + (1-f_2) \cdot F_3(x)) \\
S_4(x) &= f_1 \cdot F_1(x) + (1-f_1) \cdot (f_2 \cdot F_2(x) + (1-f_2) \cdot (f_3 \cdot F_3(x) + (1-f_3) \cdot F_4(x)))
\end{aligned}
$$

Recursive sum definitions like these can be constructed with the factory operator `RSUM()`

```
w->factory("RSUM::model(g1frac*gauss1, g2frac*gauss2, argus") ;
```

The difference between this p.d.f and the equivalent one with `SUM()` is that here `g2frac` now represents the *relative* fraction of `gauss2` w.r.t. `gauss2+argus`, rather than the absolute fraction of `gauss2` w.r.t. `gauss+gauss2+argus`.

5.2 Extended maximum likelihood and summed p.d.f.s

Composite p.d.f.s are often used in conjunction with the extended maximum likelihood formalism. In the extended likelihood formalism, a term

$$-\log(\text{Poisson}(N_{obs}, N_{exp}))$$

is added to the regular likelihood that allows to estimate a parameter that represents the number of events in the sample, N_{exp}. For composite models it is practical to rearrange the parameters as

$$\begin{pmatrix} f_{sig} \\ N_{exp} \end{pmatrix} \rightarrow \begin{pmatrix} N_{sig} = f_{sig} \cdot N_{ex} \\ N_{bkg} = (1 - f_{sig}) \cdot N_{exp} \end{pmatrix},$$

so that the extended ML procedure estimates the number of signal and background events rather than a signal fraction and a total number of events. Composite models of this form can be constructed with the `SUM()` operator by multiplying the last p.d.f. with a coefficient term

```
w->factory("SUM::model(NG1[0,1000]*gauss1, NG2[0,1000]*gauss2, NG3[0,1000]*argus") ;
```

A `fitTo()` operation on a sum p.d.f. with equal number of p.d.f.s and coefficients will automatically included the extended term in the likelihood.

6 Convolution

Data models that describe physics distributions smeared by an experimental (detector) resolution can be modeled with convolution of a p.d.f. describing the physics and a p.d.f. describing the experimental resolution.

$$M(x, \vec{p}, \vec{q}) = T(x, \vec{p}) \otimes R(x, \vec{q})$$

If the two p.d.f.s are sufficiently different in shape, it may be possible to extract both the physics parameters of interest as well as the parameters of the smearing model from the data. Most practical difficulties arise in the calculation of the convolution integral

$$M(x, \vec{p}, \vec{q}) = \int_{-\infty}^{\infty} T(x, \vec{p}) \cdot R(x - x', \vec{q}) dx'$$

which is analytically calculable only for selected combinations of R an T. In addition, for a properly normalized p.d.f on a finite domain of x, the function M needs to be explicitly divided by a normalization integral

$$\int_{x_{min}}^{x_{max}} \int_{-\infty}^{\infty} T(x, \vec{p}) \cdot R(x - x', \vec{q}) dx' dx$$

6.1 Comparison of convolution calculation strategies

RooFit provides three strategies for the convolution of p.d.f.s, each with their own set of advantages and disadvantages.

- Analytical formulation for selected p.d.f.s (mostly related to B-physics),
- Numeric convolution using Fourier Transforms
- Brute-force calculation of convolution integral

Analytical formulation's big advantages are speed and precision, but are only available for select choices of T and R. Numeric convolution using Fourier Transforms is also quite fast, works for all choices of T and R, but requires sampling of both p.d.f.s at a finite resolution and may present some special complications for choices of R and T that do not vanish at the observable domain boundaries. Brute force calculation finally is generally very slow, especially at the required precision for MINUIT minimization, but can calculate convolutions for all R and T.

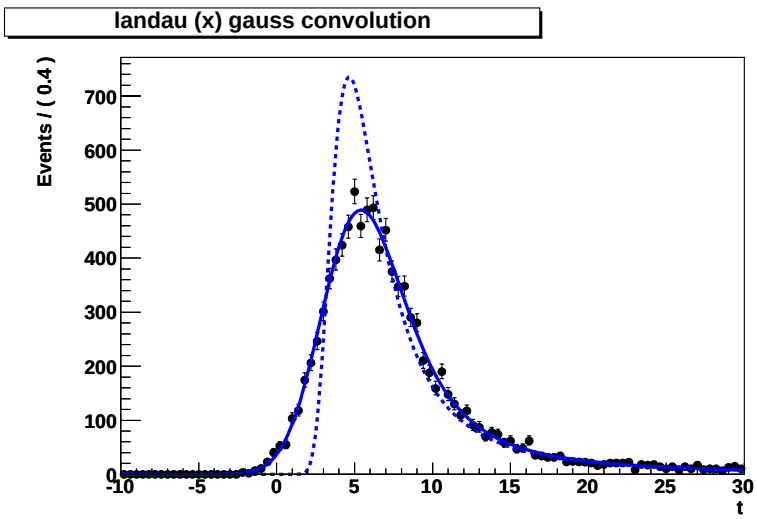

Fig. 5: P.d.f. of a convolution of a Landau with a Gaussian fitted to data and overlaid on that data(solid). The dashed line shows the unconvoluted Landau shape.

6.2 Numeric convolution

Both the FFT-based and brute-force numeric convolution calculators are implemented in the general RooFit operator p.d.f. form and can be easily instantiated through the workspace factory with the operators `FCONV` and `NCONV` respectively :

```
// Make T(x) and R(x)
w.factory("Landau::T("x[-10,30],m_t[5,-20,20],s_t[1,0.1,10])") ;
w.factory("Gaussian::R("x,m_r[0],s_r[2,0.1,10])") ;

// Request fine binning for FFT sampling
w::x.setBins(10000,"cache") ;

// Make FFT convolution M(x) = T(x) (*) R(x)
w.factory("FCONV::M(x,T,R)") ;

// Make brute-force numeric convolution M(x) = T(x) (*) R(x)
w.factory("NCONV::N(x,T,R)") ;
```

Figure 5 shows the above p.d.f. M fitted to a dataset of 10000 events, along with the unconvoluted p.d.f. T (dashed). This fit with three floating parameters required 84 evaluations of the likelihood, took 8 seconds, and demonstrates the power of the FFT algorithm : each likelihood evaluation with 10000 events was executed in roughly 100 milliseconds. By comparison, a single evaluation the likelihood constructed with `NCONV` takes roughly 7 seconds, about two order of magnitude slower that the FFT algorithm.

6.2.1 Special issues with Fourier convolution

While the computational performance of FFT-based convolution is superior to brute-force numeric calculation, it makes some approximations that are acceptable in many, but not all use cases.
Finite sampling resolution and range. Since discrete Fourier transforms are used, both T and R are sampled at a finite resolution over a finite range. Care should be taken that the sampling resolution is sufficiently high (through `w::x.setBins()` as shown above), and that the range is sufficiently wide. The default strategy of `RooFFTConvPdf` is to sample the physics model T over the defined range of the observable and the resolution model R over symmetric range around zero with the same width as the range for T. If the resolution model does not fit in this range, the convolution is effectively calculated with a truncated resolution function.
Circular effects from convolution theorem The convolution theorem, used to calculate the convolution in Fourier space, treats both T and R as cyclical in the convolution observable. Such cyclical effects will be apparent in the final convolution M if the input p.d.f. T does not decrease to zero at the observable boundaries. To decrease the effect of this spillover on output p.d.f. it is possible to perform the convolution in a range wider than the nominal observable range and then truncate the convoluted p.d.f to the nominal range, cutting away the regions most effected by the spillover. The size of this spillover buffer is 10% of the observable range by default, but can be controlled through the `RooFFTConvPdf::setBufferFraction()` method. Figure 6 demonstrates the effect on modified version of the Landau $\otimes$ Gaussian example with a buffer fraction of zero (dotted), 10% (dashed) and 50% (solid). For inherently circular observables like angles, circular convolution is in fact the appropriate solution, and for such observables the buffer fraction should be set to zero.

6.3 Analytical convolution

For selected p.d.f.s RooFit provides an interface to analytically calculated integrals. These p.d.f. must be of the form

$$P(x,...) = \sum_k c_k(...)\,(f_k(x,...) \otimes R(x,...))$$

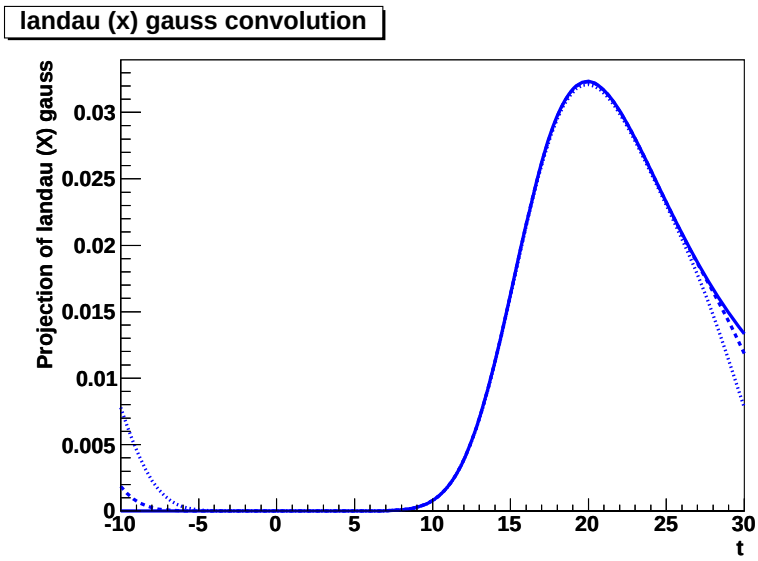

Fig. 6: Cyclical spillover effects of Fourier convolutions when untreated (dotted), with a 10% spillover buffer (dashed) and a 50% spillover buffer(solid)

For example, a B^0 meson decay with $B^0/\bar{B}^0$ mixing from an $\Upsilon(4s)$ decay at a B-factory can be written in this form with

$$\begin{array}{ll} c_0 = 1 \pm \Delta w & f_0 = e^{-|t|/\tau} \\ c_1 = \pm(1-2w) & f_1 = e^{-|t|/\tau} \cdot \cos(\Delta m \cdot t) \end{array}$$

Such p.d.f.s are implemented in two parts : a physics class inheriting from base class `RooAbsAnaConvPdf` that implement coefficients c_k and declare the associated basis functions f_k, and a resolution model inheriting from `RooResolutionModel` that implement $f_k(x,...) \otimes R(x,...)$. This separation allows the combinations of physics and resolution to be chosen at run time. Since these `RooAbsAnaConvPdf` derived objects are not functional without a resolution model, an operator form like `FCONV` to construct such convolutions is not appropriate. Instead the resolution model must be specified in the constructor.

The example below illustrates the procedure for class `RooDecay`, which implements a plain decay function, with a variety of resolution models

```
// Convolution of Decay with delta function
w.factory("Decay::decay_tm(t[-20,20],tau[1.548],TruthModel(t),DoubleSided)") ;

// Convolution of Decay with Gaussian resolution
w.factory("Decay::decay_gm1(t,tau,GaussModel::gm1(t,bias[0],resol1[1]),DoubleSided)") ;

// Convolution of Decay with double-Gaussian resolution
w.factory("Decay::decay_gm2(t,tau,
           AddModel({gm1,GaussModel::gm2(t,bias,resol2[5])},frac[0.5]),DoubleSided)") ;

RooPlot* frame = w::t.frame(-10,10) ;
w::decay_tm.plotOn(frame) ;
w::decay_gm1.plotOn(frame,LineStyle(kDashed)) ;
w::decay_gm2.plotOn(frame,LineStyle(kDotted)) ;
frame->Draw()
```

Figure 7 shows the about of this example.

7 Building Multi-Dimensional Model

While many data analysis and modeling problems involve more than one observable, multi-dimensional models are less frequently used because of difficulties in formulating such models. In RooFit, the construction of multi-dimensional models is straightforward and several strategies are discussed to construct these.

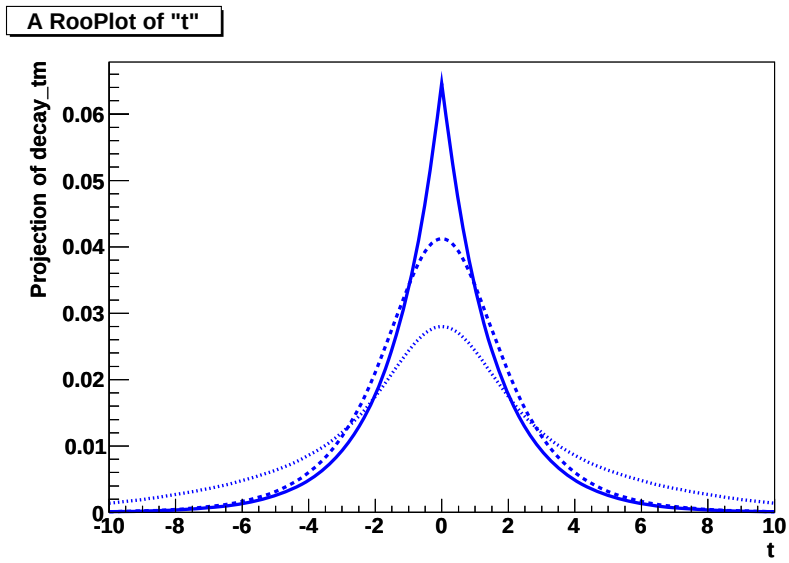

Fig. 7: Analytical convolution of decay function with delta-function (solid), Gaussian (dashed) and double Gaussian resolution model (dotted)

7.1 Constructing N-dimensional models without correlations

The simplest way to construct a multi-dimensional model is to multiply two or more one-dimensional p.d.f.s. Mathematically this is straightforward :

$$H(x,y) = F(x) \cdot G(y) \qquad\qquad H(\vec{x}) = \prod_i F_i(x_i)$$

It is easy to see that if $F(x)$ and $G(x)$ are normalized p.d.f.s, that $H(x,y)$ is also normalized by construction. This is important as the normalization of a N-dimensional p.d.f, which is generally a difficult problem, is here reduced to a series of lower dimensional integrals.

The RooFit class to implement these other products (discussed later) is `RooProdPdf`, and is created with the factory operator `PROD()`

```
w.factory("Gaussian::gx(x[-10,10],mx[-2,-10,10],sx[3,0.1,10])") ;
w.factory("Gaussian::gy(y[-10,10],my[4,-10,10],sy[1,0.1,10])") ;

w.factory("PROD::gxy(gx,gy)") ;
```

7.2 Using N-dimensional models

Using multi-dimensional models in RooFit is not really different from using one-dimensional models, as is illustrated by an example below that generates toy data from the above 2-dimensional Gaussian, fits the p.d.f. to that data

```
// Generate a toy dataset D(x,y)
RooDataSet* data = w::gxy.generate(RooArgSet(w::x,w::y),10000) ;

// Fit p.d.f G(x,y) to D(x,y)
w::gxy.fitTo(*data) ;
```

Most of the differences are in plotting as a two-dimensional models can be visualized in multiple ways. Below we create two one-dimensional plots with projections on the x and y axis overlaid on data, as well as a two-dimensional plot of the p.d.f.

```
// Plot projections of p.d.f. and data on x and y
RooPlot* framex = w::x.frame() ;
data->plotOn(framex) ; w::gxy->plotOn(framex) ;
```

```
RooPlot* framey = w::y.frame() ;
data->plotOn(framey) ; w::gxy->plotOn(framey) ;

// Make 2D plot of p.d.f
TH2* hh_pdf = w::gxy.createHistogram("x,y") ;
hh_pdf->Draw("surf") ;
```

If a multi-dimensional dataset is added to a one-dimensional `RooPlot` frame, all observables of that dataset are memorized and any p.d.f. that is subsequently plotted on that same frame will automatically be integrated over those observables so that the projection of `gxy` on x that is shown is $\int g_{xy}(x,y)dy$ and the projection of `gxy` on y is $\int g_{xy}(x,y)dx$. Figure 8 shows the output of this example.

The product p.d.f operator class `RooProdPdf` automatically recognizes the factorizing structure of its product and and will analytically deduce that these integrals can be resolved to $g_x(x)$ and $g_y(y)$ respectively.

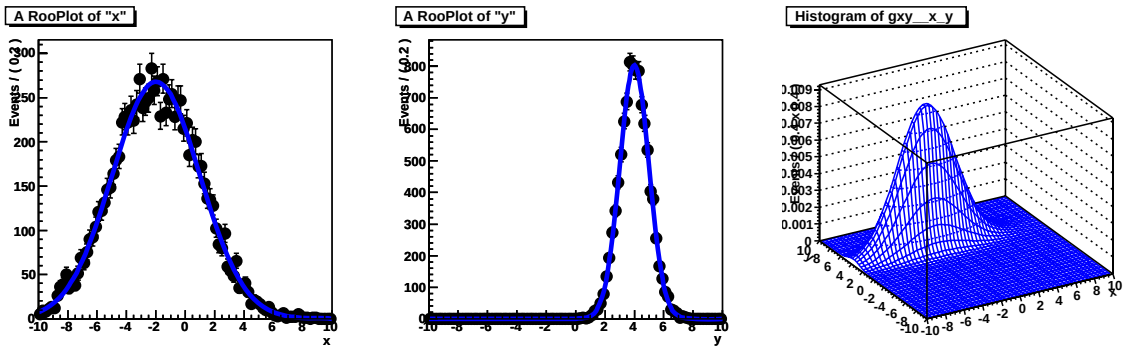

Fig. 8: Visualization of 2-dimensional Gaussian p.d.f. $G(x,y) = G(x) \cdot G(y)$: Projection on x (left), Projection on y(middle) and two-dimensional view(right)

7.3 Constructing N-dimensional models with correlations

The easiest way to introduce correlations in a multi-dimensional model is to start with a one-dimensional model and replace one of the parameters with a function that depends on a second observable, e.g.

$$F(x;p) \rightarrow F(x,p(y,q)) \rightarrow F(x,y,q)$$

This approach maps to many practical problems. For example, the reconstruct mass m is modeled by a Gaussian $S(m,m_0,\sigma)$ for the signal component, but the mass peak turns out to have a slight bias depending on the value on another observable y. In that case the data model can be improved by accounting for that y-dependent bias explicitly in the model $S'(x,m_0(m_0^{true},y,\vec{p}),\sigma)$.

In RooFit the transformation of one-dimensional models into multi-dimensional models with this technique is performed exactly as formulated mathematically

```
w.factory("Gaussian::g(x[-10,10],expr::mx('m0+my*y',y[-10,10],m0[-0.5],m1[3]),sx[3])") ;
```

The above p.d.f g, when used as a two-dimensional p.d.f. in x,y does however also make an explicit prediction for the distribution of events in y. In the particular case of the above model $g(x,y)$, it comes out to a flat distribution in y, which can be seen by plotting the p.d.f. vs y as shown in Figure 9

If the distribution in y is unknown or irrelevant, as in the physics example above, and one cares only bout the distribution in x *given* a value of y, the p.d.f should be used as conditional p.d.f. $g(x|y)$ rather than as two-dimensional p.d.f. $g(x,y)$. The difference between these two forms is in the normalization

$$\int g(x,y)dxdy \equiv 1 \qquad \int g(x,y)dx \equiv 1 \text{ for each value of } y$$

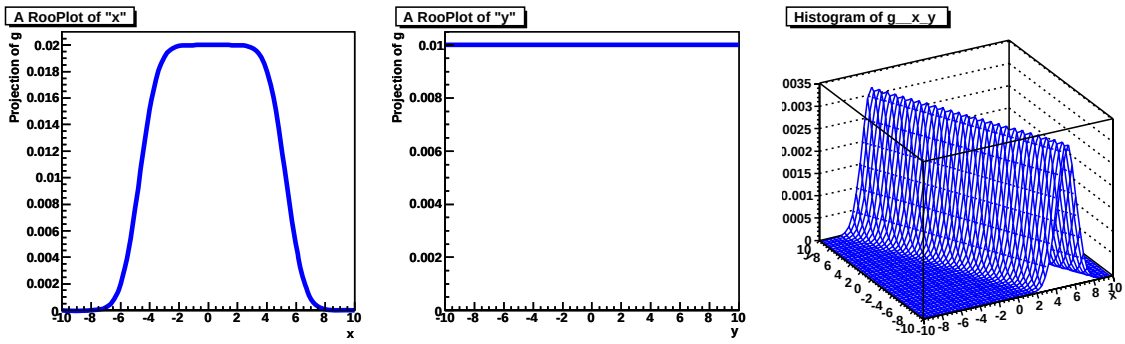

Fig. 9: Visualization of Gaussian p.d.f with a shifting mean $G(x,y) = G(x, f(y), \sigma)$: Projection on x (left), Projection on y(middle) and two-dimensional view(right)

The object `gxy` created above can be used in either way. To use it as conditional p.d.f. the observables that are to be taken conditional must be specified

```
// Fit g(x|y) to a dataset D(x,y)
gxy.fitTo(data,ConditionalObservables(y)) ;
```

Conditional p.d.f.s require some extra input in toy event generation and plotting as these operation require knowledge of the distribution of the data in the conditional observable, which is (by construction) not provided by a conditional p.d.f, thus these need to be supplied externally

```
// Generate dataset D(x,y) from g(x|y) and D(y)
RooDataSet* dataxy = w::gxy.generate(w::y,ProtoData(datay)) ;

// Project distribution of g(x|y) on x
RooPlot* framex = w::x.frame() ;
gxy.plotOn(framex,ProjectWData(w::y,datay)) ;
```

For the latter projection, the usual integral $\int g(x,y)dy$ is replaced by $1/N_{data} \sum_{data} g(x,y_i)$.

Aside from these practical issues of conditional p.d.f.s, there is also a more fundamental problem, as pointed out by Punzi [4] : if you take the sum of two conditional p.d.f.s

$$f \cdot S(x|y) + (1-f) \cdot B(x|y)$$

your fit results may be incorrect if the true distribution of S and B differ in the conditional observable y, as the model has no way of accounting for that. For this reason, and the aforementioned practical reasons, it is usually better to multiply a conditional p.d.f. with another p.d.f. modeling the conditional observable

$$H(x,y) = G(x|y) \cdot F(y)$$

A conditional product of this form is normalized by construction, if $G(x|y)$ and $F(y)$ are. The 'conditional product' form is both easy to write and to easy use. Here we extend the original example with gxy to a conditional product form

```
w::factory("PROD::gxy2(gxy|y,Exponential::fy(y,slope[-1]))") ;
```

Figure 10 shows the above conditional product p.d.f. in various projections.

7.4 Per-event errors, a physics example of conditional product p.d.f

A common application of conditional p.d.f.s in physics is the incorporation of 'per-event' errors in a data model. Often a measured observable, like the B meson decay time, consists of a value (a flight time

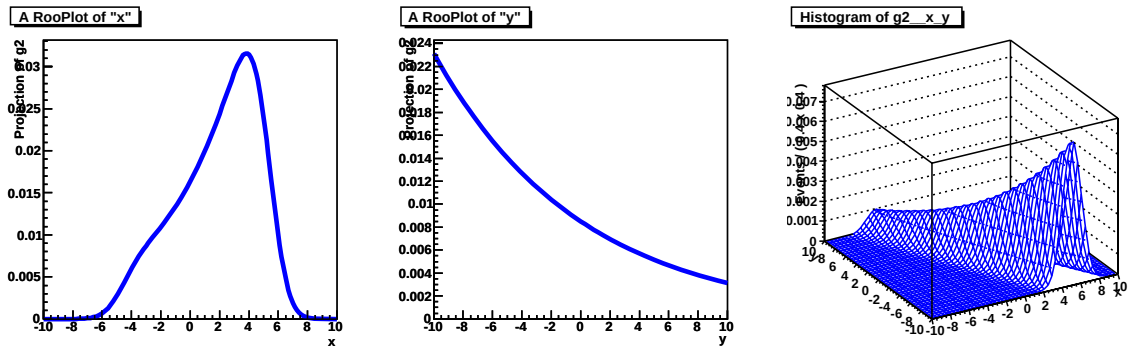

Fig. 10: Visualization of 2-dimensional model $H(x,y) = G(x|y) \cdot F(y)$: Projection on x (left), Projection on y(middle) and two-dimensional view(right)

t) and an associated measurement error on that time (δt, as e.g. provided by the vertexing algorithm used to construct the B meson decay point). The decay distribution of B mesons can to first order be described with an exponential decay function convoluted with Gaussian resolution model

$$F(t) = D(t,\tau) \otimes R(t,\mu,\sigma)$$

But in this form the difference in quality of the measurement of each event, is not taken into account. Instead, one can rewrite $F(t)$ as

$$F(t|\delta t) = D(t,\tau) \otimes R(t,\mu,\delta t \cdot \sigma')$$

in which case the experimental resolution of the model is adjusted to each event, based on the provided per-event error. This enhanced model has potentially greater statistical sensitivity as the information on the per-event error is also used. The parameter σ' of this modified model serves as an overall scale correction to the per-event error in the data. If these error estimates are correct, the fitted value of σ' will be consistent with one. If σ' comes out larger than one, the per-event errors underestimated the true error. Note that even when σ' is not consistent with one, this model can correctly describe the data, as long as the true error is proportional by a constant scale factor to the provided per-event error.

Since $F(t|\delta t)$ does in all likelihood not describe the distribution of δt in the data it must be used as a conditional p.d.f, or as conditional p.d.f. multiplied with an second p.d.f. describing the distribution of the per-event errors. If the event sample contains also background events, and if it cannot be assumed that these background events have the same per-event error distribution as the signal events, a conditional product form should be used to avoid biases due to differences in per-event error distributions

$$M(t,\delta t) = f_{sig} \cdot S_1(t|\delta_t) \cdot S_2(\delta t) + (1 - f_{sig}) \cdot B_1(t|\delta_t) \cdot B_2(\delta t)$$

8 Data modeling with discrete observables

All data models described so far involved continuous (real-valued) observables and parameters. In this section we explore a number of common analysis problems involving discrete observables

8.1 Fitting for efficiencies

A common analysis problem is the parameterization and fitting of efficiencies as function of one or more observables. The simplest approach taken is to take an efficiency function $f(x)$ and perform a χ^2 fit to a histogram $H(x)$ with efficiency measurements calculated from the ratio of accepted to all events in each bin. The errors on these bins are usually calculated from a symmetricised binomial error formula, $\sigma(\epsilon) = \sqrt{\epsilon(1-\epsilon))/n}$, or the equivalent (asymmetric) interval as obtained from e.g. a Feldman-Cousins type method. In either case, such a fit suffers from approximations made in the error calculation and/or

approximations made in the χ^2 formalism in general.

It is possible however, to perform a simple unbinned maximum likelihood fit for the efficiency that does not suffer from these approximations, if one sees this as problem with two observables rather than one : each event is described by a continuous observable x and a discrete observable c with two possible states *accept* and *reject*. The probability density function corresponding to the efficiency measurement is

$$E(c|x,\vec{\alpha}) = \delta(c = \text{accept}) \cdot \epsilon(x,\vec{\alpha}) + \delta(c = \text{reject}) \cdot (1 - \epsilon(x,\vec{\alpha}))$$

where $\epsilon(x,\vec{\alpha})$ is the efficiency function to be fitted. The model $E(c|x)$ should be taken as conditional in x as the model does not aim to describe the distribution of x itself, but merely the distribution in c for each value of x. In RooFit the efficiency pdf should above is coded as follows

```
w.factory("expr::epsilon('(1-a)+a*cos((x-c)/b)',a[0.4,0,1],b[5],c[-1,-10,10])") ;
w.factory("Efficiency::E(epsilon,cut[accept,reject],'accept')") ;
```

The factory token `cut[accept,reject]` creates a discrete RooFit variable of the type `RooCategory` that has finite set of states labeled by a name and optionally an index (e.g. `cut[accept=0,reject=1]`). Given a binned or unbinned dataset $D(x,c)$, the efficiency $\epsilon(x)$ is fit and visualized as follows

```
w::E.fitTo(data,ConditionalObservables(w::x)) ;

RooPlot* frame = w::x.frame() ;
data.plotOn(frame,Efficiency(w::c)) ;
epsilon.plotOn(frame) ;
```

Here the `Efficiency(cut)` argument passed to the `plotOn()` call for data modifies the default behavior : instead of plotting the distribution in x it plots the efficiency of c=`accept` in bins of x. In the efficiency mode, the error bars reflect 68% binomial confidence intervals instead of Poisson intervals. Figure 11 shows the output of this plot (on the right), as well as the x distribution of all (black) and accepted (red) events (on the left).

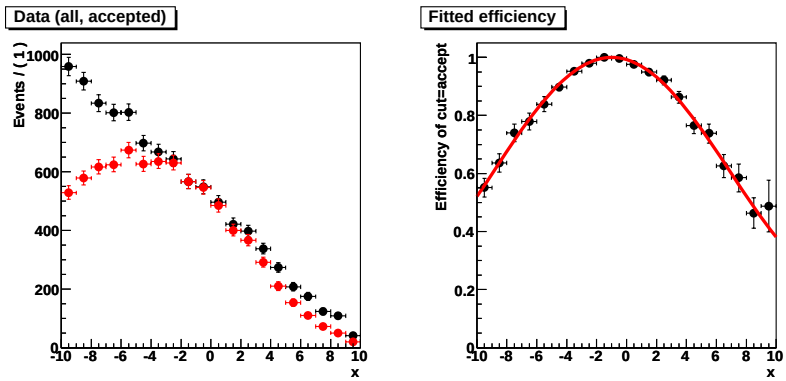

Fig. 11: Dataset with encoded efficiency in discrete observable. Left : Distribution of all events in x (black) and subset of accepted events (red). Right : Acceptance efficiency in data overlaid with fitted efficiency from model $E(c|x)$

8.2 Joint fits

Discrete observables can also be used to label data from various sources into a unified dataset and to allow joint fit fit to those samples.

8.2.1 Definition of joint likelihood

The general concept of a joint fit is easiest explained at the likelihood level. Given two models/data pairs

$$F_1(x;a,b), D_1(x) \quad \text{and} \quad F_2(x,a,c), D_2(x)$$

two likelihoods $L_1(a,b)$ and $L_2(a,c)$ can be constructed that can be used to estimate the parameters a,b and a,c respectively. This approach results in two measurements of a to be combined a posteriori. Instead one can perform a joint fit by minimizing the sum of the $-\log$ likelihoods.

$$-\log(L_{1,2}(a,b,c)) = -\log(L_1(a,b)) + -\log(L_2(a,c)) \tag{2}$$

This joint estimate results in a more precise estimation of the common parameter a as well as a full and correct error propagation between all three parameters.

8.2.2 Definition of joint p.d.f.

Such a joint fit can also be described at the p.d.f level by first constructing a joint dataset $D(x,k)$ that contains events from both $D_1(x)$ and $D_2(x)$ where a discrete observable $k(1,2)$ labels the origin of each event. A joint p.d.f $F_{12}(x,k;a,b,c)$ is then construct as

$$F_{12}(x,k;a,b,c) = \delta(k-1)F_1(x;a,b) + \delta(k-2)F_2(x,a,c),$$

which, when combined with the dataset $D(x,k)$ results again in the likelihood of Eq. 2. The advantage of the formulation of a joint p.d.f. over a joint likelihood is that it remains possible to describe all modeling problems at the p.d.f. level.

8.3 Application and construction of joint fits

Joint fits have two general classes of use. The first use case is a fit to dissimilar samples that measure a common physics parameter α of interest. Examples of such fits are combined fits for the Higgs mass from different Higgs decays channels (α is the Higgs mass), or joint a fit to signal and control/sideband region in the data. In the latter case $\vec{\alpha}$ are usually properties of the background shape that are better constrained in the control region than in the signal region, but needed in the signal region for a better determination of the signal component.

Joint models of the first type, defined from two existing p.d.f.s are represented by class `RooSimultaneous` and are created by the `SIM()` factory operator

```
// Make p.d.f.s F(x) for samples A,B
w.factory("Gaussian::pdfA(x[-10,10],mean[-10,10],sigmaA[3,0.1,10]") ;
w.factory("BreitWigner::pdfB(x,mean,sigmaB[3,0.1,10]") ;

// Make joint p.d.f. F(x,k)
w.factory("SIM::model(k[sampleA,sampleB],sampleA=pdfA,sampleB=pdfB)") ;
```

A joint dataset $D(x,k)$ to be used with `w::model` can be created as follows

```
RooDataSet data12("data12","joint data",Index(w::k),
                  Import("sampleA",data1),Import("sampleB",data2)) ;
```

The second use is a fit to *similar* samples that have slight variations in sample properties. Example of such fits are the measurement of the CP violation parameter $\sin 2\beta$ in different bins of flavor tagging purity, or measurement of e.g. a single-leptonic top cross section in different bins of b-tagging probability. These samples have the same observables and can generally all be fit by the same shape, and their division in subsamples of different purity serves solely to enhance the statistical precision of the measurement by exploiting these differences in purity.

Joint fits of the second type can be made in the same way as joint fits of the first type, e.g.

```
// Make p.d.f.s for samples A,B,C,D,E,F
w.factory("Gaussian::pdfA(x[-10,10],mean[-10,10],sigmaA[3,0.1,10]") ;
w.factory("Gaussian::pdfB(x,mean,sigmaA[3,0.1,10]") ;
w.factory("Gaussian::pdfC(x,mean,sigmaB[3,0.1,10]") ;
```

```
w.factory("Gaussian::pdfD(x,mean,sigmaC[3,0.1,10]") ;
w.factory("Gaussian::pdfE(x,mean,sigmaD[3,0.1,10]") ;
w.factory("Gaussian::pdfF(x,mean,sigmaE[3,0.1,10]") ;

// Make joint p.d.f.
w.factory("SIM::model(k[sampleA,sampleB,sampleC,sampleD,sampleE,sampleF],"
                     "sampleA=pdfA,sampleB=pdfB,sampleC=pdfC,
                      sampleD=pdfD,sampleE=pdfE,sampleF=pdfF)") ;
```

but it is often impractical, resulting in repetitive code, as all p.d.f.s are similar in construction. It is possible to request construction of such p.d.f. in a more compact form using the dedicated `SIMCLONE()` factory operator. The code fragment below produces the same simultaneous p.d.f. as the example above, but is much shorter.

```
w.factory("Gaussian::pdf(x[-10,10],mean[-10,10],sigma[3,0.1,10])") ;
w.factory("k[sampleA,sampleB,sampleC,sampleD]") ;
w.factory("SIMCLONE::model(pdf,\$SplitParam(sigma,k)") ;
```

The `SIMCLONE()` operator takes a prototype pdf, `pdf` in this case and a customization prescription `SplitParam(sigma,k)`, which specifies that the p.d.f that is associated with each state of `k` must have its own copy of parameter `sigma`, named `sigma_A` trough `sigma_F` (thus `sigma` is 'split' over `k`). It is possible to split any number of parameters in a category, use multiple splitting categories, and to split any given variable by more than one category.

In case the subdivision of the data represent by the index `k` is not fundamental – i.e. it arises from a binning of a continuous observable in the data such as a b-tagging probability, rather than choice of a flavor tagging algorithm of which there exist a fixed and finite number – the process of defining the index category `k` and may be further automated. Given a dataset with two observables $D(x, y)$ we can create an index category `k` that represents bins of y as follows

```
// Define binning in y
w::y.setBinning(20,"kbins") ;

// Create a real-to-category function kfunc(y) that maps the value of y to a 'kbin' number
w.factory("BinningCategory::kfunc(w::y,'kbins')") ;

// Add an observable 'k' to dataset 'data' with a value calculated by 'kfunc(y)'
w.factory("dataobs::k(data,kfunc)") ;
```

9 Practical aspects of Maximum Likelihood estimation

So far, we have focused on the formulation and construction of probability density functions. This section will focus on practical aspects and problems that may arise ML estimation of non-trivial models.

9.1 Creating a likelihood from a pdf and data

While `pdf.fitTo(data)` command performs all aspects of fitting automatically, it is sometimes useful to perform the required steps by hand, for greater control over the minimization, or to be able to visualize likelihoods in the process for analysis an debugging purposes. The standard execution of a fit operation can be replaced by the following few lines of code, which construct the likelihood function, minimize the likelihood function and perform the error analysis :

```
// Construct likelihood function L(p)
RooAbsReal* nll = pdf.createNLL(data) ;
```

```
// Instantiate (MINUIT) minimizer operating on this likelihood
RooMinimizer m(*nll) ;

// Minimize the likelihood (estimate parameter values)
m.migrad()

// Perform default error analysis (estimate parameter uncertainties using d2L/dp^2)
m.hesse() ;
```

The `createNLL()` method accepts both binned and unbinned datasets and returns a corresponding likelihood. The returned likelihood function is a regular RooFit function object and can be plotted as function of its variables (the p.d.f. parameters) in the usual way

```
RooPlot* frame = param.frame() ;
nll->plotOn(frame) ;
```

The `createNLL()` method accepts a number of optional arguments to modify the constructed likelihood. These arguments are the same as can be passed to `fitTo()`. The most important ones are

ConditionalObservables(x) Specify that `x` should be treated as a conditional observable

NumCPU(n) Request that likelihood calculated is parallelized over 8 processors.

Verbose() Request that extra information is printed during minimization process

9.1.1 Automatic optimizations in the likelihood calculation

As likelihoods are computationally intensive, a number of automatic optimizations is applied in RooFit likelihood objects when used in `RooMinimizer` minimizations

- Component p.d.f.s that have exclusively constant parameters, are precalculated
- Dataset variables that are not used by the p.d.f. are dropped
- P.d.f normalization integrals are only recalculated when the value of p.d.f parameters change (i.e. not for every data point, unless the p.d.f. has a conditional observable)
- In simultaneous fits, on the part of the likelihood that depend on a changed parameter are recalculated

The applicability of all techniques is re-evaluated for each fit, ensuring maximal optimization potential for each fit. In typical complex fits (e.g. the 35 parameters, 5 observable BaBar fit for the CP violation parameter $\sin 2\beta$ a factor 3 to 10 speedup is not uncommon.

9.2 Understanding and using the MINUIT minimizer in RooFit

The default minimization and error analysis package used by RooFit is ROOT implementation of the MINUIT[5] package. The `RooMinimizer` interface class takes care propagating information from `RooFit` variables and function from and to MINUITs internal representation, and interfaces its high level operations for minimization and error analysis. The most important of these are

MIGRAD Find function (likelihood or χ^2) minimum by alternatingly calculating the gradient w.r.t the parameters and following that gradient to the (local) minimum. The number of likelihood evaluations needed to do this depend strongly on the number of floating parameters, the shape of the likelihood and the initial distance to the minimum.

HESSE Calculates the second derivative w.r.t. all parameters at the minimum and estimates a (symmetric) error for each parameter by assuming that the likelihood is locally parabolic ($\hat{\sigma}(p)^2 = \hat{V}(p) = \left(\frac{d^2 \log L}{dp^2}\right)^{-1}$. Requires approximately $N_{par}^2/2$ likelihood evaluations.

MINOS Calculate (potentially asymmetric) errors by finding a contour in likelihood defined by $L = L_{min} + 0.5$. The errors are then defined the (hyper)box that fits tightly around that likelihood contour. The number of likelihood evaluations can be prohibitively expensive for a large number of floating parameters ($> 10 - 20$)

CONTOUR Find and visualize contours of $L = L_{min} + X$ in two parameters. Mostly an interactive tool

9.3 Choosing the correct starting point for minimization

For all but the most trivial problems, there is no guarantee that MINUIT will find the correct (global) minimum as its search strategy will stop at the first minimum that it finds in its search path through parameter space. To increase your chances of proper convergence, it is important to provide reasonable initial estimates for the parameters to be fitted. It may also be beneficial to supply reasonable initial stepping sizes in the parameters so that MINUIT scan the likelihood (at least initially) at an appropriate granularity. The default initial step size provide by RooFit to MINUIT is one tenth of the range of each parameters. You can override this default by specifying an initial error on the fit parameters through `RooRealVar::setError()`.

9.4 Understanding difference between MINOS and HESSE errors

The MINOS and HESSE algorithms to estimate parameter errors are quite different, as explained above. If the likelihood is (approximately) parabolic in shape close to the estimated minimum of the parameter, both algorithms will report a similar error. If MINOS and HESSE error come out substantially different, this implies the likelihood not asymptotically parabolic at the minimum and care should be taken in the interpretation of the errors. It may be worthwhile to plot the likelihood explicitly as function of the parameters, as shown earlier in this section, to understand the problem. Figure 12 shows a somewhat pathological likelihood configuration that results in very different error estimates between HESSE and MINOS.

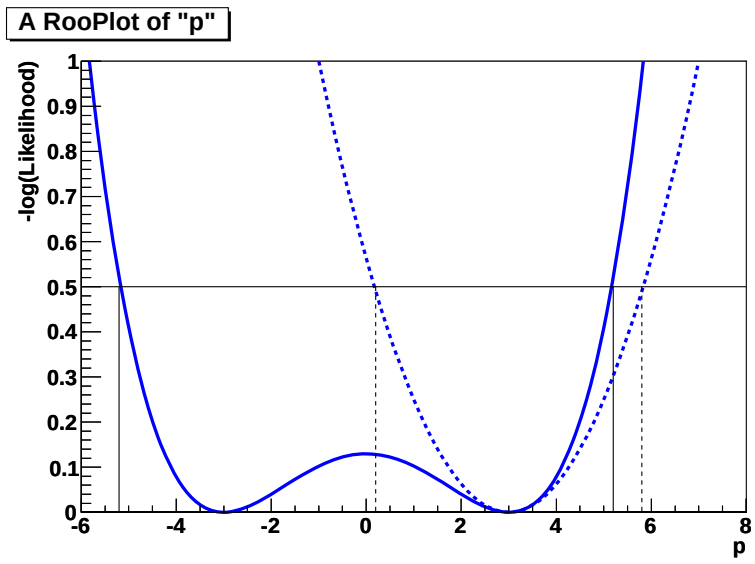

Fig. 12: Example of a non-parabolic likelihood that gives different estimates for the error on p : The MINOS estimate is defined by the intersection of the likelihood curves (solid) with $\Delta(L) = 0.5$, the HESSE estimate is defined by the intersection of the parabolic approximation around the minimum (dashed) with $\Delta(L) = 0.5$

9.5 Problems in the likelihood calculation

Sometimes, in the course of a minimization, the likelihood cannot be evaluated due to an error condition. The two most probable causes for such errors are

- The probability density function evaluates to zero or a negative value for one or more data points, resulting in an infinite or undefined likelihood contribution from that point.

– The normalization integral of a p.d.f evaluates to zero, or has numerical convergences problems.

If such errors occur during a minimization sessions, these errors are collected and summarized in a structured way for each likelihood evaluation for the user to analyze

```
[#0] WARNING:Minimization -- RooFitGlue: Minimized function has error status.
Returning maximum FCN so far (-2417.08) to force MIGRAD to back out of this region.
Parameter values: k=-36.7074, m0=5.2901
RooArgusBG::argus[ m=m m0=m0 c=k p=0.5 ]
     getLogVal() top-level p.d.f evaluates to zero or negative number @ m=m=5.2901, m0=m0=5.290,
                                                                        c=k=-36.7074, p=0.5=0.5
```

At the same time, a very high value of the likelihood is returned to encourage MINUITS MIGRAD algorithm to retreat from this region of parameters space. Note that it is *not* OK to simply drop problematic events from the likelihood, as this can create false minima. This is illustrated in figure 13 showing an ARGUS phase space background model being fitted with a floating end point parameter (left). Since the ARGUS p.d.f has an discontinuity at the end point value, it is quite prone to likelihood evaluation error : If the end point parameter is moved to too low values, one more events will be assigned a zero probability by this p.d.f. and errors will occur.

The middle and right plots show the likelihood as function of the end point parameter in case problematic events are dropped (middle) and in case a 'wall' is put up (right).

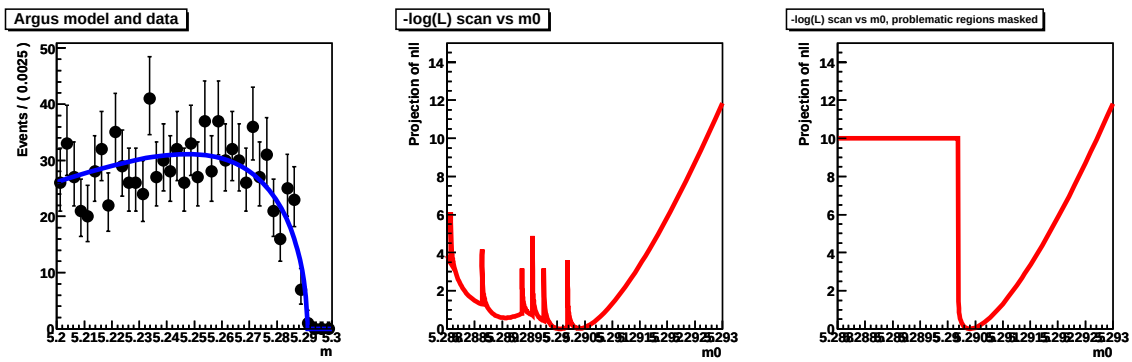

Fig. 13: Left : ARGUS probability density function overlaid on data. Middle : Likelihood as function of end point parameter when events with zero probability are dropped. Right : Likelihood as function of end point parameter with 'wall' as presented by RooFit to MINUIT.

If such evaluation errors occur once or twice, MIGRAD may successfully recover from this, but if there are many regions in parameter space that cause problem MIGRAD will terminate with failure. Most of these problems can be fixed constraining parameter values to exclude problematic regions. For example, do not allow Gaussian distributions to become arbitrarily small, as any point evaluated at 50 σ will evaluate to zero due to numerical precision. Also take care of degrees of freedom in polynomial-based p.d.f.s that can easily go negative if left unchecked.

9.6 Mitigating fit stability problems

Sometimes fits do not converge properly, even if there are no evaluation errors in the likelihood. Often these problems are caused by strong correlations between parameters, which make the minimization problem difficult to solve numerically. The primary tool to investigate such correlation problems is correlation matrix provided by HESSE.

To illustrate this problem and its solution we look at an example problem : Below we fit a sum of two Gaussians to data. The widths of the two Gaussians have been chosen very similar.

```
w->factory("Gaussian::g1(x[-20,20],mean[-10,10],sigma_g1[3])") ;
w->factory("Gaussian::g2(x,mean,sigma_g2[3.1,2,6])") ;
```

```
w->factory("SUM::model(frac[0,1]*g1,g2)") ;

RooDataSet* data = w::model.generate(w::x,1000) ;
RooFitResult* r = w::model.fitTo(*data,Save()) ;
```

Inspecting the returned fit result we see

```
RooFitResult: minimized FCN value: 2533.87, estimated distance to minimum: 0.000261785
              covariance matrix quality: Full matrix, but forced positive-definite

  Floating Parameter    FinalValue +/-  Error
--------------------  --------------------------
               frac    5.5907e-01 +/-  7.12e-01
               mean    1.4846e-02 +/-  9.68e-02
            sigma_g1    2.8392e+00 +/-  9.54e-01
            sigma_g2    3.2973e+00 +/-  1.08e+00
```

A closer look at the correlation matrix, reveals the source of the problem :

```
ROOT> fitresult_model_modelData->correlationMatrix().Print()

4x4 matrix is as follows

     |      0    |      1    |      2    |      3    |
-----------------------------------------------------------
   0 |          1   -0.006309      0.9656      0.9664
   1 |  -0.006309           1     0.02252    -0.03403
   2 |     0.9656     0.02252           1      0.8746
   3 |     0.9664    -0.03403      0.8746           1
```

very strong correlations between the `frac`, `sigma_g1` and `sigma_g2` parameters. There are two strategies to solves such problems : the first and simplest one is to fix of the parameters involved in the strong correlations. This will eliminate the (almost) redundant degree of freedom and restore the stability of the fit. An alternative strategy is to reparameterize the model so that it results in less correlated parameters, for example by choosing `sigma_g2 = ratio * sigma_g1`

```
w->factory("Gaussian::g2(x,mean,prod(ratio[1.05,0.5,2],sigma_g1))") ;
```

This second approach keeps the original degrees of freedom of the model, but is not guaranteed to reduce the problematic correlations sufficiently for the fit to become stable. Some trial and error is usually needed.

Problematic correlations are particularly common between parameters of regular polynomials, and regular polynomials should for that reason be avoided as models A good substitute are Chebychev polynomials, in which terms have been rearranged to reduce correlations between coefficients. Chebychev polynomials of the first type are implemented in class `RooChebychev`.

9.7 Profile likelihood and MINOS error estimation

We conclude this section with a short introduction on the concept of profile likelihood and its relation to MINOS-type errors.

Given a likelihood $L(a, \vec{b})$ where a is our parameter of interest (e.g. our physics observable) and $\vec{b}$ are other parameters (background and signal shape parameters etc, often called 'nuisance' parameters), we can define the profile likelihood as

$$PL(a) = \frac{L(a, \hat{\vec{b}})}{L(\hat{a}, \hat{\vec{b}})}$$

where $\hat{a}$ refers to the estimated value of a. Thus the profile likelihood at a given value a is the likelihood $L(a, \vec{b})$ minimized w.r.t to all parameters $\vec{b}$ divided by the value of the likelihood in the global minimum.

Profile likelihood can be used to estimate uncertainties on physics parameters of interest in the presence of a large number of 'nuisance' parameters, which are eliminated by the profiling technique. The uncertainty on the parameter of interest is simply estimated from the interval in the profile likelihood, where $\Delta(PL) = 0.5$ (for a 'one-sigma' equivalent error). The error estimated by the profile likelihood method is identical to that estimated by the MINOS method, but is computationally (much) more efficient if there are many nuisance parameters.

The concept of profile likelihood errors and their relation to MINOS errors are best illustrated by a 2-parameter example. For such a 2-parameter given likelihood, it is instructive to start out with a plot comparing profile likelihood versus the parameter of interest a versus the plain likelihood $L(a, b)$. Starting with the double-Gaussian example of the preceding section, but taking `sigma_g2` and `mean` constant at 4 and 0 respectively, we make such a plot versus the remaining floating parameters `frac` and `sigma_g1`

```
// Create L(frac,sigma_g1), PL(frac) and PL(sigma_g1)
RooAbsReal* nll = w::model.createNLL(*data) ;
RooAbsReal* pll1 = nll->createProfile(w::frac) ;
RooAbsReal* pll2 = nll->createProfile(w::sigma_g1) ;

// Plot L(frac,sigma_g1) and PL(frac) vs frac
RooPlot* frame1 = w::frac.frame() ;
nll->plotOn(frame1,LineStyle(kDashed),ShiftToZero()) ;
pll1->plotOn(frame1) ;

// Plot L(frac,sigma_g1) and PL(sigma_g1) vs sigma_g1
RooPlot* frame2 = w::sigma_g1.frame() ;
nll->plotOn(frame2,LineStyle(kDashed),ShiftToZero()) ;
pll2->plotOn(frame2) ;

// Plot contour at DeltaL=0.5,2.0 in (frac,sigma_g1)
RooMinuit m(*nll) ;
RooPlot* frame3 = m.contour(w::frac,w::sigma1) ;
```

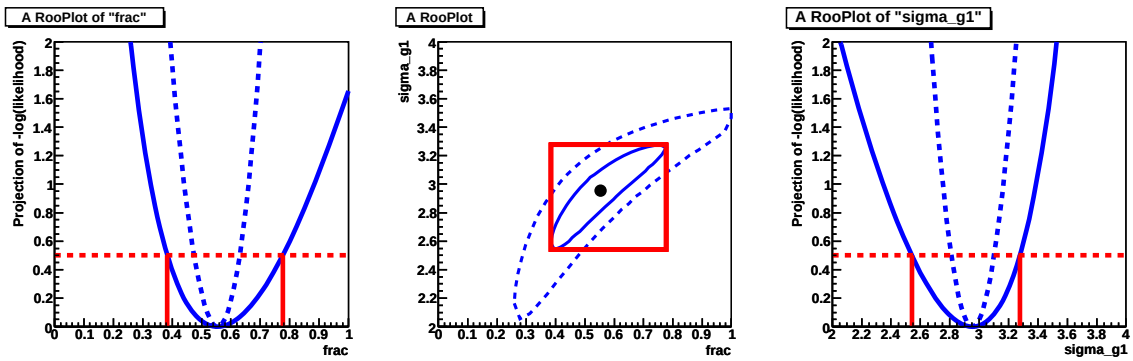

Fig. 14: Left : Profile likelihood (solid) and likelihood ratio (dashed) as function of `frac`. Right : Profile likelihood (solid) and likelihood ratio (dashed) as function of `sigma_g1`. Middle : Contours in delta-likelihood at 0.5 (solid) and 2(dashed). The box represent the MINOS errors.

.

The three plots that result from this code fragment are shown in Figure 14 The profile likelihood curves are always wider that the likelihood curve in the one-dimensional projections, because the profile represents the minimum of the likelihood for each value of the plotted parameter of interest w.r.t. the other nuisance parameters. In the best case, one was already at the minimum, in which case the likelihood value and

the profile likelihood value are identical, but one usually finds a lower minimum, resulting in a wider valley. The profile likelihood curve is usually not parabolic, especially at more than 1-2 sigma away from the minimum. This reinforces the point that parabolic (Gaussian) approximations are usually not valid beyond 1-2 'sigma'.

Figure 14 also shows the errors calculated by MINOS, defined by the box around the $\Delta(L) = 0.5$ contour in the 2-d plane. These are also also indicated in the 1-D projections. The horizontal dashed line at $\Delta(L) = 0.5$ in both 1-d projection demonstrate that the MINOS errors are identical to errors that would be obtained from an interval in the profile likelihood a $\Delta(L) = 0.5$. This identity hold in general, also in more than 2 dimensions, but the proof of the identity is not discussed here.

Références

[1] `http://root.cern.ch`

[2] W. Verkerke, D.Kirkby, , *The RooFit Users Manual 2.91*, available at [1]

[3] `http ://root.cern.ch/drupal/content/roofit`

[4] G. Punzi, *Comments on likelihood fits with variable resolution*, `physics/0401045`

[5] F. James, *MINUIT Function Minimization and Error Analysis, Reference Manual*, CERN Program Library Long Writeup D506