BaBar Analysis Document #18, Version 12

A User's Guide to the RooFitTools Package for Unbinned Maximum Likelihood Fitting

John Back (QMW), Chih-Hsiang Cheng (Stanford), Ulrik Egede (RAL),
David Kirkby*(Stanford), Gerhard Raven (UC San Diego),
Chris Roat (Stanford), Alessio Sarti (Rome), Marco Serra (Rome),
Abi Soffer (Colorado State), Jan Stark (Paris), Max Turri (UC Santa Cruz),
Wouter Verkerke (UC Santa Barbara), Cecilia Voena (Rome)

June 21, 2001

Abstract

This analysis document is a user's guide to the RooFitTools package for performing unbinned maximum likelihood fits, and includes working examples and instructions on how to run them. RooFitTools is a library of C++ classes for creating compact and self-documenting macros to perform unbinned likelihood fits and display the results, without needing to compile and link any code. The package provides a flexible framework for building complex fit models and is straightforward to extend. This note describes several examples which are included in the package to help new users get started.

*primary contact

(New or recently modified sections are listed with underlined headings below.)

Contents

1 Introduction

This guide describes how to use the general framework for performing unbinned maximum likelihood fits that is implemented in the BaBar software package RooFitTools[1]. This document is intended to serve as a practical user's guide rather than a technical reference to the package's implementation. This guide does not assume expertise with ROOT or BaBar software.

RooFitTools is a library of C++ classes whose goal is to provide a reasonably natural vocabulary for efficiently specifying a unbinned maximum likelihood fit. To illustrate this goal, consider the following example of a binned χ^2 fit performed in PAW

```
PAW> h/file 1 data.hbook
PAW> hrin 10
PAW> h/fit 10 g+e
```

These instructions are compact and the notation `g+e` is a reasonably intuitive way to request a fit to the sum of a Gaussian and an exponential. One shortcoming of this notation, which the RooFitTools package tries to address, is that it is not self documenting: what data does histogram 10 contain? How are the fit parameters defined? etc. The main shortcoming, which is the real motivation for this package, is that PAW only provides convenient support for binned fits.

An analogous fit using an unbinned maximum likelihood method in RooFitTools looks like this:

```
RooRealVar // define the data variables and fit model parameters
  m("m","Reconstructed Mass",0.5,2.5,"GeV"),
  rmass("rmass","Resonance Mass",1.5,1.4,1.6, "GeV"),
  width("width","Resonance Width",0.15, 0.1, 0.2, "GeV"),
  bgshape("bgshape","Background Shape",-1.0,-2.0,0.0, "GeV"),
  frac("frac", "signal fraction",0.5, 0.0, 1.0);

// create the fit model components: Gaussian and exponential PDFs
RooGaussian signal("signal","Signal Distribution",m,rmass,width);
RooExponential bg("bg","Background Distribution",m,bgshape);

// combine them using addition (add a relative normalization parameter)
RooAddPdf model("model","Signal + Background",signal,bg,frac);

// read values of 'm' from a text file
RooDataSet *data= RooDataSet::read("mvalues.dat", m);
```

[1]The examples in this document were tested with V00-02-40 of the RooFitTools package.

```
// fit the data to the model with an unbinned maximum likelihood method
model.fitTo(data);
```

This is a working example which can either compiled into a standalone program, or else run interactively as an interpreted macro in ROOT[2] (since most of the computationally intensive code is already compiled into shared libraries, there is generally little performance benefit in compiling your macros.) The rest of this guide explains how to get started by running the examples in this guide and then describes each of the main sections in a typical fit macro which are used to:

- specify of the variables use to describe the data and the parameters used in the fit model,
- construct a probability model to fit to the data by combining basic building blocks, and
- run the fit and plot the results.

The final section covers techniques for using Monte Carlo samples to validate a fit and study statistical and systematic errors.

1.1 Building the Library

In order to use the RooFitTools package, you need a copy of the library containing its code. There are two versions of the library for each platform architecture: one (the "static" library) which you can link your compiled programs against, and one (the "shared" library) which you can load dynamically in an interactive ROOT session. The shared library, which is the most useful, is not normally built as part of a standard BaBar release so you will need to build your own copy.

Use the following steps to create your own BaBar test release in a directory `fitrel` and build the RooFitTools dynamic library (you can replace `newest` with your favorite recent release, use V00-02-40 for `<version>` or else a more recent tag announced in the "Interactive ROOT Help" hypernews forum)

```
newrel -t newest fitrel
cd fitrel
srtpath
addpkg RooFitTools <version>
addpkg workdir
gmake workdir.setup
gmake RooFitTools.rootlib ROPT=-Shared
```

[2] Experience with ROOT is not a prerequisite for using this package. Refer to `http://root.cern.ch` for an introduction to ROOT. You can post general ROOT questions and questions about RooFitTools to the BaBar "Interactive ROOT Help" hypernews forum.

Note that the `srtpath` will prompt you for two parameters: you can hit `RETURN` and use the defaults in both cases. This whole process should only take a few minutes, and the size of the resulting library should be a few Mb.

1.2 Running Interactively in ROOT

Once you have created the RooFitTools shared library in your test release, you can start an interactive ROOT session from your release's `workdir` subdirectory using the command `bbrroot`. You should see something like this (and a colorful opening graphic which you can suppress using `bbrroot -l`)

```
% cd workdir
% bbrroot
...
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
...using style 'plain'
...found RELEASE directory
...found PARENT directory
...running in release 8.6.5a-physics-1
root [0]
```

Your input will now be parsed by ROOT's C++ interpreter, which accepts standard C++ expressions as well as some cryptic built-in commands that begin with a period: the ones to remember are `.x` (to execute a macro – see below) and `.q` to quit ROOT. Note that the output lines above beginning with '`...`' are generated by a startup file `RooLogon.C` in your `workdir` directory (equivalent to PAW's `pawlogon.kumac`) which you are free to customize.

Once you have a ROOT prompt, load the RooFitTools dynamic library with the commands[3]

```
gSystem->Load("libProof.so");
gSystem->Load("libTreePlayer.so");
loadSrtLib("RooFitTools");
```

To do this automatically when you start ROOT, add the command to your `RooLogin.C` file (see the comments at the end of the file). You are now ready to test and run macros using the classes in the RooFitTools package. A ROOT macro is a sequence of C++ statements enclosed in a subroutine (the subroutine's return type is optional)

[3]The first two of these commands are usually only necessary on Linux platforms (due to a linking problem in the core ROOT libraries), but will not do any harm on other platforms.

```
hello() {
  cout << "hello world" << endl;
}
```

To run this macro in ROOT, save it to a file called `hello.cc` in your `workdir` directory and execute it using

```
.x hello.cc
```

The `.x` command reads a file and executes a subroutine with the same name. To run any of the examples in this guide, simply enclose them in a subroutine and save it to a file of the same name. You can use the `RELEASE` and `PARENT` links in your `workdir` directory to access macros in your test release (`RELEASE`) or your base release (`PARENT`). To run the example in Section 1, copy the data file `mvalues.dat` from `RooFitTools/examples/` into your `workdir/` directory and run the macro in ROOT (after loading the shared library) using

```
.x RELEASE/RooFitTools/examples/intro.cc
```

When reading a macro, remember that all class names in the RooFitTools package begin with 'Roo' and most ROOT built-in class names begin with the letter 'T'.

Comments in ROOT macros follow C++ conventions. Semicolons at the end of each statement are optional at the command line, but required in a macro. ROOT has some useful shell-like features which make interactive work easier. Use the up/down arrow keys to access your recent commands. Use the tab key to complete the command you have started, or get a list of possible completions. Tab completion is a convenient way to find out what methods an object supports and what arguments they take.

If you have general questions about ROOT or specific questions about this package, please post them to the BaBar "Interactive ROOT Help" hypernews forum.

2 Defining Variables

The first section of a fit macro is where you define your data's dependent variables and the parameters of your likelihood model. RooFitTools does not make a distinction between dependent variables and model parameters: they are all `RooRealVar` objects. Variables have a value, minimum and maximum limits, an error, a name, a description, and (optionally) units. Variables can also be fixed, which means that they are treated as constants in a fit. Section 2.1, below, describes the general methods available for working with variables. There is a special type of discrete variable, called a "category" variable, which is described in Section 2.2.

2.1 General-Purpose Variables

You can choose from three forms of the constructor for a variable

```
RooRealVar
  eta("eta","Mistag Rate",0.15,0.0,1.0), // specify value and range
  m("m","Invariant Mass",0.5,2.5,"GeV"), // specify range, no value
  tauB("tauB","B Lifetime",1.63,"ps");   // specify value only: fixed
```

The form with a value specified is convenient for fit parameters: the value will be used as the starting point in the fit. The form without a value specified is convenient for dependent variables whose value will be taken from data.

All forms of the constructor begin with a name and description, and end with an optional string describing their units. The name is used in messages and as a key for storing variable objects to a file, and by convention should be set equal to the object's name. The description and units are used to create histogram labels. Choosing a useful description makes it easier for others reading your macro to understand how you are performing a fit. A variable's range is used differently for dependent variables and fit parameters: the range of a dependent variable specifies an effective cut on the data and so specifies the region for normalizing a likelihood probability density function. The range of a variable used as a fit parameter is not normally limited in the fit, unless you request this with

```
eta.setLimits();
```

In either case, the range is used to construct histograms for a variable. Variable errors are set from a fit and not by hand. Once a variable is constructed, you can change its value

```
eta= 0.2;
Eoverp= (Double_t)E/(Double_t)p;
```

Note that an explicit cast is required when using a variable on the right-hand side of an assignment. To fix a variable so that it will not be floating in a fit use (`kFALSE` is a ROOT built-in boolean constant)

```
eta.setConstant();       // fixed
eta.setConstant(kFALSE); // floating
```

To get information about a variable you can print its value, range, description, and units using

```
eta.Print();
```

To access a parameter's fitted error, use the `GetError()` method, or to print its value and error use

```
eta.Print("E");
```

2.2 Category Variables

RooFitTools supports a specialized type of variable which is useful for identifying different types of events being used in a fit (e.g., from a signal or background control sample). These are called "category" variables, and have their own construction method in which you only specify a name and title (which have the same meaning as for other variables)

```
RooCategory eventCat("eventCat","Type of Event (Signal,BG1,BG2)");
```

A newly created category variable does not define any cateogries. Use the `defineType()` method to declare the types of categories you are interested in

```
eventCat.defineType("B0 -> D* l nu signal candidate");
eventCat.defineType("Fake lepton control sample candidate");
eventCat.defineType("Uncorrelated control sample candidate");
```

Use the general-purpose ROOT `Print()` method with the `"T"` option to list the types defined for a category variable

```
root [1] eventCat.Print("T")
  [0] "B0 -> D* l nu signal candidate" (code = 1)
  [1] "Fake lepton control sample candidate" (code = 2)
  [2] "Uncorrelated control sample candidate" (code = 3)
```

Note that each type of category is associated with an integer code: by default these are assigned automatically in ascending order starting from one. These codes are used to assign the value of a category variable and for reading and writing data sets to ASCII files (as described in Section 3). You can override the default code assignments for any category by providing an extra parameter to the `defineType` method

```
mixingCat.defineType("Mixed Events",-1);
mixingCat.defineType("Unmixed Events",+1);
```

Category variables are also RooRealVariables and so inherit all of the behavior described above (although in some cases this does not make much sense). For example, you can assign a value like this

```
root [1] mixingCat = -1;
root [2] mixingCat.Print()
RooCategory: mixingCat = "Mixed Event" (code=-1) : Mixing Event Category
```

If the right-hand side of an assignment (after rounding to the nearest integer) does not match a defined type, then a default value will be used and you will get the following warning message

```
root [1] mixingCat = 0;
mixingCat: code 0 is not a valid category: forcing to -1
```

2.3 Blind Analysis Variables

RooFitTools supports the use of blind model parameters. A blind parameter is implemented as a `RooBlindVar` object which is derived from RooRealVar and can as such be used in any place where a `RooRealVar` is expected.

Before creating a blind variable the type of blinding needs to be defined as a *blinder* of type `RooBlindBase`.

```
// Define blinding string and magnitude.
Double_t mag=0.00003;
RooBlindGaussian blinderMass("blinderMass", "This is a blinding string",mag);
```

The blinding string is used to compute a seed for a platform independent random number generator. For objects of type `RooBlindGaussian` the blind offset will be a random number from a Gaussian distribution with 0 mean and σ = mag. It is also possible to get a blind offset taken from a uniform distribution in the interval $[-\text{mag}; \text{mag}]$ by declaring a blinder of type `RooBlindUniform`. A blind variable is then declared by giving the blinder in the constructor as

```
RooBlindVar rDeltaM("rdeltaM","Mass Difference",&blinderMass,
                    0.1454,0.13957,0.15457, "GeV");
```

Each blind variable should use its own blinder unless there is a specific requirement for two blind variables to have the same blind offset. A specific example of using blind variables can be found in section A.5.

Care should be taken not to show yourself the blind offset used. This means you should never use blinding on MC data where you presumably know the true answer. The way to solve this is to turn off the blinding before the actual fit

```
rDeltaM.setBlind(kFALSE);
modelDeltaM.fitTo(data,"M");
```

Also do not turn on/off the blinding using `setBlind(kFALSE)` after the fit has been performed as this might reveal the offset.

If a variable only has a physical meaning within a given interval this interval will be changed by the blinding method. An example could be the relative fraction of signal and background events which only is physical on the interval $[0; 1]$ and most likely give negative values of the PDF outside the limits. Two problems arise from this:

- The start value might be outside the physical region causing the fit to fail straight away. This can be solved by using the RooBlindUniform blinder and have the maximum blind offset and start value defined such that the hidden start value is guaranteed to be within the interval.
- If the variable during the fit either gets "stuck" at the edge of the interval or runs outside the physical region causing the fit to fail, you might by accident be able to infer the blind offset.

3 Reading Data

After defining your variables, the next section of a fit macro specifies the data you want to fit to. The data used in a fit consists of a set of values for a vector of dependent variables, $\{\vec{x}_k\}_{k=1}^n$. Data sets in RooFitTools are represented by `RooDataSet` objects. There are two ways to fill a dataset with values: from a text file, or from a ROOT tree (the equivalent of a PAW ntuple). If instead you want to fit data contained in a PAW ntuple, you can either create an intermediate text file (using the PAW `ntuple/dump` command) or else convert the PAW ntuple into a ROOT tree (using the `h2root` command provided with ROOT).

The easiest way to create a `RooDataSet` object using data from an external source is to read values from a text file

```
RooDataSet *data= RooDataSet::read("JPsiKsMC.dat", dz, sigma, tag);
```

In this example, the variables `dz`, `sigma`, and `tag` must already be defined (see Section 2). The ranges you specify for these variables will be interpreted as cuts to apply on the data being read: any event with at least one dependent variable out of range will be excluded from the data set. By default, the `read()` method will print a warning for each excluded entry. To supress these warnings, add `"Q"` (for Quiet) at the end of the `read()` arguments:

```
RooDataSet *data= RooDataSet::read("JPsiKsMC.dat", dz, sigma, tag, "Q");
```

After reading the file, the `read()` method prints a summary of the number of events which were read and which passed these cuts. A dataset may contain variables which are not used in a fit.

The text file containing the data may include blank lines or comment lines beginning with a hash (`#`) symbol. Any other lines are expected to contain a list of values for each dependent variable, in the order in which they are declared in the `read()` method. There are no special spacing or formatting requirements.

An alternative way of initializing a dataset is from a TTree (the ROOT equivalent of a PAW ntuple). If the TTree is already in memory, then use:

```
RooRealVar m("xmjpsi","Muon Pair Invariant Mass",2.5,3.4,"GeV");
...
RooDataSet *data= new RooDataSet("dataset",tree,m,...);
```

where `tree` is a pointer to a TTree object and you can list up to six variables. The names of the variables (more precisely, the first argument of the variable's constructor) determines the value that will be used to fill that variable from each entry in the tree. In the simplest case, the name exactly matches a variable in the tree. For example, to convert the following tree:

```
root [1] dataTree->Print()
******************************************************************************
*Tree    :dataTree  : data for fitting                                       *
*Entries :     4302 : Total  Size =    188126 bytes  File  Size =      97373 *
*        :          : Tree compression factor =   2.54                       *
******************************************************************************
*Branch  :dt        : dt/F                                                   *
*Entries :     4302 : Total  Size =     21365 bytes  File Size  =      14377 *
*Baskets :        1 : Basket Size =     32000 bytes  Compression=   1.49     *
*............................................................................*
*Branch  :dtTrue    : dtTrue/F                                               *
*Entries :     4302 : Total  Size =     21365 bytes  File Size  =      14377 *
*Baskets :        1 : Basket Size =     32000 bytes  Compression=   1.49     *
*............................................................................*
*Branch  :error     : error/F                                                *
*Entries :     4302 : Total  Size =     21371 bytes  File Size  =      13994 *
*Baskets :        1 : Basket Size =     32000 bytes  Compression=   1.53     *
*............................................................................*
```

to a dataset use:

```
RooRealVar dt("dt","Measured delta(t)",-15,+15,"ps");
RooRealVar error("error","Per-Event Error",0.1,2.4,"ps");
RooDataSet dataset("dataset","Resolution Data",dt,error);
```

Note that not all variables need to be transferred into the dataset. More generally, the name can be any valid expression involving the names of tree variables and built-in functions. For example:

```
RooRealVar pull("(dt-dtTrue)/error","Delta(t) Pull",-150,+150);
RooDataSet dataset("dataset","Resolution Pulls",pull);
```

If the TTree you would like to use is not in memory, but is stored in a ROOT file (which may or may not have already been opened) then use instead:

```
RooDataSet dataset("dataset","file.root","h1",dt,error);
```

where `file.root` is the name of the ROOT file to use (which will be opened if necessary) and `h1` is the name of the TTree object in the file. As with the `read()` methods described above, the ranges specified for variables are used to select events.

You can apply standard TTree cuts on any of the TTree variables (including ones not transferred to the dataset) to select which entries are used in the dataset using an optional extra parameter with the constructors described above, for example:

```
RooDataSet dataset("dataset","file.root","h1",pull,"abs(dtTrue)<10");
```

If you encounter problems with undefined symbols when you try this example, make sure that you have loaded the Proof and TreePlayer shared libraries, as described in Section 1.2.

Once you have created a data set, you can plot the distribution of any of its variables using

```
data->Plot(dz)->Draw();     // 1-d histogram
data->Plot(dz,tag)->Draw(); // 2-d histogram
```

The return value of the `Plot()` method is actually a pointer to a ROOT histogram object: see Section 4.8.2 for more information on using histograms. You can also use a data set just like a TTree (the ROOT equivalent of a PAW ntuple) and make plots of functions of a variable with optional cuts

```
data->Draw("dz*tag","abs(sigma)<0.2");
```

In this case, the histogram range will be calculated automatically and you will need to add labels yourself.

4 Defining a Fit Model

A fit model consists of a normalized probability density function (PDF), $F(\vec{x}\,;\vec{p})$, of the dependent variables used to describe the data being fit, $\vec{x}$, which is parameterized by variables $\vec{p}$. The corresponding likelihood to be maximized (with respect to the parameters $\vec{p}$) for a sample of events, $\{\vec{x}_k\}_{k=1}^{n}$, is

$$\mathcal{L}(\vec{p}) = \prod_{k=1}^{n} F(\vec{x}_k\,;\vec{p}) \; .$$

A PDF must be positive over the range of $\vec{x}$ values covered by the data in order to avoid $\mathcal{L} \leq 0$, and must be normalized so that

$$\frac{\partial}{\partial p_k} \int dx_1 \ldots \int dx_n\, F(\vec{x}\,;\vec{p}) \;\cdot\; \theta(\vec{x}) = 0 \; ,$$

where the term $\theta(\vec{x})$ represents any cuts applied to the dependent variables before making the fit (i.e., the variable limits), and so generally introduces normalization factors which depend on cut values. Figure 1 shows the importance of including cuts in the normalization: the two curves compare exponential fits to a dataset with cuts sampled from an exponential distribution, using PDFs normalized with and without cuts taken into account. Cuts are also important when using PDFs like a polynomial which are not necessarily well behaved beyond the region covered by the data.

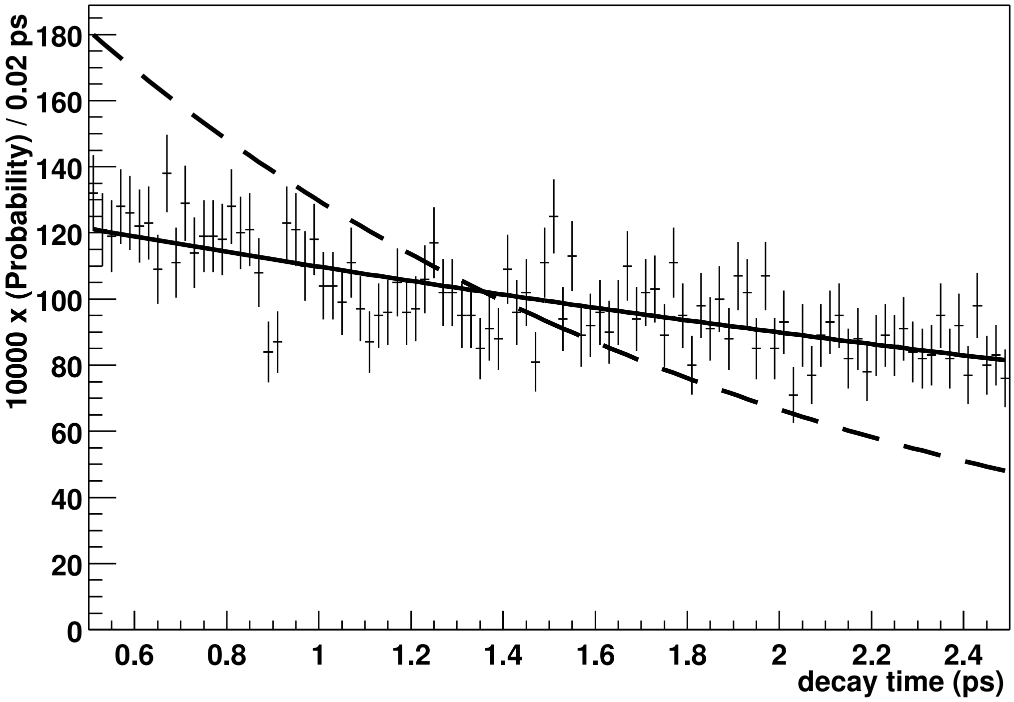


Figure 1: A comparison of fits to data generated from an exponential distribution with $\tau = 5$ and cut at $0.5 \leq t \leq 2.5$. The solid curve is the result of fitting with the cut taken into account in the normalization, which gives the expected result. The dashed curve is the result of fitting without taking the cut into account, which gives a value of about $< t > \simeq 1.5$.

The general approach to building a PDF in RooFitTools is to combine a set of predefined PDFs using addition, multiplication, and convolution. The software takes care of normalizing the model you build, and includes checks to avoid parameters which can give probabilities ≤ 0. It is straightforward to extend the package with new predefined PDFs or composition methods (see Section B).

This section describes how to use predefined PDFs and how to combine them. It also covers some general tools for displaying information about any PDF.

4.1 Model Building Blocks

This section reviews the predefined building-block PDFs available in RooFitTools. The names of the PDF objects and variables constructed in the examples in this section are arbitrary: choosing suitable names will make your fit macro easier for others to understand. The symbol N represents a normalization factor in the equations below, which generally depends on the parameter values and the limits on the dependent variables (and which the package calculates for you.)

The first two arguments used to construct any predefined PDF are a name and a title. The name is used in printed messages and, by convention, should be set equal to the object's name. The title should be descriptive and will be used in plots (the titles in this guide do not set a good example since they are generally kept short to fit within one line.)

4.1.1 Exponential Distribution

This section describes two general-purpose parameterizations of an exponential distribution which are intended as building blocks for more complex shapes. For lifetime fits to a decay time distribution, consider using the more powerful PDFs described in Section 4.2 instead.

Create the exponential distribution

$$E_1(t\,;\tau) = \frac{1}{N}\,\exp(-t/\tau)\;\cdot\;\theta\,(t \geq 0)$$

with the constructor

```
RooLifetime E1("E1","A Lifetime Distribution",t,tau);
```

(The symbol names `E`, `t`, and `tau` are arbitrary.) Note that this function is defined to be zero for negative values of t. The minimum value of τ must be positive and the minimum value of t must be ≥ 0. An alternate constructor

```
RooExponential E2("E2","An Exponential Distribution",x,c);
```

creates the double-sided exponential distribution

$$E_2(x\,;c) = \frac{1}{N}\,\exp(c\,x)$$

with no restrictions on the ranges of x or c.

4.1.2 Quadratic Exponential Distribution

The function

$$E(x\,;a,b,c) = \frac{1}{N}\cdot\exp\left(a(x+b)^2+c\right)$$

can be constructed using

```
RooQuadExponential E("E", "Quadratic Exponential", x, a, b, c);
```

This function has been found to be useful to parameterise PDFs of the cosine of the angle of B candidates w.r.t. the thrust axis.

4.1.3 Gaussian Distribution

The Gaussian distribution

$$G_1(x\,;\mu,\sigma) = \frac{1}{N} \cdot \exp\left(-(x-\mu)^2/(2\sigma^2)\right)$$

can be constructed using

```
RooGaussian G1("G1","A Gaussian Distribution",x,mu,sigma);
```

This form of the Gaussian is appropriate when you are fitting to a global resolution parameter, σ. If you have measured a per-event error then use the alternate form

$$G_2(x,\sigma\,;\mu,s) = \frac{1}{N} \cdot \exp\left(-(x-\mu)^2/(2s^2\sigma^2)\right) \cdot g(s\sigma, x_{min}, x_{max}) \cdot \rho(\sigma)$$

which can be constructed using

```
RooGaussEvt G2("G2","Per-Event Weighted Gaussian",x,sigma,mu,s,rho);
```

where `rho` is a one-dimensional PDF describing the distribution of σ. An alternate form of the constructor does not use the `rho` parameter and corresponds to a flat distribution in σ: this model is generally not realistic, but will not bias fitted parameters (which don't model the σ distribution) as long as the true distributions of x and σ are uncorrelated (see Section A.3 for a demonstration of this.) The per-event error PDF replaces the global resolution parameter, σ, by a global scale factor parameter, s, which multiplies the per-event error estimate. The function, $g(s\sigma, x_{min}, x_{max})$, is chosen so that the projected distribution of σ reproduces $\rho(\sigma)$

$$\int_{x_{min}}^{x_{max}} G_2(x,\sigma\,;\mu,s)\,dx = \frac{1}{N} = \frac{1}{\sigma_{max} - \sigma_{min}}\,.$$

This is generally not true of the data, but using this model will not bias the fitted parameters (which don't model the σ distribution) as long as the distributions of x and σ are uncorrelated.

Refer to Section 4.4 for an alternate method of implementing PDFs which depend on a per-event error, without using the RooGaussEvt building block.

4.1.4 Bifurcated Gaussian Distribution

The bifurcated Gaussian distribution is a Gaussian with a different value of σ on either side of the mean:

$$G_b(x\,;\mu,\sigma) = \frac{1}{N} \cdot \exp\left(-(x-\mu)^2/(2\sigma(x-\mu)^2)\right),$$

where $\sigma(x-\mu) = \sigma_L$ for $x < \mu$ and σ_R for $x \geq \mu$. This PDF is constructed using

```
RooBifurGauss Gb("Gb","A Bifurcated Gaussian Distribution",x,mu,sigL,sigR);
```

Note that when the validity range of x contains only values larger (smaller) than μ, then σ_L (σ_R) must be fixed by a call to setConstant() in order for the fit to work properly.

4.1.5 Polynomial Distribution

The polynomial distribution

$$P(x\,; c_0, \dots, c_n) = \frac{1}{N} \cdot \left(1 + \sum_{k=1}^{n} c_k x^k\right)$$

can be constructed using

```
RooPolynomial P("P","A Polynomial Distribution",x,c1,c2);
```

where the number of parameters used in the constructor (up to a maximum of 5) determines the order of the polynomial: using two parameters gives a second order, or quadratic, distribution.

4.1.6 Chebychev Polynomial Distribution

The polynomial distribution

$$T(x\,; c_0, \dots, c_n) = \frac{1}{N} \cdot \left(T^0(x) + \sum_{k=1}^{n} c_k T^k(x)\right)$$

where $T^k(x)$ is a k^{th} order Chebychev polynomial can be constructed using

```
RooChebychev T("T","A Chebychev Distribution",x,c1,c2);
```

where the number of parameters used in the constructor (up to a maximum of 5) determines the order: using two parameters gives a second order, or quadratic, distribution.

The 'natural' domain of Chebychev polynomials is the range $[-1, 1]$, so the range specified by the (instance of the) RooRealVar class specified in the RooChebychev constructor is used to map the variable onto this range.

Chebychev polynomials have properties which make them usefull for fitting: they are orthogonal in the interval $[-1, 1]$ over a weight $(1 - x^2)^{-1/2}$, and at all the maxima (minima) $T^k(x_{max}) = 1$ $(T^k(x_{min}) = -1)$.

4.1.7 Argus Background Distribution

The ARGUS distribution[1] for the background shape in a beam-constrained mass plot

$$A(m\,; m_0, c) = \frac{1}{N} \cdot m\,\sqrt{1 - (m/m_0)^2} \cdot \exp(c\,(1 - (m/m_0)^2)) \cdot \theta(m < m_0)$$

is constructed using

```
RooArgusBG A("A","ARGUS Background Shape",m,mmax);
```

Note that m_0 represents the kinematic upper limit for the constrained mass and is usually held fixed at half of the center of mass energy (nominally 5.29 GeV for $e^+e^- \to \Upsilon(4S)$) in a fit. The minimum value of m_0 must be $\geq$ the maximum value of m. Using $c = 0$ gives the the expected distribution of beam-constrained mass values for background which is uniformly distributed in phase space[1]. Figure 6 shows an example of a model which uses this PDF.

4.1.8 Breit wigner Distribution

The Breit Wigner distribution takes the form [2]

$$B(x\,; m, \Gamma) = \frac{1}{N} \cdot \frac{1}{(x-m)^2 + (\Gamma/2)^2}$$

$1/N$ is a normalization factor that depends on the range of x, and the parameters m and Γ are the central mass and width of the resonance, respectively. Create the PDF with

```
RooBreitWigner bw("bw", "A Breit Wigner Distribution", x, m, Gamma);
```

4.1.9 Crystal Ball Lineshape Distribution

The Crystal Ball lineshape distribution[3]

$$C(m\,; m_0, \sigma, \alpha, n) = \frac{1}{N} \cdot \begin{cases} \exp\left(-(m-m_0)^2/(2\sigma^2)\right) & , \; m > m_0 - \alpha\sigma \\ \dfrac{(n/\alpha)^n \, \exp(-\alpha^2/2)}{((m_0-m)/\sigma + n/\alpha - \alpha)^n} & , \; m \leq m_0 - \alpha\sigma \end{cases}$$

is useful for fitting a radiative tail, e.g., $m_{e^+e^-}$ in $J/\psi \to e^+e^-(\gamma)$ decays. It consists of a Gaussian signal peak matched to a power law tail. Figure 7 shows an example of a model using this distribution. Construct this PDF using

```
RooCBShape C("C","Crystal Ball Lineshape",m,m0,sigma,cut,power);
```

Note that the parameter n is not necessarily integer, and is usually held fixed in a fit: lower values generate a longer tail. The parameter α determines the crossover point from the Gaussian distribution to the power law tail distribution, in units of the peak width, σ. Typical values for $|\alpha|$ are 0.6–1.1. With $\alpha > 0$ the tail is below the peak, and with a negative value the tail is above the peak.

4.1.10 Threshold Background Distribution

The background distribution for a mass (or mass difference) spectrum near a lower threshold is well described by a PDF of the form

$$D(m\,; m_0, c, a) = (1 + m - m_0)^a \; [1 - \exp\left(-(m-m_0)/c\right)] \cdot \theta(m > m_0)$$

which can be constructed with either of these forms

```
RooDstarBG D("D","Mass Diff Background Shape",deltam,deltammax,c);
RooDstarBG D("D","Mass Diff Background Shape",deltam,deltammax,c,a);
```

Note that the a parameter is optional and defaults to the constant zero if it is not specified in the constructor. The choice of name for this class reflects the fact that this PDF is often used to model the background contributing to a $\mathrm{D}^* - \mathrm{D}^0$ mass difference distribution. The parameter m_0 is the kinematic threshold to use, and should be equal to the $\pi^\pm$ mass for a $\mathrm{D}^* - \mathrm{D}^0$ mass difference fit. The minimum value of m must always be set larger than m_0 (to avoid regions with exactly zero probability). Figure 10 shows an example of a model which uses this threshold background distribution.

4.1.11 Johnson SU Distribution

The Johnson SU distribution [4]

$$J(x;\xi,\lambda,\gamma,\delta) = \frac{\delta}{\lambda\sqrt{2\pi}\sqrt{1+\left(\frac{x-\xi}{\lambda}\right)^2}} \exp\left\{-\frac{1}{2}\left[\gamma + \delta\,\mathrm{Argsh}\left(\frac{\mathrm{x}-\xi}{\lambda}\right)\right]^2\right\}$$

can be constructed using

```
RooJohnsonSU J("J","Johnson SU distribution",x,xi,lambda,gamma,delta);
```

This distribution can be used to fit experimental data that are 'almost' gaussian, except for a Landau-like tail on one side. An example is shown in figure 2. The parameters δ and γ determine the shape of the distribution: for $\delta > 0$, $J(x;\xi,\lambda,\gamma,\delta)$ is positive and normalized to one, and the sign of γ determines if the tail is located at low x ($\gamma > 0$) or at high x ($\gamma < 0$). Higher values of $|\delta|$ and $|\gamma|$ result in a more symmetrical distribution and a sharper peak. The parameters ξ and λ control the position and width of the distribution, respectively.

4.1.12 Non-Parametric Distributions

A non-parametric PDF describes a distribution empirically, without referring to any model of the expected shape. A histogram is an example of a non-parametric description of a dataset, but is not directly suitable for use in unbinned fits where we require a continuous distribution. One solution is to smooth a histogram and use the resulting sequence of splines as a continuous description of the underlying distribution. A more powerful technique is advocated in Reference [5], and leads to the "KEYS" (Kernel Estimating Your Shapes") method which has been developed by Kyle Kranmer in ALEPH, and used by the LEP Higgs working group.

The basic idea in the KEYS approach is to model the distribution of a sample of events as the sum of individual equal-area Gaussian kernels, G, for each event

$$K(x) = \sum_{k=1}^{n} \frac{1}{nh_k} G\left(\frac{x - x_k}{h_k}\right) \ ,$$

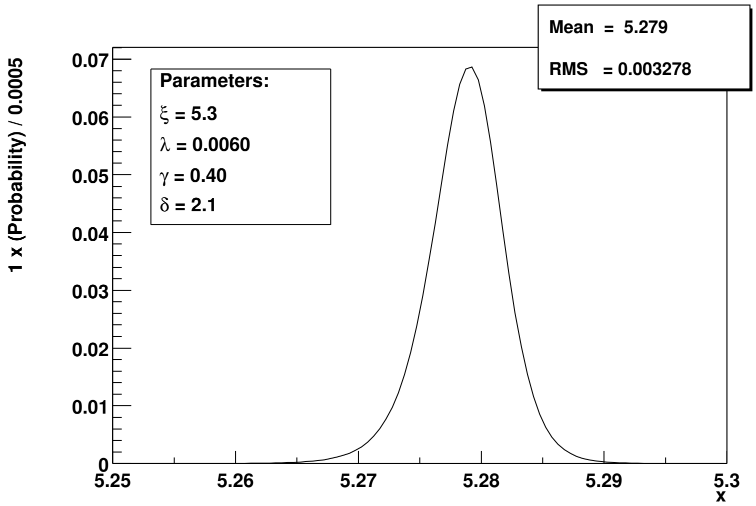


Figure 2: The Johnson SU distribution for a particular choice of parameter values.

where the extent of each event's contribution, h_k, is calculated adaptively as a function of the local density of events.

Construct a non-parametric KEYS PDF using

```
RooKeysPdf K("K","Monte-Carlo Background Shape",mass,mcData);
```

to describe the distribution of the dependent variable `mass` in the dataset `mcData`. To load a new dataset `mcData1` of the same dependent variable `mass`

```
K->LoadDataSet(mcData1);
```

For distributions which have substantial density near the range boundaries, the default parameterization can have difficulties. If kernals are thought of as tiny gaussians at each event value, one can imagine that probability is "leaking" out the edges for points near the edge. To solve this problem, a final, optional parameter can be specified. It takes values `RooKeysPdf::NoMirror` (default), `RooKeysPdf::MirrorLeft`, `RooKeysPdf::MirrorRight`, or `RooKeysPdf::MirrorBoth`.

```
RooKeysPdf K("K","Monte-Carlo Background Shape",mass,mcData,RooKeysPdf::MirrorBoth);
```

This causes the distribution to be mirrored on neither, the left, the right, or both boundaries when summing the kernals. One should choose to mirror a distribution *at least* on the boundaries where significant density occurs. An event's contribution in the kernal is still adaptively calculated using the original distribution only. Events in the mirrored distributions have identical contributions as their counterparts in the original distibution. This type of mirroring does not increase the computation time significantly, so it is recommended to use this option to obtain the best distributions.

A non-parametric PDF is useful for including distributions in a fit that have a complex shape which cannot be easily modelled. Examples would be a background distribution obtained from a Monte Carlo simulation, or the distribution of a neural-network output used as a background discriminating variable. Refer to Section A.6 for an example of using a non-parametric PDF.

4.1.13 DIRC Cerenkov angle PDFs

The `RooDircPdf` model allows the calculation of PDFs of the DIRC Cerenkov angle, θ_C. The PDFs are represented as double gaussian functions, as outlined in [11]. A primary (core) gaussian is used to describe the distribution of the difference between the observed and expected Cerenkov angles for the given hypothesis (pion or kaon), while a satellite (tail) gaussian represents the contamination (mis-identification) from tracks of the other hypothesis. The mean and sigma values of the core and tail parts, as well as the relative normalisation between them, are parameterised in terms of the track momentum (at the DIRC), p, and/or the cosine of the angle of the track, $\cos\theta$. The total PDF, for kaons, can be written as:

$$P_K(\theta_C; p, \cos\theta) = \frac{R_K}{\sigma_{\text{core},K}\sqrt{2\pi}N_{\text{core}}} \exp\left[-\frac{\left(\theta_C - \left(\theta_K^{\text{off}} + \theta_K^{\text{ex}}\right)\right)^2}{2\sigma^2_{\text{core},K}}\right] + \frac{1 - R_K}{\sigma_{\text{tail},K}\sqrt{2\pi}N_{\text{tail}}} \exp\left[-\frac{\left(\theta_C - \left(\theta_K^{\text{off}} + \theta_\pi^{\text{ex}}\right)\right)^2}{2\sigma^2_{\text{tail},K}}\right]$$

Here, the mean and sigma values of the core gaussian are $\theta_K^{\text{off}} + \theta_K^{\text{ex}}$ and $\sigma_{\text{core},K}$, repectively. The variable θ_K^{off} is the offset (for the kaon hypothesis), while θ_K^{ex} is the expected Cerenkov angle for the kaon hypothesis

$$\theta_K^{\text{ex}} = \cos^{-1}\left(1/n\beta_K\right)$$

The offset and sigma values are parameterised in terms of functions of p and/or $\cos\theta$:

$$\theta_K^{\text{off}} = f(\cos\theta, p)$$

$$\sigma_{\text{core},K} = g(\cos\theta, p)$$

The tail gaussian has the mean and sigma values $\theta_K^{\text{off}} + \theta_\pi^{\text{ex}}$ and $\sigma_{\text{tail},K}$, repectively, where the variable θ_π^{ex} is just the expected Cerenkov angle for the track under the alternative (pion) hypothesis. The sigma value for the tail is also parameterised in terms of p and/or $\cos\theta$:

$$\sigma_{\mathrm{tail},K} = h(\cos\theta, p)$$

The relative normalisation, R_K, is equal to the ratio of the area of the core gaussian to that of the total distribution (core plus tail), and can also be parameterised in terms of p and $\cos\theta$. The values N_{core} and N_{tail} are just the extra normalisation factors to ensure that, depending on the range of Cerenkov angles used, the total PDF is normalised to unity. The PDF for pions follows the same form as above, with appropriate substitutions between K and π in the above variables. The constructor for the class is

```
  RooDircPdf dirc("dirc", "Dirc Model", drcMtm, thetaC, trkTheta,
                  refraction, mass, otherMass, coreMeanFun, coreSigmaFun,
                  tailMeanFun, tailSigmaFun, relNormFun, milliRadians);
```

Here, `drcMtm` is the momentum of the track at the DIRC, `thetaC` is the Cerenkov angle, while `trkTheta` is the polar angle of the track (w.r.t the z axis). Also, we have the constants `refraction`, `mass` and `otherMass`, which are the index of refraction, the mass for the given hypothesis (pion or kaon) and the mass for the other hypothesis (kaon or pion), respectively. All of these are represented by `RooRealVar` objects.

The parameterisations of the mean and sigma values of the core and tail gaussians are given by the ROOT `TF1` objects `coreMeanFun`, `coreSigmaFun`, `tailMeanFun` and `tailSigmaFun`. The parameterisation of the relative normalisation is given by the `TF1` object `realNormFun`. All of these functions use both $\cos\theta$ and p as input variables, in that order. Note that if any of the these functions are just constants, then they can be represented by functions like

```
TF1 constant("constant", "pol0");
constant.SetParameter(0, <value>);
```

Also, if any of the above `TF1` functions only depend on one of the variables (p or $\cos\theta$), then the other variable must be set to zero when reading in the data. Finally, there is the option to use either radians or milliradians in the calculation (depending on what the parameterisations use). The `milliRadians` boolean needs to be set to true (or `kTRUE`) if milliradians are used (default is false). An example using the `RooDircPdf` class can be found in `RooFitMacros/BBDecays/Charmless/TestD0piFit.cc` in the package `RooFitMacros V00-00-18`.

4.2 Decay-Time Distributions

In addition to the basic building blocks described in the previous section, RooFitTools provides some more complex models for the efficient fitting of a class of decay-time distributions which are often encountered in particle physics. In order to introduce the formalism used in this package, we first define a "true" physics decay-time distribution (i.e., one that is not biased or smeared) as

$$f_0(t\,;\tau,\vec{p}) = \frac{1}{2\tau}\exp(-|t|/\tau)\, g(t\,;\vec{p})\,\theta(t)\ ,$$

where $g(t\,;\vec{p})$ is an arbitrary function (with parameters $\vec{p}$) which modulates a lifetime distribution characterized by the parameter τ. The function $\theta(t)$ takes the values zero and one, and restricts the range of t over which f_0 takes non-zero values: for example, a fit in which the decay time is measured relative to a primary decay vertex would be restricted to $t \geq 0$ (in the absence of smearing or bias), while a fit to the relative decay time between two particles would cover the full range of t.

Next, we introduce possible biases and the effects of resolution smearing using

$$f(t\,;\tau,\vec{p},\vec{q}) = \int_{-\infty}^{\infty} ds\, f_0(s\,;\tau,\vec{p})\, h(t-s\,;\vec{q})\ ,$$

where $h(t{-}s\,;\vec{q})$ models the probability density (in terms of the parameters $\vec{q}$) of measuring a value t corresponding to a true value s. Finally, we define normalized PDFs for three different possible ranges of the true value ("double-sided", "single-sided", and "flipped") as

$$f_{\rm double} \equiv \frac{f_+ + f_-}{N_+ + N_-} \quad , \quad f_{\rm single} \equiv \frac{f_+}{N_+} \quad , \quad f_{\rm flip} \equiv \frac{f_-}{N_-}\ ,$$

where $N_\pm$ are normalization factors, and f_+, f_-, are restricted to $s \geq 0$ and $s < 0$, respectively.

We can write the $f_\pm$ functions as integrals over positive (unsmeared) times, s, as

$$f_\pm(t\,;\tau,\vec{p},\vec{q}) = \int_0^{\infty} ds\, \frac{1}{2\tau}\exp(-s/\tau)\ [g_+(s\,;\vec{p}) \pm g_-(s\,;\vec{p})]\ h(t \mp s\,;\vec{q})\ ,$$

where

$$g_\pm(s\,;\vec{p}) = \frac{g(s\,;\vec{p}) \pm g(-s\,;\vec{p})}{2}$$

are the symmetric $(+)$ and antisymmetric $(-)$ projections of the modulation function. In order to normalize these decay-time distributions, we use the integrals

$$F_\pm(x\,;\tau,\vec{p},\vec{q}) = \int_0^{\infty} ds\, \frac{1}{2\tau}\exp(-s/\tau)\ [g_+(s\,;\vec{p}) \pm g_-(s\,;\vec{p})]\, H_\pm(s,x\,;\vec{q})$$

where

$$H_\pm(s,x\,;\vec{q}) = \int_{x_0}^{x} dt\ h(t \mp s\,;\vec{q})$$

is a definite integral of the resolution function relative to a conveniently chosen fixed point x_0. Using these integrals, we can write

$$N_\pm = F_\pm(t_2\,;\tau,\vec{p},\vec{q}) - F_\pm(t_1\,;\tau,\vec{p},\vec{q})\ ,$$

where we normalize on the interval $t_1 \leq t \leq t_2$.

The general approach used in RooFitTools is to factorize a decay-time distribution into separate models for the underlying physics and the resolution smearing and bias. For example, to apply a double-Gaussian resolution model to a physics model for unmixed $B^0\overline{B^0}$ events, you would use

```
RooGaussModel resolution("resolution","Resolution Model",
                         resid,sig0,nfrac,bias1,sfac1,bias2,sfac2);
RooBMixDecay unmixed("unmixed","Unmixed Events",dtrec,tau,dm,wlep,
                     resolution,RooBMixDecay::Unmixed);
```

where the first line declares the resolution model, and the second line applies this resolution model to a particular physics model.

The sections below describe the various physics and resolution models which have been implemented. In order to factorize these models, we must assume a definite set of basis functions, $u_k(t\,;\vec{p}_k)$, for the physics models

$$f_0(t\,;\tau,\vec{p}) = \sum_k c_k(\vec{p}_k) \cdot u_k(t\,;\vec{p}_k) \; .$$

For the models which are now implemented, the functions $\{1, \cos(\omega t), \sin(\omega t)\}$ are a sufficient set of basis functions.

4.2.1 Physics Models

A physics model is specified by its modulation function, $g(s\,;\vec{p})$. The simplest choice is $g = 1$ which gives the distribution for a pure lifetime decay. Create this model using

```
RooDecay Blife("Blife","B0 Lifetime Decay",dt,tauB,resolution);
RooDecay Dlife("Dlife","D0 Lifetime Decay",t,tauD,resolution,
               RooDecayPdf::SingleSided);
```

where the first form assumes the default unrestricted range of the decay-time variable, and the second form specifies a single-sided distribution restricted to $s \geq 0$ (you could also use `RooDecayPdf::Flipped` for a distribution restricted to $s < 0$.) In both of these examples, `resolution` is the name of a previously-created resolution model (see Section 4.2.2). The dependent variable in these examples is `dt` or `t`, and the single lifetime parameter is `tauB` or `tauD`. Figure 3 shows some examples of this physics model using different resolution models, and was created with the macro in `examples/lifetime.cc`.

The remaining physics models involve an oscillating term in the modulation function. First, we consider B^0-$\overline{B^0}$ mixing which can be parameterized using

$$g_{\pm}(t\,;\tau,\Delta m,\eta) = 1 \pm (1-2\omega)\cos(\Delta m t) \; ,$$

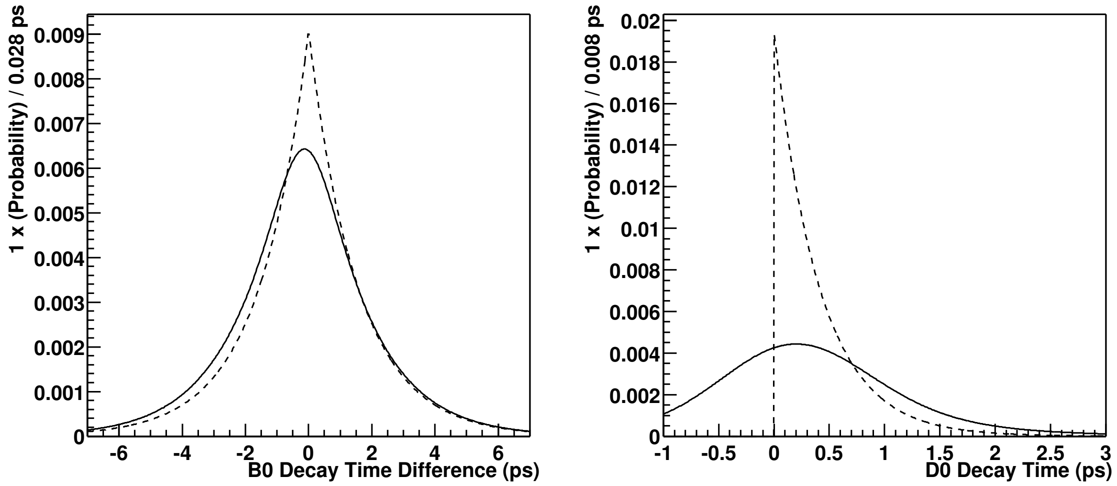


Figure 3: Examples of lifetime decay distributions modeled with the `RooDecay` class. The left-hand plot shows the default double-sided distribution, and the right-hand plot shows a single-sided distribution created with the `RooDecayPdf::SingleSided` option. The dashed curves show the models created with a "truth" unsmeared resolution model, and the solid curves use a realistic double-Gaussian resolution model.

where the $\pm$ selects between models for unmixed $(+)$ and mixed $(-)$ events, Δm is the oscillation frequency, and ω (the "mistag rate") is the probability for an unmixed (mixed) event to be incorrectly tagged as a mixed (unmixed) event (these are actually two probabilities which we assume to be identical). Create this mixing model using

```
RooBMixDecay Bunmix("Bunmix","Unmixed BB Decays",dt,tau,dm,omega,
                    resolution,RooBMixDecay::Unmixed);
RooBMixDecay Bmixed("Bmixed","Mixed BB Decays",dt,tau,dm,omega,
                    resolution,RooBMixDecay::Mixed);
```

where the choice of sign in the modulation function is specified after the resolution model: use `RooBMixDecay::Unmixed` for unmixed events, and `RooBMixDecay::Mixed` for mixed events. As in the lifetime case (and for all decay-time PDFs), you can add an aditional parameter to select the range of t to use (in this example, we use the default `RooDecayPdf::DoubleSided`). Figure 4 shows some examples of this physics model using different resolution models, and was created with the macro in `examples/bmixing.cc`. There is also the possibility to express the dilution as:
1-2ω=(1-2Rω_1)(1-2ω_2)= 1-2(Rω_1+ω_2)+4R$\omega_1\omega_2$
ω_1 and ω_2 have different origin and give a resulting dilution which is the product of the two (e.g mistag due to non perfect PID and to charm cascades).

R is a scale factor which allows to express a mistag as a function of another (and so to reduce the number of parameters). To create this model, use:

```
RooBMixDecayDilu Bunmix("Bunmix","Unmixed BB Decays",dt,tau,dm,omega1,
                         R,omega2,resolution,RooBMixDecay::Unmixed);
RooBMixDecayDilu Bmixed("Bmixed","Mixed BB Decays",dt,tau,dm,omega1,
                         R,omega2,resolution,RooBMixDecay::Mixed);
```

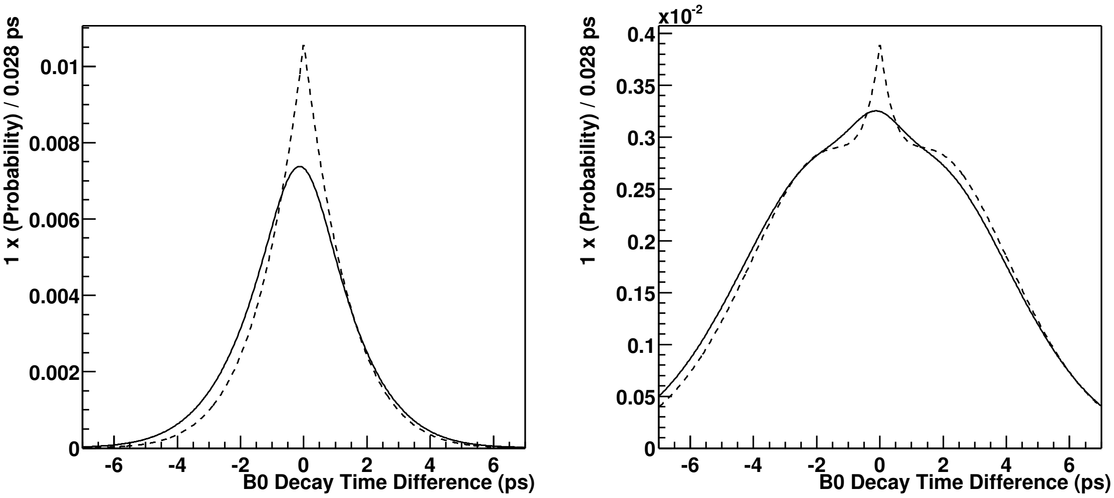


Figure 4: Examples of B^0 mixing distributions modeled with the `RooBMixDecay` class. The left-hand plot shows distributions for unmixed (+1) decays, and the right-hand plot shows mixed (−1) decays. All distributions are shown with a mistag rate of 10%. The dashed curves are calculated using a "truth" unsmeared resolution model, and the solid curves use a realistic double-Gaussian resolution model.

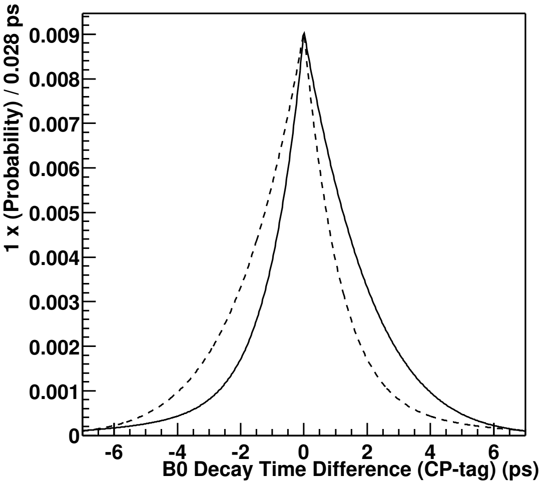
Next, we consider the case of CP violating asymmetries in B^0 decays, which can be modeled by

$$g_\pm(t\,;\tau,\Delta m,\omega,b) = 1 \mp \eta_{CP} b(1-2\omega)\sin(\Delta m t) \ ,$$

where the new parameters relative to the B^0-$\overline{B^0}$ mixing case are the CP eigenvalue of the final state being considered, η_{CP}, and the strength of the asymmetry, b. For $B^0,\overline{B^0} \to$ $J/\psi K_S$ decays, $\eta_{CP} = -1$ and $b = \sin(2\beta)$. Create this model using the constructors

```
RooBCPDecay BCP("BCP","CP Asymmetry in BB Decays",dt,tau,dm,omega,etaCP,
                 sin2beta,resolution,RooBCPDecay::Minus);
RooBCPDecay BCP("BCP","CP Asymmetry in BB Decays",dt,tau,dm,omega,etaCP,
                 sin2beta,resolution,RooBCPDecay::Plus);
```

The final parameter specifies the flavor of the B decaying into the CP eigenstate at $t = 0$ (which is opposite to the flavor of the tagging B in the case of B^0-$\overline{B^0}$ decays): B^0 (-1) or $\overline{B^0}$ ($+1$). The mistag rate, ω, is the probability that this flavor is incorrectly assigned (again, there are really two probabilities which we assume to be equal). Figure 5 shows some examples of this physics model using different resolution models, and was created with the macro in `examples/cpasymm.cc`.



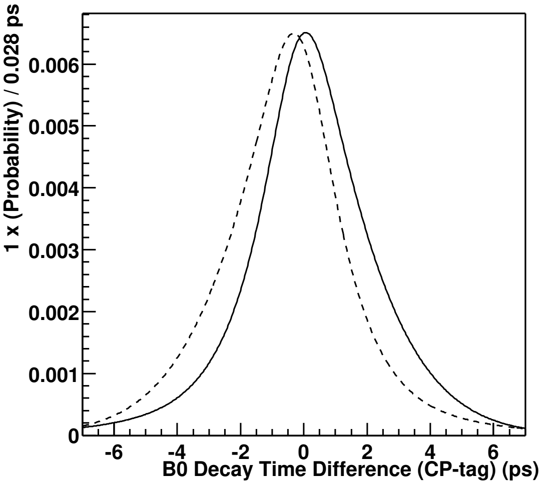


Figure 5: Examples of CP asymmetry in B^0-$\overline{B^0}$ decays modeled with the `RooBCPDecay` class. The left-hand plot shows distributions calculated with a "truth" unsmeared resolution model, and the right-hand plot use a realistic double-Gaussian resolution model. All distributions are calculated using $\sin(2\beta) = 0.5$, and a mistag rate of 10%. The solid curves are for a $\overline{B^0}$ decay into a CP eigentstate and the dashed curves are for a B^0 decay.

We can write a more general form of the CP violating asymmetry with

$$g_{\pm}(t\,;\tau,\Delta m,\omega,a,b) = \frac{1+a^2}{2} \mp (1-2\omega)\left[\eta_{CP} a\, b\sin(\Delta mt) + \frac{1-a^2}{2}\cos(\Delta mt)\right] ,$$

which reduces to the case above when $a^2 = 1$. In this case, the parameters a and b are related to a general CP-violating amplitude λ_{CP} by

$$a = |\lambda_{CP}| \quad \text{and} \quad b = \sin(\arg(\lambda_{CP})) .$$

Add an additional parameter in the constructor to use this more general form

```
RooBCPDecay BCP("BCP","CP Asymmetry in BB Decays",dt,tau,dm,eta,a,b,
                resolution,RooBCPDecay::Minus);
RooBCPDecay BCP("BCP","CP Asymmetry in BB Decays",dt,tau,dm,eta,a,b,
                resolution,RooBCPDecay::Plus);
```

where all other parameters have the same meaning as above.

4.2.2 Resolution Models

With our choice of modulation basis functions, $\{1, \cos(\omega t), \sin(\omega t)\}$, the integrals we need to evaluate in order to describe a resolution model are

$$f_\pm(t\,;\tau,\omega,\vec{q}) = \frac{1}{2\tau}\int_0^\infty ds\, \exp(-s/\tau)\, \exp(\pm i\omega s)\, h(t\mp s\,;\vec{q}) \; ,$$

and

$$F_\pm(t_1,t_2\,;\tau,\omega,\vec{q}) = \int_{t_1}^{t_2} dt\, f_\pm(t\,;\tau,\omega,\vec{q}) \; .$$

With these integrals in hand, we take $\omega = 0$ for $g = 1$, and use the real or imaginary parts for $g = \cos(\omega t)$ and $g = \sin(\omega t)$, respectively. The first integral is used to evaluate the smeared physics model, and the second integral is used to normalize it over a finite domain, $[t_1, t_2]$.

The simplest scenario we consider is a model with no smearing but a possible bias, t_0,

$$h(t\,;t_0) = \delta(t - t_0)$$

This model is useful for directly viewing the underlying physics model (although most models will reduce to this case with certain choices of parameters, they may not be numerically stable in this limit). In this case, we have

$$f_\pm(t\,;\tau,\omega,t_0) = \frac{1}{2\tau}\exp(-t_\pm/\tau)\, \exp(\pm i\omega t_\pm)\,\cdot\,\theta(t_\pm > 0) \; ,$$

and

$$F_\pm(t_1,t_2\,;\tau,\omega,t_0) = \mp\frac{1}{2z}\left[\exp\left(\frac{-zy_2}{\tau}\right) - \exp\left(\frac{-zy_1}{\tau}\right)\right] \; ,$$

where

$$t_\pm \equiv \pm(t - t_0) \quad , \quad y_k \equiv \max(0, \pm(t_k - t_0)) \quad , \quad z \equiv 1 \mp i\omega\tau \; .$$

Create this resolution model using the constructor

```
RooTruthModel truth("truth","Truth Resolution Model",residual);
RooTruthModel truth("truth","Truth Resolution Model",residual,bias);
```

where `residual` is the model's dependent variable and the optional parameter `bias` specifies the value of t_0. Note that the variable `residual` is not the same as the dependent variable of a physics model which uses this resolution model: you should declare both of these, as separate `RooRealVar`'s. The choice of range for the residual variable only affects how a plot of the resolution model is displayed (see below).

The next resolution function we consider is a sum of Gaussians, each with its own bias, δ_k, and width, σ_k,

$$h(t\,;\vec{f},\vec{\delta},\vec{\sigma}) = \sum_{k}^{n} f_k h_G(t\,;\delta_k,\sigma_k) \;,$$

with relative-fraction parameters, $\{f_1, f_2, \ldots, f_{n-1}\}$, used to calculate

$$f_n \equiv 1 - \sum_{k=1}^{n-1} f_k \;,$$

and

$$h_G(t\,;\delta,\sigma) \equiv \frac{1}{\sqrt{2\pi}\sigma}\;\exp(-(t-\delta)^2/(2\sigma^2)) \;.$$

In this case, we can write

$$f_\pm = \sum_k f_\pm^{(k)} \quad \text{and} \quad F_\pm = \sum_k F_\pm^{(k)} \;.$$

We can evaluate the necessary integrals analytically, obtaining

$$f_\pm^{(k)}(t\,;\tau,\omega,\delta,\sigma) = \frac{1}{4\tau}\;\exp\left(z\left(s_\pm + c^2 z\right)\right)\;\mathrm{erfc}\left(\frac{s_\pm}{2c} + cz\right)$$

and

$$F_\pm^{(k)}(t_1,t_2\,;\tau,\delta,\sigma) = \mp\frac{\exp(c^2z^2)}{4z}\;[\mathrm{erf}(y_2/2c) - \mathrm{erf}(y_1/2c) + \\ \exp(zy_2)\,\mathrm{erfc}\left(\frac{y_2}{2c} + cz\right) - \exp(zy_1)\,\mathrm{erfc}\left(\frac{y_1}{2c} + cz\right)\;]\;,$$

with (z is defined above)

$$s_\pm \equiv \mp\frac{t-\delta}{\tau} \quad , \quad y_k \equiv \mp\frac{t_k-\delta}{\tau} \quad , \quad c \equiv \frac{\sigma}{\sqrt{2}\tau} \;.$$

To create a resolution model with up to three Gaussian contributions, use a constructor like

```
RooGaussModel gauss1("gauss1","Single-Gaussian Resolution Model",
                     residual,sig0,bias1,sfac1);
RooGaussModel gauss2("gauss2","Double-Gaussian Resolution Model",
                     residual,sig0,frac,bias1,sfac1,bias2,sfac2);
RooGaussModel gauss3("gauss3","Triple-Gaussian Resolution Model",
                     residual,sig0,frac1,frac2,bias1,sfac1,
                     bias2,sfac2,bias3,sfac3);
```

where `residual` is the model's dependent variable and the comments above apply here too. The parameters `frac` and `frac1`, `frac2` are the relative fractions for the double- and triple-Gaussian models, respectively. The RMS parameters for each Gaussian, σ_k, are factorized into a base resolution parameter, `sig0`, which is common to all three models, and a scale factor, s_k (`sfac1`, etc), for each contribution

$$\sigma_k \equiv s_k \cdot \sigma_0 \ .$$

The reason for this factorization is to allow decay-time models which use this resolution model to be easily adapted for per-event errors (see Section 4.4.) By default, the bias parameters (`bias1`, `bias2`, and `bias3`) are interpreted as absolute biases defined in the same units as the residual. However, as an alternative you can specify that these parameters are also relative to the base resolution parameter, `sig0`, by adding the optional `RooGaussModel::ScaledBias` parameter at the end of the constructor

```
RooGaussModel gauss1("gauss1","Single-Gaussian Resolution Model",
                     residual,sig0,bias1,sfac1,RooGaussModel::ScaledBias);
```

This form of the resolution model (when combined with per-event errors) is appropriate if the parameters are obtained from a fit to a pull distribution.

The third resolution function we consider ("GExp") is the sum of an unbiased gaussian with width σ, and the same gaussian convoluted with an exponential $e(t;\tau_r)$:

$$h_{\mathrm{GExp}}(t;f_c,\sigma,\tau_r) = f_c h_G(t;\delta=0,\sigma) + (1-f_c)\, h_{\mathrm{Exp}}(t;\tau_r)$$

with

$$h_{\mathrm{Exp}}(t;\tau_r) = \int_{-\infty}^{\infty} du\, e(u;\tau_r)\, h_G(t-u;\delta=0,\sigma)$$

and

$$e(t;\tau_r) = \begin{cases} \frac{1}{\tau_r}\exp\left(\frac{t}{\tau_r}\right) & t \le 0 \\ 0 & t > 0 \ . \end{cases}$$

The parameter τ_r is the "lifetime" of the exponential and f_c is the fraction of events in the central gaussian. We have already evaluated the integrals necessary for the first contribution to $h_{\mathrm{GExp}}(t;f_c,\sigma,\tau_r)$, namely $f_\pm^{(k)}(t\,;\tau,\omega,\delta,\sigma)$. The integrals necessary for the second contribution can be expressed in terms of other integrals that we have already calculated [6]. We need to evaluate

$$f_\pm^{\mathrm{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \\ \frac{1}{2\tau}\int_0^\infty ds\, \exp\left(-\frac{s}{\tau}\right)\cdot\exp\left(\pm i\omega s\right)\cdot\int_{-\infty}^0 du\,\frac{1}{\tau_r}\exp\left(\frac{u}{\tau_r}\right)\cdot h_G(t\mp s-u;\delta=0,\sigma)\ .$$

As outlined in appendix C, this can be expressed in terms of $f_\pm^{(k)}(t\,;\tau,\omega,\delta,\sigma)$:

$$f_\pm^{\mathrm{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \frac{1}{\frac{1}{\tau_r}\pm\frac{1}{\tau}-i\omega}\cdot\left[\frac{1}{\tau_r}f_\pm^{(k)}(t\,;\tau,\omega,\delta=0,\sigma)\pm\frac{1}{\tau}f_-^{(k)}(t\,;\tau_r,0,0,\sigma)\right]\ .$$

Under certain conditions, this expression exhibits a mathematical pathology that can be circumvented by performing a Taylor expansion of the $f_-^{(k)}$; for details see appendix C. To create a "GExp" resolution model, use a constructor like

```
RooGExpModel gexp("gexp","GExp Resolution Model",
                  residual,sig0,fc,sigma,taur);
RooGExpModel gexp("gexp","GExp Resolution Model",
                  residual,sig0,fc,sigma,taur,kTRUE);
```

where the first variant uses only the leading order term of the Taylor expansion, whereas the second variant also takes into account the next-to-leading-order correction. The variables `residual` and `sig0` have the same meaning as in the previous examples. This model can be used with per-event errors in the same way as the `RooGaussModel`. In the case of the "GExp" model, σ plays the role of the scale factor, and the values for f_c, σ and τ_r can be obtained from a simple fit to a pull distribution.

In addition to its role in building a complete decay-time distribution, a resolution model is also a self-contained PDF in its residual variable. This means, for example, that you can plot a model by itself, or fit it to a pull distribution.

4.3 Building Complex Models

RooFitTools provides several methods to combine PDFs and construct more complex PDFs. The PDFs being combined can either be predefined or else themselves combinations of PDFs. Each combination method is described below.

4.3.1 Adding Distributions

Two PDFs can be added to construct a new PDF

$$F_{12}(\vec{x}_{12}\,;\vec{p}_{12},f) = \frac{f}{N_1}\cdot F_1(\vec{x}_1\,;\vec{p}_1) + \frac{1-f}{N_2}\cdot F_2(\vec{x}_2\,;\vec{p}_2)$$

using

```
RooAddPdf F12("F12","Signal + Background Model",F1,F2,f);
```

where f is a new parameter introduced to set the relative normalization of F_1 and F_2, and $\vec{x}_{12}$ ($\vec{p}_{12}$) represents the union of the dependent variables (parameters) in F_1 and F_2. There are no restrictions on the dependents and parameters used in F_1 and F_2. In particular, some parameters can be common to both, and the dependent variables need not match between them (in this case, any missing dependents are assumed to be uniformly distributed for that PDF). The normalization factors, N_1 and N_2, are both one if F_1 and F_2 depend on the same variables; otherwise, they are a product of normalization factors for the (uniformly distributed) variables which each PDF does not explicitly depend on (j and k index these variables)

$$N_1 = \prod_j \left[x_j(max) - x_j(min)\right] \ , \ N_2 = \prod_k \left[x_k(max) - x_k(min)\right]$$

The minimum value of f must be ≥ 0 and the maximum value must be ≤ 1. Note that since PDFs are normalized in the closed interval ranges specified for each dependent variable, the value of f gives the fraction of F_1 contributing to F_{12} within this finite region. Figures 6 and 7 show examples of a fit models constructed using addition of two PDFs.

The generalization to addition of more than two PDFs is

$$F(\vec{x}_{12\ldots n}\,; \vec{p}_{12\ldots n}, f_1, f_2, \ldots, f_{n-1}) = \sum_{k=1}^{n} \frac{f_k}{N_k} \cdot F_k(\vec{x}_k\,; \vec{p}_k) \ ,$$

where the relative fraction of the last PDF, F_n, is

$$f_n \equiv 1 - \sum_{k=1}^{n-1} f_k \ .$$

Use the following constructors for addition of up to five PDFs

```
RooAddPdf("add3","F1+F2+F3",F1,F2,F3,f1,f2);
RooAddPdf("add4","F1+F2+F3+F4",F1,F2,F3,F4,f1,f2,f3);
RooAddPdf("add5","F1+F2+F3+F4+F5",F1,F2,F3,F4,F5,f1,f2,f3,f4);
```

An extension of the `RooAddPdf` class to allow an arbitrary number of PDFs to be added is planned but not yet implemented.

The `printEventStats()` utility method described in earlier versions of this guide is now deprecated. Use the extended maximum likelihood techniques described in Section 4.5 instead.

4.3.2 Multiplying Distributions

Up to six independent distributions can be multiplied to construct a new PDF

$$F(\vec{x}_1, \vec{x}_2, \ldots \vec{x}_n\,; \vec{p}_{12\ldots n}) = \prod_{k=1}^{n} F_k(\vec{x}_k\,; \vec{p}_k)$$

using the constructors

```
RooProdPdf F12("F12","F1*F2",F1,F2);
RooProdPdf F123("F123","F1*F2*F3",F1,F2,F3);
RooProdPdf F1234("F1234","F1*F2*F3*F4",F1,F2,F3,F4);
RooProdPdf F12345("F12345","F1*F2*F3*F4*F5",F1,F2,F3,F4,F5);
RooProdPdf F123456("F123456","F1*F2*F3*F4*F5*F6",F1,F2,F3,F4,F5,F6);
```

The dependent variables for the F_k must not overlap, but there are no restrictions on their parameters. Section A.4 provides an example of using PDF products. There is also the possibility to multiply any number of PDFs using the new constructor

```
RooProdPdf FProd("FProd","Product of PDFs", pdfArray);
```

where `pdfArray` is a ROOT `TObjArray` containing the PDFs to be multiplied:

```
TObjArray pdfArray(10);
pdfArray.Add(&F1);
pdfArray.Add(&F2);
.
.
.
pdfArray.Add(&F10);
```

(The ampersand symbol, &, is required for the `Add` function because the `RooAbsPdf` objects are not pointers in this example).

4.3.3 Convoluting Distributions

Two PDFs can be convoluted to create a new PDF

$$F_{12}(t', \vec{x}_{12}\,; \vec{p}_{12}) = \int_{s_{min}}^{s_{max}} F_1(t, \vec{x}_1\,; \vec{p}_1)|_{t=t'+s} \cdot F_2(s, \vec{x}_2\,; \vec{p}_2)\, ds$$

using the constructor

```
RooConvPdf F12("F12","Smeared Lifetime Distribution",F1,F2,s,t,tp);
```

A typical application for convolution is to apply Gaussian resolution smearing, but this package allows arbitrary convolutions (e.g., to model non-Gaussian tails or biases). Note that both t and t' must be specified when constructing a convolution and that these are distinct because they have different ranges:

$$t'_{min} = t_{min} - \max(0, s_{max})\ ,\ t'_{max} = t_{max} - \min(0, s_{min})$$

The range of t' will be set to these values, overriding the existing range if necessary. For this reason, it is convenient to construct `tp` as a copy of `t` like this

```
RooRealVar tp(t);
```

There is no requirement that $s_{min} = -s_{max}$ or that F_2 have zero mean.

Note that a PDF created using a numerical convolution will generally be quite slow to evaluate. In some cases, the integrals involved can be done analytically and you can improve the performance of a PDF by implementing the analytic results directly in a new PDF. See Section B for general instructions on adding a new PDF, and Section 4.2.2 for a description of the RooFitTools framework for efficiently implementing decay-time PDFs.

4.4 Models which Depend on a Per-Event Error

Fit models often include a parameter, σ, which describes the resolution with which a dependent variable, x, is measured

$$F(x, \ldots; \sigma, \vec{p}) \; .$$

In cases where the value of this resolution parameter can be estimated on an event-by-event basis, and the distribution of this event-by-event error is sufficiently wide, we can improve the statistical error of the fit by replacing the parameter σ_x by a new dependent variable, $\tilde{\sigma}$,

$$F(x, \ldots; \sigma, \vec{p}) \to \tilde{F}(x, \tilde{\sigma}, \ldots; \vec{p}) \, \rho(\tilde{\sigma}\,; \vec{q}) \; .$$

The function $\rho(\tilde{\sigma}\,; \vec{q})$ multiplying the new PDF models the distribution of the new dependent variable, $\tilde{\sigma}$, and we assume that its parameters, $\vec{q}$, are independent of the original PDF parameters, $\vec{p}$. The function $\tilde{F}$ matches F at any given value of $\sigma = \tilde{\sigma}$:

$$\tilde{F}(x, \tilde{\sigma}, \ldots; \vec{p}) = \left. F(x, \ldots; \sigma, \vec{p}) \right|_{\sigma=\tilde{\sigma}} \; ,$$

and so has an implicit dependence on $\tilde{\sigma}$ due to the normalization factor for F.

You can transform any PDF of a global resolution parameter into a corresponding PDF that models per-event errors using:

```
RooEvtErrors perEvtPDF("perEvtPDF","Model Using Per-Event Errors",
                       PDF,errorsModel,error,sigma);
```

where `PDF` is a model with a parameter `sigma` and the result is a new model, `perEvtPDF`, which uses the per-event dependent variable `error`. In this example, the variables `error` and `sigma` should have the same ranges, with a minimum value greater than zero (the `RooEvtErrors` constructor will check this).

The construction of a per-event PDF described above assumes that the distribution of per-event errors, ρ, is not correlated with the distribution of the other dependent variables. When this is true, the results of a fit to the parameters, $\vec{p}$, of the original PDF, $F(x, \ldots; \sigma, \vec{p})$, do not depend on ρ: different choices of ρ correspond to global translations of the likelihood surface. This means that you fit using $\rho = 1$, which is the default when you omit the `errorsModel` parameter in the constructor:

```
RooEvtErrors perEvtPDF("perEvtPDF","Model Using Per-Event Errors",
                       PDF,error,sigma);
```

Using this approach will speed up the fit somewhat, but you should include a model of per-event errors to generate a realistic toy Monte Carlo sample (see Section 4.8.4).

The examples above reduce the number of parameters in a fit by one, by eliminating `sigma`. It is often interesting to add a new parameter `sfac` to the fit which is interpreted as a global scale factor, s, that multiplies each per-event error,

$$\sigma = s \cdot \sigma_k \; .$$

You can do this using the constructors (with or without an explicit model for the per-event errors):

```
RooRealVar sfac("sfac","Per-Event Error Global Scale Factor",0.5,1.5);
RooEvtErrors perEvtPDF("perEvtPDF","Model Using Per-Event Errors",
                       PDF,errorsModel,error,sigma,sfac);
RooEvtErrors perEvtPDF("perEvtPDF","Model Using Per-Event Errors",
                       PDF,error,sigma,sfac);
```

Note that `RooEvtErrors` can be used with the decay-time resolution models described in Section 4.2.2, but that those models define their own scale factors and so no additional scale factor should be applied in the `RooEvtErrors` constructor.

Making plots of PDFs that use per-event errors is complicated by the fact that these PDFs are at least two-dimensional. In the case of PDFs which are exactly two-dimensional (for example, a `RooDecay` lifetime distribution with per-event errors), you are usually interested in the projection of the PDF onto the time axis, which involves integrating out the per-event error variable for each bin of the histogram. This can be done using the `Project` methods described in Section 4.8.2.

4.5 Extended Maximum-Likelihood Fits

The likelihood calculated with a normalized PDF estimates the probability of obtaining the measured *shapes* in the distributions of the dependent variables in a sample of events, given some parameter values. This approach implicitly assumes that the total number of events expected is independent of the parameter values.

If we want to also determine a parameter corresponding to the expected number of events, η, we should add a factor to the likelihood which describes the probability of observing the actual number of events, N, given this parameter. This probability is described by the Poisson distribution

$$P(N\,;\eta) = \frac{1}{N!}\cdot \eta^N \, \exp(-\eta) \ ,$$

and we refer to the likelihood including this factor as the "extended likelihood"

$$\tilde{\mathcal{L}}(\vec{x}, N\,;\vec{p}, \eta) \equiv P(N\,;\eta)\cdot\mathcal{L}(\vec{x}\,;\vec{p}) \ .$$

A fit to a single dataset using this extended likelihood yields exactly the same estimates of the original parameter values and errors, as well as the additional estimate

$$\eta = N \pm \sqrt{N} \ ,$$

and so does not add any useful information. This approach does become interesting, however, when η is allowed to be a function of other parameters. To allow for this, any building-block PDF can implement a method, `expectedEvents()`, which returns the expected number of events with the current parameter values. When fitting with the 'E' option ('E' stands for extended), this value is used to calculate an extended maximum likelihood for use in the fit.

4.6 Maximum-Likelihood Fits for Branching Ratio Measurements

The first step towards an analysis usually involves the cut and count method, whereby progressive cuts are made to some data to yield a branching ratio measurement, for example. A better way would be to use (unbinned) maximum likelihood fits, since these can take into account any correlations between the distributions of various discriminating variables used in an analysis, improving the reconstruction efficiency and background rejection. This can be done by using the `RooBRArrayPdf` class.

The aim of the maximum-likelihood method is to find the number of signal and background events in a given data sample that maximises the likelihood function, $\mathcal{L}$:

$$\mathcal{L} = \frac{e^{-\sum n_j}}{N!} \prod_{i=1}^{N} \sum_{j=1}^{m} n_j \, \mathcal{P}_j$$

where $\mathcal{P}_j$ is the total probability for each of the m (signal and background) hypotheses, n_j is the number of events for the j^{th} hypothesis, and N is the total number of input data points. Note that the above expression represents the extended likelihood function, since it contains the relevant Poisson term. The probabilities $\mathcal{P}_j$ are just the total PDFs that represent the probability of observing an event of type j, based on a set of discriminating variables. Usually, the discriminating variables that are used are uncorrelated (or assumed to be), so that the total PDF for a given hypothesis is just the product of the individual PDFs of the discriminating variables (see section 4.3.2):

$$\mathcal{P}_j = \prod_{v=1}^{V} \mathcal{P}_v$$

where $\mathcal{P}_v$ is the PDF for the v^{th} discriminating variable for the j^{th} hypothesis.

Before doing any fitting, a preliminary set of cuts is usually imposed to clean up the data sample. However, care must be taken to ensure that the cuts do not impose any bias in the distributions of the discriminating variables that are used in the fit. After the relevent PDFs have been found, they are used to form the likelihood function. The fit then finds the values of n_j, the number of events for each hypothesis j.

The constructor of `RooBRArrayPdf` takes two ROOT `TObjArrays`. The first stores a set of `RooAbsPdfs`, which represent the total PDFs $\mathcal{P}_j$ (`pdfArray`), while the second stores a set of `RooRealVars`, which represent the number of events to be fit, n_j (`coefArray`):

```
RooBRArrayPdf model("model", "BR Model", pdfArray, coefArray);
```

We can then perform an extended maximum likelihood fit to the data stored in a given `RooDataSet` object, `data`, by calling the usual function:

```
model.fitTo(data, "e");
```

Consider the case where we want to fit for the number of $B \to \rho\pi$ and $B \to \rho K$ events in a data sample. Let the total PDF for the $B \to \rho\pi$ and $B \to \rho K$ hypotheses be represented by the `RooAbsPdf` objects `Prhopi` and `PrhoK`, respectively. Let us also assume that we have two background contributions, one containg a pion track, the other a kaon track, represented by the `RooAbsPdf` objects `PBgpi` and `PBgK`, respectively. We now form the `TObjArray` of the PDFs:

```
TObjArray pdfArray(4);

pdfArray.Add(&Prhopi);
pdfArray.Add(&PrhoK);
pdfArray.Add(&PBgpi);
pdfArray.Add(&PBgK);
```

and the `TObjArray` of the number of events (each represented by a `RooRealVar` object):

```
TObjArray coefArray(4);

coefArray.Add(&Nrhopi);
coefArray.Add(&NrhoK);
coefArray.Add(&NBgpi);
coefArray.Add(&NBgK);
```

Note that the ordering of the hypotheses must be the same for both arrays. Also, if the objects going in the `TObjArrays` are pointers, the ampersand (&) symbol is not required in the `Add` function. The model can then be formed by doing

```
RooBRArrayPdf model("model", "BR Model", pdfArray, coefArray);
```

After the fit, the array `coefArray` will contain the numbers of events for each of the various hypotheses. It is sometimes necessary to set limits and provide initial estimates of the errors for the coefficents in `coefArray` in order for the fit to behave properly. For example,

```
Nrhopi.setLimits();     // Set the limits to be those provided in the
                        // RooRealVar constructor of Nrhopi (say, 0 to N)
Nrhopi.SetError(1e-2); // Set an initial 0.01 error for the number.
```

An example of using this class can be seen in `RooFitMacros/BBDecays/Charmless/TestD0piFit.cc` in the package `RooFitMacros V00-00-18`. A more extensive description of the maximum likelihood method (applied to quasi-two-body charmless B decays) can be found in [12], although they do not use the class described here.

4.7 Simultaneous Fits to Independent Datasets

We perform a simultaneous fit of two independent samples, $\{\vec{x}_i\}_{i=1}^{n}$ and $\{\vec{y}_j\}_{j=1}^{m}$, to determine a common set of parameters, $\vec{p}$, by maximizing the product of the two likelihoods

$$\mathcal{L}_{12}(\vec{p}) = \mathcal{L}_1(\vec{p}) \cdot \mathcal{L}_2(\vec{p}) \ ,$$

where

$$\mathcal{L}_1(\vec{p}) = \prod_{i=1}^{n} F_1(\vec{x}_i\,;\vec{p}) \quad , \quad \mathcal{L}_2(\vec{p}) = \prod_{j=1}^{m} F_2(\vec{y}_j\,;\vec{p}) \ .$$

This combined likelihood can be expressed in terms of a combined PDF, $F_{12}(\vec{z}\,;\vec{p})$, as

$$\mathcal{L}_{12}(\vec{p}) = N_{12} \cdot \prod_{k=1}^{n+m} F_{12}(\vec{z}_k\,;\vec{p}) \ ,$$

where $\vec{z}$ is the union of dependent variables used in the two data sets[4], together with a new discrete "category index" dependent variable c,

$$\vec{z} = (\vec{x}, \vec{y}, c) \ ,$$

with $c = 1$ for events in the first dataset, and $c = 2$ for events in the second dataset. The combined dataset, $\{\vec{z}_k\}_{k=1}^{n+m}$, is the concatenation of the two independent data sets,

$$\begin{aligned}
\vec{z}_1 &= (\vec{x}_1, \vec{0}, 1) \\
&\quad \ldots \\
\vec{z}_n &= (\vec{x}_n, \vec{0}, 1) \\
\vec{z}_{n+1} &= (\vec{0}, \vec{y}_1, 2) \\
&\quad \ldots \\
\vec{z}_{n+m} &= (\vec{0}, \vec{y}_m, 2)
\end{aligned}$$

The combined PDF is defined as

$$F_{12}(\vec{z}\,;\vec{p}) \equiv \frac{\Delta(c\,;1)}{2N_y} \cdot F_1(\vec{x}_k\,;\vec{p}) \ + \ \frac{\Delta(c\,;2)}{2N_x} \cdot F_2(\vec{y}_k\,;\vec{p}) \ ,$$

where the constants,

$$N_x \equiv \int 1 \cdot d\vec{x} \ \text{ and } \ N_y \equiv \int 1 \cdot d\vec{y} \ ,$$

[4] We can always define the two sets of variables to be distinct, even if some have the same definition.

are defined so that F_{12} is normalized according to

$$1 = \int F_{12}(\vec{z}\,;\vec{p})\,d\vec{z} \;=\; \sum_{c=1}^{2} \int d\vec{x} \int d\vec{y}\; F_{12}(\vec{x},\vec{y},c\,;\vec{p}) \;.$$

and $\Delta(c\,;n)$ is defined to be one if $c = n$ or otherwise zero. With these definitions, the factor N_{12} appearing above is

$$N_{12} = 2^{n+m} \cdot N_x^m \cdot N_y^n \;,$$

and is independent of the parameters so can be ignored for the purposes of maximizing the likelihood.

Up to this point, we have described a simultaneous fit to the *shapes* of the distributions of dependent variables in the two datasets, but which does not exploit any information that may be contained in the relative sizes of the two samples. To include this information, we define a "simultaneous extended likelihood"

$$\tilde{\mathcal{L}}_{12}(\vec{p}) \equiv \tilde{\mathcal{L}}_1(\vec{p}) \cdot \tilde{\mathcal{L}}_2(\vec{p}) = \mathcal{L}_{12}(\vec{p}) \cdot P(N_1\,;\eta_1) \cdot P(N_2\,;\eta_2) \;.$$

We can re-write the product of the Poisson probabilities in this expression as the product of a single Poisson distribution in the total number of observed events, $N = N_1 + N_2$, multiplied by the Binomial probability of the observed fraction, N_1/N, of events in the first dataset

$$\begin{aligned} P(N_1\,;\eta_1) \cdot P(N_2\,;\eta_2) &= \frac{\eta_1^{N_1}}{N_1!}\,\exp(-\eta_1) \cdot \frac{\eta_2^{N_2}}{N_2!}\,\exp(-\eta_2) \\ &= \frac{\eta^N}{N!}\,\exp(-\eta) \cdot \frac{N!}{N_1!N_2!}\left(\frac{\eta_1}{\eta}\right)^{N_1}\left(\frac{\eta_2}{\eta}\right)^{N_2} \\ &= P(N\,;\eta) \cdot B(N_1, N\,;f) \;, \end{aligned}$$

where $\eta \equiv \eta_1 + \eta_2$, $f \equiv \eta_1/\eta$, and the Binomial distribution is

$$B(N_1, N\,;f) = \frac{N!}{N_1!(N-N_1)!}\,f_1^N (1-f)^{N-N_1} \;.$$

In the case where η and f are considered as new parameters, the use of a simultaneous extended likelihood does not change the estimated values or errors for the original parameters, $\vec{p}$, and yields

$$\eta = N \pm \sqrt{N} \;\text{ and }\; f = \frac{N_1}{N} \pm \sqrt{\frac{N_1(N-N_1)}{N^3}} \;.$$

However, the interesting case is where the fraction, f, expected in the first dataset is itself a function of the other parameters, $\vec{p}$, and not an independent parameter

$$f = f(\vec{p}) \;.$$

4.8 Tools for Working with PDFs

This section describes some general tools for working with any PDF in RooFitTools. The examples below use the name `PDF` for the distribution but any name will work. Use the common base class for all PDFs, `RooAbsPdf` if you want to write a general-purpose subroutine for PDFs in your macros (note the ampersand for a C++ reference)

```
myPlotMaker(RooAbsPdf &PDF) { ... }
```

4.8.1 Getting Information

Once you have create a PDF, you can check what dependent variables and parameters it uses with

```
PDF.Print()
```

which produces output like this (a semicolon separates the dependents and the parameters)

```
PDF model(m;bmass,bresn,endpt,c,f)
```

The variable names appearing here are those specified in the constructor which, by convention, match the object names. To trace the evaluation of a PDF, use the `PDF.setTrace()` method: without any argument, the next evaluation will be traced like this

```
[1] model(m=5289;bmass=5279,bresn=3,endpt=5290,c=-50,f=0.5) = 0.00142
```

With an integer argument n, the next n evaluations will be traced. You can also evaluate a PDF directly using the current values of its dependent variables and parameters

```
m = 5285.3;
cout << model() << endl;
```

4.8.2 Making Plots

Use the `Plot()` method to create and fill a ROOT histogram (classes `TH1F` and `TH2F`) with the values of a PDF sampled at the center of each bin (multiplied by the bin size). This method can create either one-dimensional histograms

```
TH1F *hist1d = PDF.Plot(x);
```

or two-dimensional histograms

```
TH2F *hist2d = PDF.Plot(x,y);
```

where `x` and `y` are dependent variables of the PDF. Both of these forms have some optional parameters which can be added after the mandatory parameters: a normalization factor to scale the PDF values by (default is one), and the number of bins to use (default is 100 for one dimension, and 40×40 for two dimensions). For example

```
TH1F *hist1= PDF.Plot(x,12.3);     // normalize to 12.3 events
TH2F *hist2= PDF.Plot(x,y,1.0,75); // use 75 bins
```

The histogram you create will have its ranges specified by the ranges of the dependent variables, and will have axis labels based on the variable titles and units. There is a shorthand for using the binning and normalization of an existing histogram

```
TH1F *model_hist= PDF.Plot(x,data_hist);
```

which is convenient for overlaying a PDF curve over a histogram of the data being fit (see the examples in Section A). Note that this probably does not do what you want unless `datahist` has the same range as `x`.

The `Plot()` method returns a pointer to a histogram object, rather than directly making a plot. You can use this pointer to set drawing options (line style, fill pattern, etc) and create a plot. For example, the following macro shows the differences

```
TH1F *histogram= PDF.Plot(x);
histogram->SetLineStyle(2);
histogram->Draw();
```

A shorthand for creating and plotting a histogram without changing the default options is

```
PDF.Plot(x,y)->Draw();
```

Refer to the ROOT documentation of the `TH1` class for details on setting histogram options and the `Draw()` command.

In the special case of a model which, at the top level, is a sum of PDFs, you can plot either of the individual PDFs correctly scaled for their relative normalization using

```
RooAddPdf model("model","Signal+Background",signal,bg,f);
model.Plot(x)->Draw();          // plot the combined signal+bg
model.Plot(1,x)->Draw("same"); // superimpose the bg contribution
```

The new `Plot()` methods with an extra initial integer parameter are equivalent to the ones described above. The integer parameter selects which PDF contributing to the sum to plot: use zero for the first PDF and one for the second. Sub-histograms created this way will have their default line style set to dashed.

Other histograms you can create from a PDF are a one-dimensional likelihood scan and a projection. Use

```
TH1F *histogram= PDF.Scan(dataset,parameter);
```

to create a histogram filled with values of the negative log-likelihood evaluated for a specified data set at varying values of the specified parameter. Use

```
TH1F *histogram= PDF.Project(u,v);
```

to plot the projection

$$f(u, \vec{x}\,; \vec{p}) \equiv \int_{v(min)}^{v(max)} F(u, v, \vec{x}\,; \vec{p})\, dv$$

where u and v are variable objects. The Scan() takes an optional argument to specify the number of histogram bins to use. The Project() method takes the same optional normalization and number of bins arguments as the one-dimensional Plot() method (you can also call it with a reference histogram).

One caveat for C++ experts: you should generally not delete the histogram objects you use to make plots in a macro since they are still being used by the plot object. This does not lead to memory leaks since, when you rerun the macro, ROOT recognizes that you are recreating an existing histogram and deletes the original first.

4.8.3 Displaying Parameters

The Parameters() method creates a graphics box containing text with a PDF's parameter values and errors. This method takes three optional arguments and returns a pointer to a ROOT TPaveText object. The optional parameters are a label to display above the parameters, the number of significant digits to display, and an option string

```
TPaveText *box= PDF.Parameters("Fitted Parameter Values:", 2, "EC");
```

The label parameter is interpreted using the TLatex class and so can include some mathematical formatting (refer to the ROOT documentation for details). The number of significant digits defaults to two and is used to truncate the parameter values and errors. The possible options (combined into a single string, following standard ROOT practice) are E (requesting that errors be shown) and C (requesting that constant values be shown). The labels shown for each parameter are, by default, the same as the names used to create them. To set alternate labels, which will also be interpreted using the TLatex class, use the variable's SetLabel() method

```
bres.SetLabel("#sigma_{B}");
```

Refer to Section A for examples of displaying parameters.

The TPaveText pointer returned by this method can be used to change its display attributes (background color, text font, etc) and draw it on the current plot

```
box->SetBackgroundColor(7);
box->Draw();
```

As with histograms, you can use the shorthand

```
PDF.Parameters("",3)->Draw();
```

when you do not need to change any display defaults. Once you have displayed the box of parameter values, you can interactively reposition and resize it with the mouse.

4.8.4 Generating a Data Set

You can create a data set containing events sampled from a PDF's distribution using

```
RooDataSet *data= PDF.generate("A Toy MC Sample",1000);
```

The first parameter is a descriptive title for this sample, and the second parameter is the number of events to generate. An optional third parameter controls how verbose the generator is: the default produces output like this

```
generated 10% of the events
generated 20% of the events
generated 30% of the events
generated 40% of the events
generated 50% of the events
generated 60% of the events
generated 70% of the events
generated 80% of the events
generated 90% of the events
Real time 0:0:34, CP time 17.190
generated 1000000 events after 1339739 calls (eff = 74.6414%)
```

Set the third parameter to `kFALSE` to suppress this output. Note that the updates every 10% will only be printed if the generation is taking long enough. A data set created this way records the values of the PDF parameters it was generated with. Use the data set's `Print()` method to see these values. Note that the `generate()` method returns a pointer to a `RooDataSet` object, which might be zero in case the data set cannot be created.

Note that the generate method uses RooRandom to give reproducible random number sequences every time you re-enter a root session. For more serious toy Monte Carlo studies the RooMCStudy class is provided (see section **??** for more information).

5 Performing the Fit

Performing a fit consists of minimizing the negative log-likelihood of a PDF calculated over a data set

$$-\log \mathcal{L}(\vec{p}) = \sum_k F(\vec{x}_k\,;\vec{p})$$

with respect to the model's parameters. The RooFitTools package uses the MINUIT[7] algorithms[5] to find the minimum of this function and estimate the errors in each parameter.

To perform a fit of a data set `events` to PDF `model`, use

```
model.fitTo(events);
```

where `events` can either be a pointer or a reference to a `RooDataSet` object. In order to fix some parameters in the fit, either create them with the fixed form of the constructor

```
RooRealVar tauB("tauB","B Lifetime",1.63,"ps");
```

or else force them to be fixed using

```
tauB.setConstant();
```

By default, MINUIT will not restrict the range of the fit parameters, but you can change this behavior for individual variables using

```
tauB.setLimits();
```

The initial step size used for each free parameter is 10% of its range. The `fitTo()` method generates the usual MINUIT output[6] displayed on the standard output stream. To modify the default behavior of the `fitTo()` method, use an optional string which can include any of these characters (in either upper or lower case): Q,L,T or M. The 'Q' (for quiet) option suppresses the usual MINUIT output, the 'L' option writes a log file which records each iteration of the fit, and the 'T' option generates a timing summary for the fit. The 'M' option requests that the errors be calculated using only the MIGRAD strategy (by default, MIGRAD is followed by a MINOS). To use HESSE rather than MINOS or just MIGRAD to calculate the errors, include 'MH' in the option string. The `fitTo()` method returns an integer status code which is non-zero if the fit fails to converge normally.

In order to assess the goodness of a fit, you can use the `chiSquare()` method to calculate the χ^2 of a PDF with respect to a histogram using the current values of the PDF's parameters (which may or may not be fitted)

[5] ROOT includes a C++ translation of the CERNLIB fortran MINUIT routines.
[6] Refer to the Reference [7] for details.

```
PDF.chiSquare(var,histogram);
```

where `histogram` is a pointer to a ROOT `TH1F` histogram object (see Section 4.8.2) and `var` is the name of the PDF variable being histogrammed. Alternatively, you can compare the minimum value of the negative log-likelihood function with the expected distribution of values for samples generated according to the fit model (this distribution is one of the standard plots generated using the Monte Carlo study methods described in the next Section.)

6 Running Monte Carlo Studies

A "toy Monte Carlo" sample is a set of random events generated according to a known PDF. These samples are useful for several reasons:

1. generating and fitting a sample using the same PDF tests the technical correctness of the generator and fit engine,
2. generating and fitting many statistically independent samples using the same PDF provides a robust estimate of the distribution of fit parameter errors and their correlations,
3. a comparison of the results from generating and fitting a sample using different PDFs can be used to study the systematic errors due to incorrect assumptions about the fit model or the values of fixed parameters, and
4. the distribution of maximum likelihoods obtained from a toy Monte Carlo study can be used to estimate the goodness of fit of a real sample under the assumption that the fit model correctly describes the sample.

Section 4.8.4 describes how to generate a single data set from a PDF. This section describes methods to automatically perform studies based on many independently generated data sets.

In general, a toy Monte Carlo study uses two different PDFs: one to generate samples of events, and another to fit them to. The `RooMCStudy` class controls the generation and fitting of samples. Create a study object by providing a name and title:

```
RooMCStudy study("study","Systematics Study");
```

You then specify the models to use for generating and fitting like this:

```
study.setGenModel(PDF1);
study.setFitModel(PDF2);
```

although they may also be set to the same PDF (to test the correctness of the fit, for example). The arguments used in the constructor are a short name and a descriptive title for the study. The PDFs are not specified in the constructor so that the same model may be used to generate and fit, but using different parameters

```
tauB= 1.63;
study.setGenModel(PDF);
tauB= 1.68;
study.setFitModel(PDF);
```

Use this approach to study systematic errors due to incorrect assumptions about the values of fixed parameters in the fit.

To perform a study use the `run()` method, specifying the number of independent samples to generate and fit and the number of events in each sample

```
study.run(1000,100); // 1000 independent 100 event samples
```

This method creates the following histograms to record the results for each sample

- the distribution of fit status codes (zero indicates successful convergence),
- the distribution of the minimum value of the negative log-likelihood,
- the distribution of each floating fit parameter's fitted value,
- the distribution of each floating fit parameter's fitted pull, and
- the two-dimensional correlation plot of each pair of floating fit parameters fitted values.

After creating a study using the run command, you can can fit the plots in the study by using the `fit()` method. as in: `study.fit()`

You may also set the options for the `fit()` function by using the `setFitOptions()` method which takes the same arguments as described for the `fitTo()` function at the end of Section 5.

Only the first histogram is filled for samples for which the fit does not converge. To save the results of a study to a file, use

```
study.save("study.root");
```

where, by convention, the file extension is `.root`. The file you create this way can be graphically browsed using the ROOT `TBrowser` class. To create a new browser, use

```
TBrowser browser;
```

and then use the browser's 'Open' menu item to open the file and display an icon for each of the histograms it contains. Click on a histogram to display it. Section A.4 shows an example of a toy Monte Carlo study and some of the resulting plots.

Once you have saved the file, you can also display the plots by loading it, and issuing individual commands for each histogram, e.g. below we show the pull distribution for "variableName":

```
TFile *f= new TFile("study.root");
TH1F* h;
 h= (TH1F*)f->Get("pulls:variableName");
 h->Draw();
```

A Examples

The code and data for these examples are available in the `RooFitTools/examples/` directory. To try the examples, copy the data files (`*.dat`) into your `workdir` directory and use the path `RELEASE/RooFitTools/examples/` to access the macros from a ROOT session started in `workdir`. The examples follow the pattern of defining variables, creating a fit model, reading or generating data, fitting, and then plotting the results. A large part of these macros is concerned with generating a nice plot after the fit is done. This part of the macros is not shown here, but is included in the `examples` directory.

A.1 B Mass

This example demonstrates a fit to the combined data from exclusively reconstructed B^0 and $B^{\pm}$ decays. The data consists of reconstructed mass values and includes about 50% background after a cut on ΔE. Refer to Section 10 and Figure 26 of Reference [8] for details. Figure 6 shows the output generated by running the following macro (named `bmass.cc` in the `examples` directory)

```
bmass() {
  RooRealVar // define the variables
    mreco("mreco","Beam-Energy Substituted B Mass", 5200.,5290.,"MeV"),
    endpt("endpt","Half of Y(4S) Mass", 5290.,"MeV"),
    bmass("bmass","B0/B+ Average Mass", 5279., 5265., 5285.,"MeV"),
    bresn("bresn","B0/B+ Mass Resolution", 3.0, 0.0, 10.0, "MeV"),
        c("c","Background Shape Parameter", -50.0, -100.0, 100.0),
        f("f","Signal Fraction", 0.5, 0.0, 1.0);

  // create the fit model
  RooArgusBG bg("bg","ARGUS Background Shape",mreco,endpt,c);
```

```
  RooGaussian signal("signal","Signal Gaussian",mreco,bmass,bresn);
  RooAddPdf model("model","Signal+Background Distribution",signal,bg,f);

  // read the data and fit it
  RooDataSet *data= RooDataSet::read("bmass.dat", mreco);
  model.fitTo(data);

  // ...code to create the plot omitted here...
}
```

In this fit, the endpoint mass of the Argus shape model (see Section 4.1.7) is fixed at half of the nominal $\Upsilon(4S)$ mass.

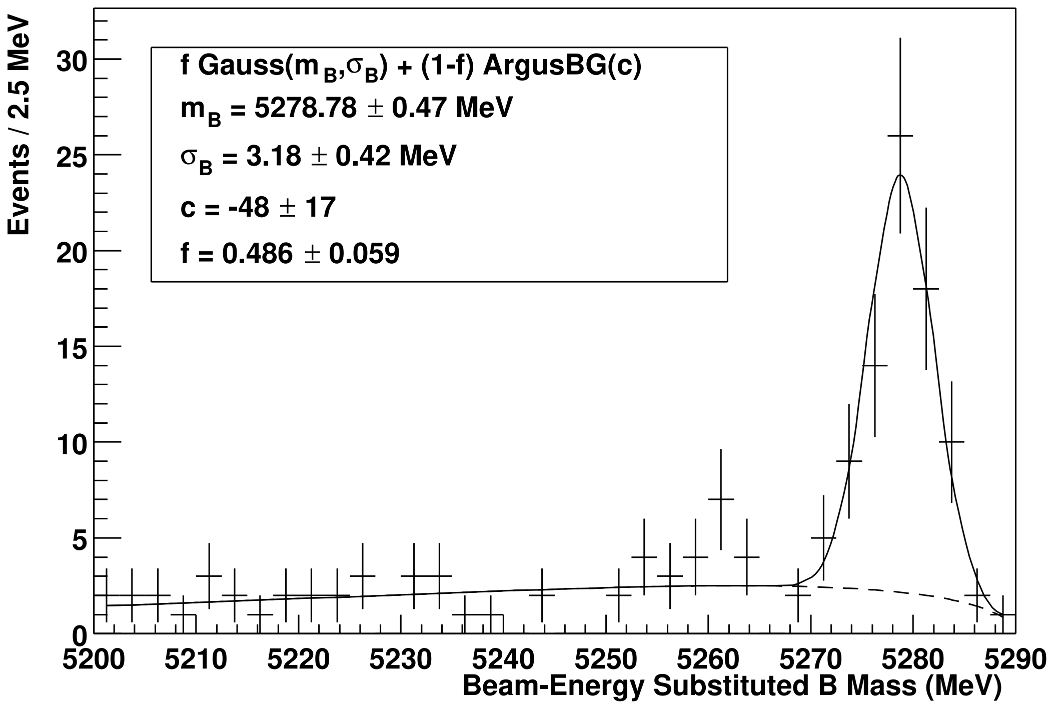


Figure 6: A fit to the beam-constrained $B^0/B^{\pm}$ mass distribution described in reference [8]. The fit model consists of the sum of the Argus background shape and a Gaussian. Refer to Section A.1 for the macro which generated this plot.

A.2 Charmonium Radiative Decays

This examples demonstrates a fit to data from inclusively selected $J/\psi \to e^+e^-(\gamma)$ decays[9]. The fit is to $m(e^+e^-)$ and so has a tail on the low side due to photon bremsstrahlung. The result of the fit is shown in Figure 7. The macro that was used is (`eetail.cc` in the `examples` directory)

```
eetail() {
  RooRealVar // define the variables
      mee("mee","Electron Pair Invariant Mass",2500.,3500.,"MeV"),
    mjpsi("mjpsi","J/Psi Mass",3100.,3000.,3200.,"MeV"),
    rjpsi("rjpsi","J/Psi Mass Resolution",12.,0.,25.,"MeV"),
    taila("taila","Tail Transition",0.7,0.1,1.5),
    tailn("tailn","Tail Power",1.0),
    bgtau("bgtau","Background Shape Parameter",555.,1.,1000.,"MeV"),
        f("f","Signal Fraction",0.3,0.0,1.0);

  // create the fit model
  RooCBShape cbs("cbs","Signal Lineshape",mee,mjpsi,rjpsi,taila,tailn);
  RooLifetime bg("bg","Background Shape",mee,bgtau);
  RooAddPdf model("model","Signal + Background Shape",cbs,bg,f);

  // read the data and fit it
  RooDataSet *data= RooDataSet::read("eetail.dat",mee);
  taila.setLimits();
  model.fitTo(data);

  // ...code to create the plot omitted here...
}
```

In this example, the tail power parameter (n in Section 4.1.9) is fixed at one. The χ^2 for this fit is 107 for 95 degrees of freedom, but similar values are obtained with different values of the tail shape parameters reflecting the fact that the background under the tail is poorly constrained.

A.3 B Lifetime

This example demonstrates a fit to a toy Monte Carlo sample of 1000 background-free B decays. The fit uses per-event errors on the B proper decay time, with the distribution shown in Figure 8. The corresponding distribution of reconstructed proper decay times is also shown in Figure 8. The fit is performed using two different models, both with the B lifetime and the global event-error scale factor (see Section 4.1.3) as free parameters. The first model includes the correct distribution of per-event errors and the second model

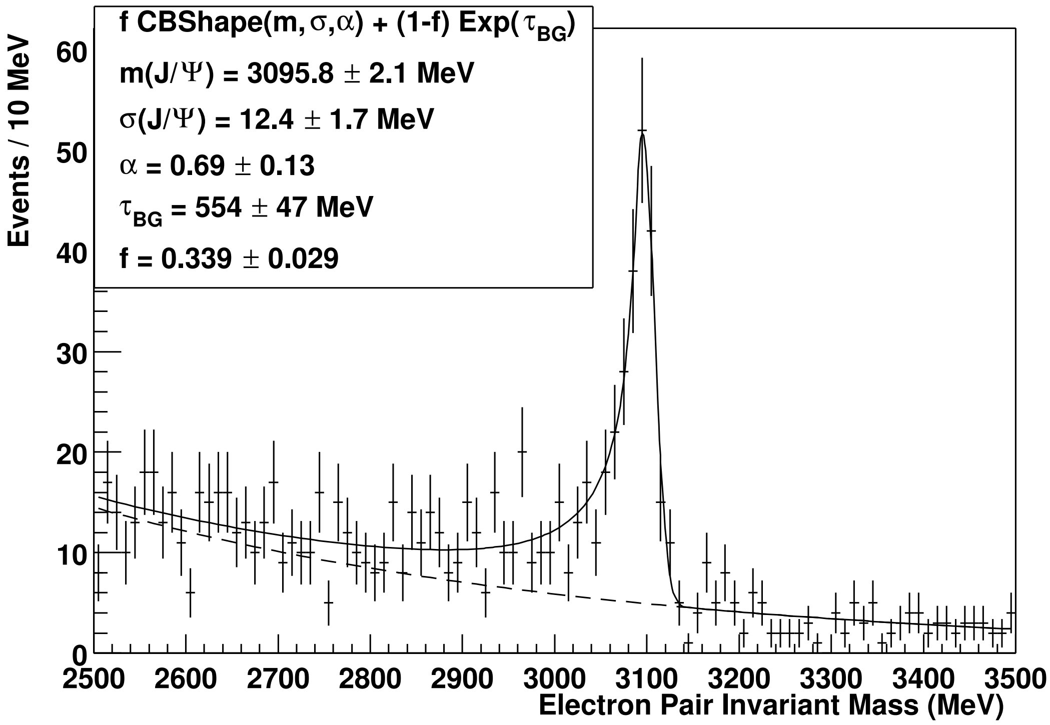


Figure 7: A fit to the e^+e^- invariant mass in events with a $J/\psi \rightarrow e^+e^-(\gamma)$ candidate. The fit model consists of the sum an exponential (for the background) and the Crystal Ball lineshape (for the signal peak and tail). Refer to Section A.2 for the macro which generated this plot.

assumes a flat distribution of these errors. Figure 8 compares the projections of these two models onto the event error and reconstructed proper decay time axes with the generated data. The macro for this example is (see `blife.cc` in the `examples` directory)

```
blife() {
  RooRealVar // define the variables
     terr("terr", "Decay Time Reconstruction Error",0.2,1.2,"ps"),
     ttru("ttru", "True Proper Decay Time",0.0,7.5,"ps"),
     trec("trec", "Reconstructed Proper Decay Time",-0.05,0.10,"ps"),
       dt("dt",   "Decay Time Resolution Smearing",-5.0,5.0,"ps"),
     taub("taub", "B0 Lifetime",1.56,1.5,1.6,"ps"),
     bias("bias", "Global Decay Time Bias",0.0, "ps"),
    scale("scale","Global Error Scale Correction",1.0,0.5,1.5),
    mperr("mperr","Most Probable Decay Time Error",0.5,"ps"),
```

```
    errms("errms","RMS Decay Time Error",0.03,"ps"),
    errp1("errp1","Error Distribution Shape Parameter 1",-1.0),
    errp2("errp2","Error Distribution Shape Parameter 2",2.0);

  // create the true distribution of proper decay times
  RooLifetime truth("truth","True Decay Time Distribution",ttru,taub);
  // create the distribution of per-event decay time errors
  RooCBShape errors("errors","Decay Time Error Distribution",
                    terr,mperr,errms,errp1,errp2);
  // convolute the true distribution with the per-event errors
  RooGaussEvt realSmear("realSmear","Real Decay Time Error Smearing",
                        dt,terr,bias,scale,errors);
  RooConvPdf realModel("realModel","Real Decay Time Distribution",
                       truth, realSmear, dt, ttru, trec);

  // create some events sampled from this model and fit it
  RooDataSet *data= realModel.generate("B Lifetime Toy MC",1000);
  realModel.fitTo(data);

  // create a model with a uniform distribution of per-event errors
  RooGaussEvt flatSmear("flatSmear","Flat Decay Time Error Smearing",
                        dt,terr,bias,scale);
  RooConvPdf flatModel("flatModel","Flat Decay Time Distribution",
                       truth, flatSmear, dt, ttru, trec);
  // reset the initial parameter values and fit to this model
  taub= 1.56;
  scale= 1.0;
  flatModel.fitTo(data);

  // ...code to create the plots omitted here...
}
```

The two fits using different distributions of errors give identical results (within the numerical tolerances), as expected since neither model has free parameters which determine the distribution of errors. The fact that the actual distribution of errors is not flat results in a small probability factor which translates into a constant positive offset of the negative log-likelihood curve, but otherwise does not affect the minimization and error estimates. The fitted value of the B lifetime is statistically consistent with the input value used to generate the data sample, and the fitted scale factor is consistent with one.

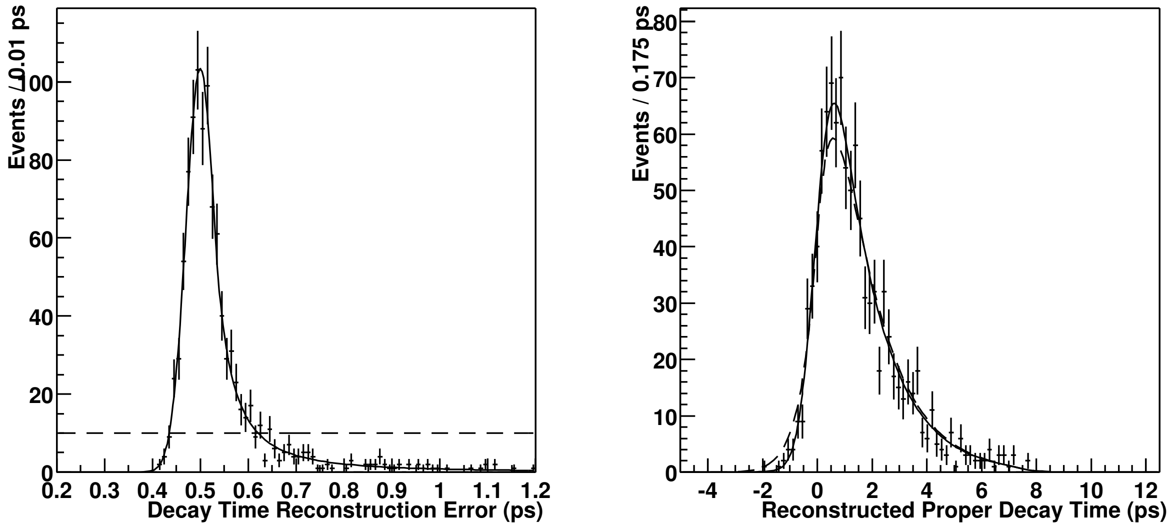


Figure 8: A fit to a toy Monte Carlo sample of background-free B decays. The free parameters in the fit are the B lifetime and a global event-error scale factor (see Section 4.1.3). The data points are the distributions of per-event errors on the reconstructed decay time (left-hand plot) and the reconstructed decay time (right-hand plot). The curves are the projections of two different PDF models onto these axes. The solid curves are for a model using the correct distribution of per-event errors, and the dashed curves are for a model assuming a flat distribution of these errors. The projections of the second model (dashed curves) are systematically different from the data distributions, but this does not bias the lifetime estimate.

A.4 CP Fitting Study

This examples demonstrates a toy Monte Carlo study of a fit for the CP violation parameter $\sin(2\beta)$. The data set consists of values for the reconstructed decay time difference Δt, the B^0 candidate's beam-substituted mass, and a per-event tagging probability which is assumed to be linearly distributed from -1 ($\overline{B^0}$) to $+1$ (B^0). Each sample contains 200 events, of which 100 are signal events and the rest are background. The signal is generated with a Gaussian distribution in the beam-substituted mass, and with perfect Δt resolution (adding global or per-event errors is straightforward but significantly increases the CPU time required for this study). The background is distributed according to the Argus shape in the beam-substituted mass (see Section 4.1.7), is Gaussian in Δt, and is flat in the tagging variable. The distribution of the generated events in the beam-substituted mass is essentially the same as shown in Figure 6.

The study consists of fits to 10000 independent samples generated and fit with the same model. Each fit has two free parameters: $\sin(2\beta)$ and the signal fraction. The macro to perform this study is (see `cpstudy.cc` in the `examples` directory)

```
cpstudy() {
  RooRealVar // define the variables
       dt("dt","Decay Time Difference (reco-tag)",-7.5,7.5,"ps"),
        x("x","Tagging Variable (+1=B0,-1=B0bar)",-1.,1.),
    mreco("mreco","Beam-Energy Substituted B Mass", 5200.,5290.,"MeV"),
      tau("tau","Neutral B Average Lifetime",1.56,"ps"),
       dm("dm","Neutral B Eigenstate Mass Difference",0.464,"/ps"),
        a("a","Magnitude of lambda(CP)",1.0),
        b("b","sin(2#beta)",0.0,-1.,1.),
     bias("bias","Background Delta(t) Bias",0.0,"ps"),
    bgrms("bgrms","Background Delta(t) RMS",1.0,"ps"),
    endpt("endpt","Half of Y(4S) Mass", 5290.,"MeV"),
    bmass("bmass","B0 Mass",5279.,"MeV"),
    bresn("bresn","B0 Mass Resolution",3.0,"MeV"),
        c("c","Background Shape Parameter",-50.0),
     sigf("sigf","f_{sig}",0.5,0.3,0.7);

  // create the signal (dt,x) distribution with linear tagging PDFs
  RooCPMixLinear sigdt("sigdt","Signal Delta(t) Distribution",
                       dt,x,tau,dm,a,b);
  // create the signal (mreco) distribution
  RooGaussian sigm("sigm","Signal m(reco) Distribution",
                   mreco,bmass,bresn);
  // multiply these together for the signal (dt,x,mreco) distribution
  RooProdPdf signal("signal","Signal Distribution",sigdt,sigm);

  // create the background (dt) distribution
  RooGaussian bgdt("bgdt","Background Delta(t) Distribution",
                   dt,bias,bgrms);
  // create the background (mreco) distribution
  RooArgusBG bgm("bgm","Background m(reco) Shape",mreco,endpt,c);
  // multiply these together for the bg (dt,x,mreco) distribution
  RooProdPdf bg("bg","Background Distribution",bgdt,bgm);

  // add together the signal and background contributions
  RooAddPdf combined("combined","Signal+Background Distribution",
                     signal,bg,sigf);

  // fix sin(2 beta) to be within +/-1
```

```
  b.setLimits();

  // initialize and run a toy Monte Carlo study and save the results
  RooMCStudy study("study","Sin(2*beta) Study");
  study.setGenModel(combined);
  study.setFitModel(combined);
  study.run(10000,200);
  study.save("study.root");
}
```

Figure 9 shows some of the histograms which are created by this macro (these plots were made using `examples/cpplot.cc`). This study demonstrates that the parameter values and errors are correctly calculated since the pull distributions are unit Gaussians centered on zero, and that the two parameters are essentially uncorrelated. The distribution of negative log-likelihood values can be used to estimate the goodness of fit for a particular set of data. The expected statistical errors for a sample of 200 events which are correctly described by this model are ± 0.32 for $\sin(2\beta)$ and $\pm 4.2\%$ for the signal fraction. All of the fits converged successfully in this example.

A.5 D^{*+} Mass Difference Fit

The example `dstar.cc` shows how to fit the distribution of the mass difference between the reconstructed D^{*+} and D^0 state in the decay $\mathrm{D}^{*+} \to \mathrm{D}^0\pi^+$ followed by $\mathrm{D}^0 \to \mathrm{K}^-\pi^+$. The signal is fitted with a single Gaussian and the background with the `RooDstarBG` PDF described in Section 4.1.10. It is also an illustration of how to use blind variables as described in Section 2.3.

In the `examples` directory the files `deltam.cc` and `deltam.dat` are used for the fit shown in Figure 10.

```
deltam() {
  // Fit model parameters for mass difference
  RooRealVar deltaM("deltaM","Reconstructed Mass Difference",
                    0.13957,0.15857,"GeV");
  RooRealVar piPlusMass("piPlusMass","The pi0 mass",0.13957,"GeV");
  RooRealVar c("c","Background Shape Parameter", 0.002, 0.0, 0.01);
  RooRealVar f("f", "Signal Fraction",0.4, 0.0, 1.0);

  // Create blind variables
  RooBlindGaussian blinderMass("blinderMass",
                               "This is a blinding string",0.00003);
  RooBlindVar rDeltaM("rdeltaM","Mass Difference",&blinderMass,
                     0.1454,0.13957,0.15457 "GeV");
```

```
  RooBlindGaussian blinderWidth("blinderWidth",
                                "This is a another string",0.00003);
  RooBlindVar widthDeltaM("widthDeltaM","Mass Difference Width",
                          &blinderWidth,
                          0.0003, 0.0001, 0.01, "GeV");

  // create the mass diff fit model components
  RooGaussian signalDeltaM("signalDeltaM","Signal Distribution",
                           deltaM,rDeltaM,widthDeltaM);
  RooDstarBG bgDeltaM("bgDeltaM","D* Background",deltaM,piPlusMass,c);
  RooAddPdf modelDeltaM("modelDeltaM","Signal + Background",
                        signalDeltaM,bgDeltaM,f);

  // Read a dataset
  RooDataSet* data = RooDataSet::read("deltam.dat",deltaM);

  // If fitting MC the blinding should be turned off before the fit.
  //  widthDeltaM.setBlind(kFALSE);
  //  rDeltaM.setBlind(kFALSE);

  // Fit the data
  c.setLimits();
  modelDeltaM.fitTo(data,"M");

  // ...code to create the plot omitted here...
}
```

A.6 A Non-Parametric Model

The example `nonparam.cc` demonstrates the use of the KEYS[5] non-parametric PDF (see Section 4.1.12). This example first constructs a complex shape from a sum of Gaussians (which happen be centered at the masses of some charmonium states). Next, we generate 1000 events sampled from this distribution and create a non-parametric model of these events. Note that the non-parameteric model actually appears to describe the data better than the true underlying model, because it can adapt to include features arising from statistical fluctuations. The full code for this example is in the `examples` subdirectory and generates the plot shown in Figure 11. Here is an excerpt demonstrating the use of the KEYS non-parametric PDF:

```
nonparam() {
  RooRealVar // define the variables
```

```
    mass("mass","Reconstructed Candidate Mass",3.0,3.8,"GeV");

  // ...code to create 5 Gaussians and define their relative weights...

  // create a PDF as a sum of the 5 Gaussians
  RooAddPdf all("all","All States Combined",g1,g2,g3,g4,g5,f1,f2,f3,f4);

  // generate and plot some data from this PDF
  RooDataSet *data= all.generate("data",1000);
  TH1F *dataplot= data->Plot(mass,"",40);
  dataplot->Draw("E");
  TH1F *exact= all.Plot(mass,dataplot);
  exact->Draw("csame");

  // make a non-parametric PDF to describe this data and plot it
  RooKeysPdf keys("keys","Non-Parametric Model",mass,*data);
  TH1F *approx= keys.Plot(mass,dataplot);
  approx->SetLineStyle(2);
  approx->Draw("csame");
}
```

B Adding New Models

This section explains how to add a new PDF building block to the RooFitTools package. First, choose a short but descriptive name such as "PowerLaw" (use capitalization to delimit words instead of spaces). You now need to create two new files in the `RooFitTools` directory of your test release, for example `RooPowerLaw.rdl` and `RooPowerLaw.cc` (using the common prefix "Roo"). You can do this most easily by copying the existing `RooPdfExample` files and replacing all occurrences of "PdfExample" with your chosen name. You will also need to replace `ROO_PDF_EXAMPLE` near the top of the `rdl` file with a similarly formatted name for your PDF. An `rdl` file is essentially the same as a C++ header file, but has a different extension to flag it for special processing so it can be used in an interactive ROOT session.

Edit the `rdl` file to declare your PDF's dependent variables and parameters with changes to the constructor arguments and public data members. Make a corresponding change to the constructor arguments in the `cc` file and edit the calls to the `addDependent()` and `addParameter()` methods (one for each public data member, in the same order as they are listed in the `rdl` file). Edit the `&x` and `&c` addresses in the `RooPdf` constructor to refer to the first dependent variable and parameter, respectively, listed in your public data members.

Next, edit the `evaluate()` method in your `cc` file to calculate your probability density in terms of the current values of the public data members which correspond to your dependent variables and parameters. If your formula is not normalized, then simply scale the result by the variable `_norm`, as shown in the example. Your formula must at least be finite for all possible values of the parameters. If you need to make some restrictions on the allowed ranges of some parameters, you will need to edit the `useParameters()` method in your `cc` file (refer to `RooLifetime.cc` for an example.) If you know an analytic expression for your PDF's normalization, use it instead of the numerical integration in the `useParameters()` method (you must do this when your PDF depends on more than one variable).

At this point, you have created a working PDF which can be used like any other predefined PDF (see Section 4.8). Simply update your shared library and restart ROOT, as described in Sections 1.1 and 1.2. Once you have successfully added your PDF, edit and run the `examples/check.cc` macro to perform some basic checks that it is working correctly.

To improve the performance of your PDF for fitting, provide an analytic expression for the normalization in `useParameters()` and use temporary variables to avoid duplicating calculations in `evaluate()`. To improve the performance of your PDF for toy Monte Carlo studies you can also implement the `initGenerator()` and `generateDependents()` methods. If you do not implement these methods, then the base class code will generate samples from your PDF using an accept/reject algorithm and a brute-force numerical estimate of your PDF's maximum value. See `RooArgusBG.cc` for an example of implementing only the `initGenerator()` method to provide an analytic calculation of the maximum value. Refer to `RooGaussian.cc` for an example where both of these methods are implemented.

C Technical Details of the "GExp" Resolution Model

To evaluate

$$f_{\pm}^{\mathrm{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \frac{1}{2\tau}\int_0^{\infty} ds\,\exp\left(-\frac{s}{\tau}\right)\cdot\exp\left(\pm i\omega s\right)\cdot\int_{-\infty}^{0} du\,\frac{1}{\tau_r}\exp\left(\frac{u}{\tau_r}\right)\cdot h_G(t\mp s-u;\delta=0,\sigma)$$

we replace the integration variable u by $T = u \pm s$ and change the order of the two integrations. This procedure leads to

$$
\begin{aligned}
&f_+^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \\
&\qquad \frac{1}{2\tau}\int_{-\infty}^{0} dT\,\frac{1}{\tau_r}\cdot h_G(t-T;\delta=0,\sigma)\cdot\exp\left(\frac{T}{\tau_r}\right)\int_0^{\infty} ds\,\exp\left\{\left(-\frac{1}{\tau}+i\omega-\frac{1}{\tau_r}\right)s\right\} \\
&\quad + \;\frac{1}{2\tau}\int_{0}^{\infty} dT\,\frac{1}{\tau_r}\cdot h_G(t-T;\delta=0,\sigma)\cdot\exp\left(\frac{T}{\tau_r}\right)\int_T^{\infty} ds\,\exp\left\{\left(-\frac{1}{\tau}+i\omega-\frac{1}{\tau_r}\right)s\right\}
\end{aligned}
$$

and

$$
\begin{aligned}
&f_-^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \\
&\quad \frac{1}{2\tau}\int_{-\infty}^{0} dT\,\frac{1}{\tau_r}\cdot h_G(t-T;\delta=0,\sigma)\cdot\exp\left(\frac{T}{\tau_r}\right)\int_0^{-T} ds\,\exp\left\{\left(-\frac{1}{\tau}-i\omega+\frac{1}{\tau_r}\right)s\right\}.
\end{aligned}
$$

The three integrals over s are straightforward to calculate and we obtain

$$
\begin{aligned}
f_+^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \frac{1}{\frac{1}{\tau_r}+\frac{1}{\tau}-i\omega}\cdot\frac{1}{2\tau\tau_r}\cdot\Bigg[&\int_0^{\infty} dT\,h_G(t-T;\delta=0,\sigma)\cdot\exp\left\{\left(-\frac{1}{\tau}+i\omega\right)T\right\} \\
&+\int_0^{\infty} dT\,h_G(t+T;\delta=0,\sigma)\cdot\exp\left(-\frac{T}{\tau_r}\right)\Bigg]
\end{aligned}
$$

and

$$
\begin{aligned}
f_-^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \frac{1}{\frac{1}{\tau_r}-\frac{1}{\tau}-i\omega}\cdot\frac{1}{2\tau\tau_r}\cdot\Bigg[&\int_0^{\infty} dT\,h_G(t+T;\delta=0,\sigma)\cdot\exp\left\{\left(-\frac{1}{\tau}-i\omega\right)T\right\} \\
&-\int_0^{\infty} dT\,h_G(t+T;\delta=0,\sigma)\cdot\exp\left(-\frac{T}{\tau_r}\right)\Bigg]
\end{aligned}
$$

which can be expressed in terms of $f_\pm^{(k)}(t\,;\tau,\omega,\delta,\sigma)$:

$$
f_\pm^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r) = \frac{1}{\frac{1}{\tau_r}\pm\frac{1}{\tau}-i\omega}\cdot\left[\frac{1}{\tau_r}f_\pm^{(k)}(t\,;\tau,\omega,\delta=0,\sigma)\pm\frac{1}{\tau}f_-^{(k)}(t\,;\tau_r,0,0,\sigma)\right].
$$

For $\omega = 0$ and $\tau_r \simeq \tau$, $f_-^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r)$ features a mathematical pathology [10]: both the denominator of the fraction in front of the square brackets in the previous expression, and the difference inside the square brackets tend to zero. For $\left|\frac{\tau-\tau_r}{\tau}\right| < \frac{1}{260}$ we replace τ and τ_r with $\tau_m = \frac{\tau+\tau_r}{2}$ and $\epsilon = \frac{\tau-\tau_r}{2}$, and perform a Taylor expansion of $f_-^{\text{Exp}}(t\,;\tau,\omega,\sigma,\tau_r)$

in $\frac{\epsilon}{\tau_m}$. The contributions to $f_-^{\rm Exp}$ in odd powers of $\frac{\epsilon}{\tau_m}$ are zero by construction. The leading order contribution to $f_-^{\rm Exp}$ is

$$\frac{1}{2\sqrt{2\pi}\tau_m^2}\cdot\exp\left(\frac{t}{\tau_m}+\frac{\sigma^2}{2\tau_m^2}\right)\cdot$$
$$\left[\sigma\cdot\exp\left(-\frac{1}{2\sigma^2}\cdot\left(\frac{\sigma^2}{\tau_m}+t\right)^2\right)-\sqrt{\frac{\pi}{2}}\cdot\left(\frac{\sigma^2}{\tau_m}+t\right)\mathrm{erfc}\left(\frac{\sigma}{\sqrt{2}\tau_{\rm m}}+\frac{\rm t}{\sqrt{2}\sigma}\right)\right]$$

and the next-to-leading-order correction is

$$\frac{\epsilon^2}{\tau_m^2}\cdot\exp\left(\frac{t}{\tau_m}+\frac{\sigma^2}{2\tau_m^2}\right)\cdot\frac{1}{4\tau_m}\cdot$$
$$\left\{\exp\left(-a(t)^2\right)\cdot\left[\frac{\sqrt{2}\sigma}{\sqrt{\pi}\tau_m}-\frac{2\sigma^2\cdot a(t)}{\sqrt{\pi}\tau_m^2}+\frac{1}{3\sqrt{\pi}}\cdot\left(4a(t)^2-2\right)\cdot\left(\frac{\sigma}{\sqrt{2}\tau_m}\right)^3\right.\right.$$
$$+\frac{2}{\sqrt{\pi}}\cdot\left(\frac{\sigma^2}{\tau_m^2}+\frac{t}{\tau_m}\right)\cdot\left(\frac{\sigma}{\sqrt{2}\tau_m}-\frac{a(t)\cdot\sigma^2}{2\tau_m^2}\right)$$
$$\left.+\frac{2\sigma}{\sqrt{2\pi}\tau_m}\cdot\left(\frac{3\sigma^2}{2\tau_m^2}+\frac{t}{\tau_m}+\frac{1}{2}\cdot\left(\frac{\sigma^2}{\tau_m^2}+\frac{t}{\tau_m}\right)^2\right)\right]$$
$$\left.-\mathrm{erfc}\left(a(t)\right)\cdot\left[\frac{2\sigma^2}{\tau_m^2}+\frac{t}{\tau_m}+\left(\frac{\sigma^2}{\tau_m^2}+\frac{t}{\tau_m}\right)\cdot\left(3\cdot\frac{\sigma^2}{2\tau_m^2}+\frac{t}{\tau_m}\right)+\frac{1}{6}\cdot\left(\frac{\sigma^2}{\tau_m^2}+\frac{t}{\tau_m}\right)^3\right]\right\}$$

with

$$a(t)=\frac{\sigma}{\sqrt{2}\tau_m}+\frac{t}{\sqrt{2}\sigma}\;.$$

References

[1] H. Albrecht, et al (ARGUS Collaboration), “Reconstruction of B Mesons”, Phys. Lett. B185 (1987) 218.

[2] D.H. Perkins, “Introduction to High Energy Physics”.

[3] T. Skwarnicki, “A Study of the Radiative Cascade Transitions Between the Upsilon-Prime and Upsilon Resonances,” DESY F31-86-02 (thesis, unpublished) (1986).

[4] N. L. Johnson, “Systems of Frequency Curves Generated by Methods of Translation”, Biometrika 36 (1949) 149.

[5] K. S. Cranmer, “Kernel Estimation for Parametrization of Discriminant Variable Distributions”, ALEPH 99–144, December 1999 (see also `http://www-wisconsin.cern.ch/~cranmer/keys.html`)

[6] R. Cahn, "Analytic Evaluation of Integrals with a Resolution Function that is the Convolution of a Gaussian with an Exponential", August 2000.

[7] F. James, "MINUIT: Function Minimization and Error Analysis", Reference Manual, Version 9.41, CERN Program Library Long Writeup D506, August 1998.

[8] Charmonium Analysis Working Group, "Exclusive B Reconstruction into Charmonium Final States: Status Report", BaBar Analysis Document 12, Version 1.

[9] Data provided by Tania McMahon, private communication.

[10] J. Chauveau *et al.*, "B Lifetime Measurement Using Exclusively Reconstructed Hadronic B Decays", BaBar Analysis Document 37, Version 5.

[11] C. Dallapicolla, "DIRC parameterization", talk given at BaBar Collaboration Meeting, December 2000: /BFROOT/www/Organization/CollabMtgs/ 2000/det-Dec2000/Tues1e/carlod.ps

[12] Colorado Group, "Maximum likelihood fit analyses of quasi-two-body charmless B decay modes", BaBar Analysis Document 117, Version 1.

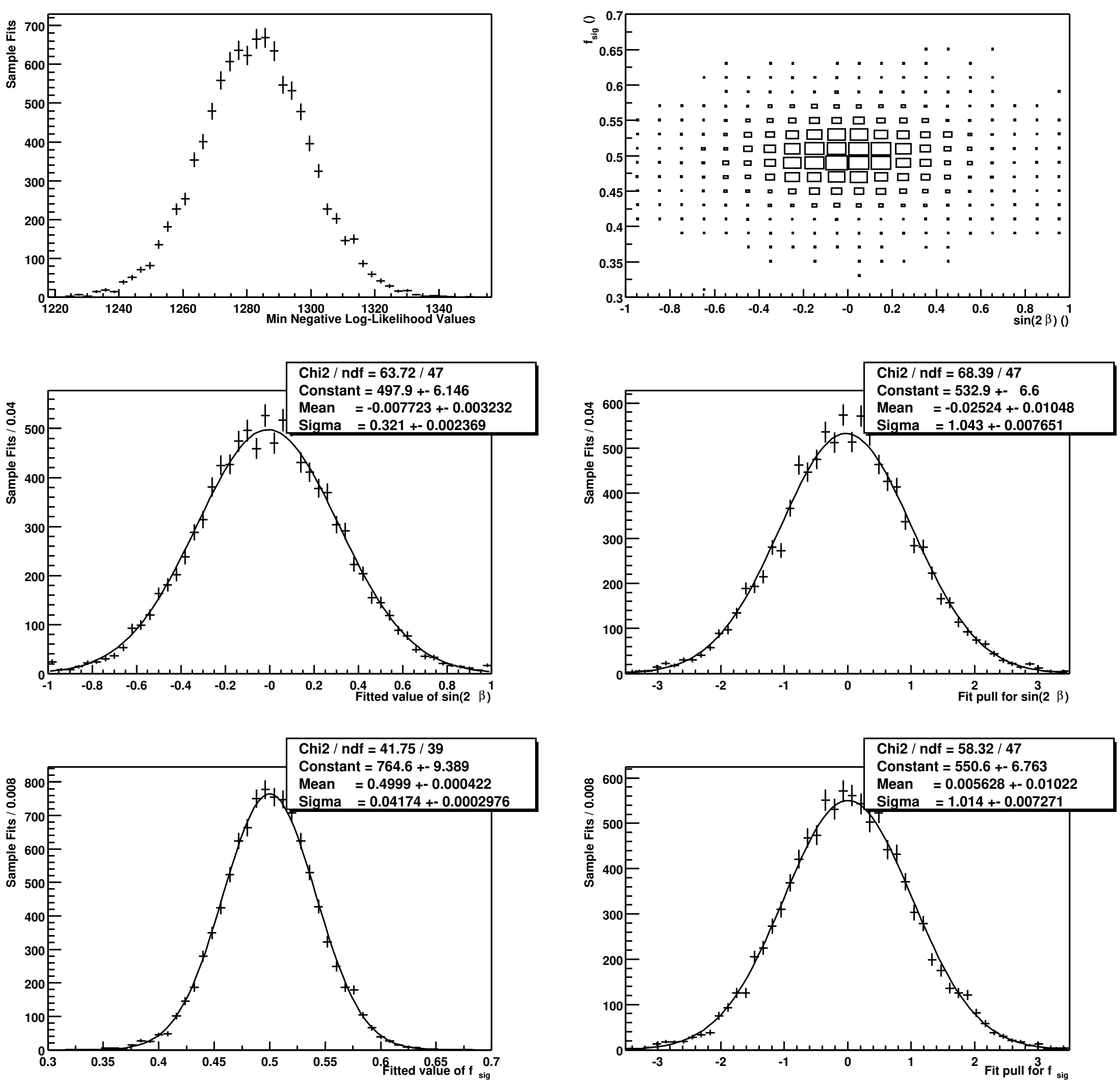


Figure 9: Results of a toy Monte Carlo study of a fit to the CP violation parameter $\sin(2\beta)$ and the signal fraction using 10000 independent samples of 200 events each. The top two plots are the distribution of minimum negative log-likelihood values (left side) and the correlation plot of fitted values for the two parameters. The lower two rows are the distributions of fitted parameter values (left side) and pulls (right side) for $\sin(2\beta)$ (middle row) and the signal fraction (bottom row).

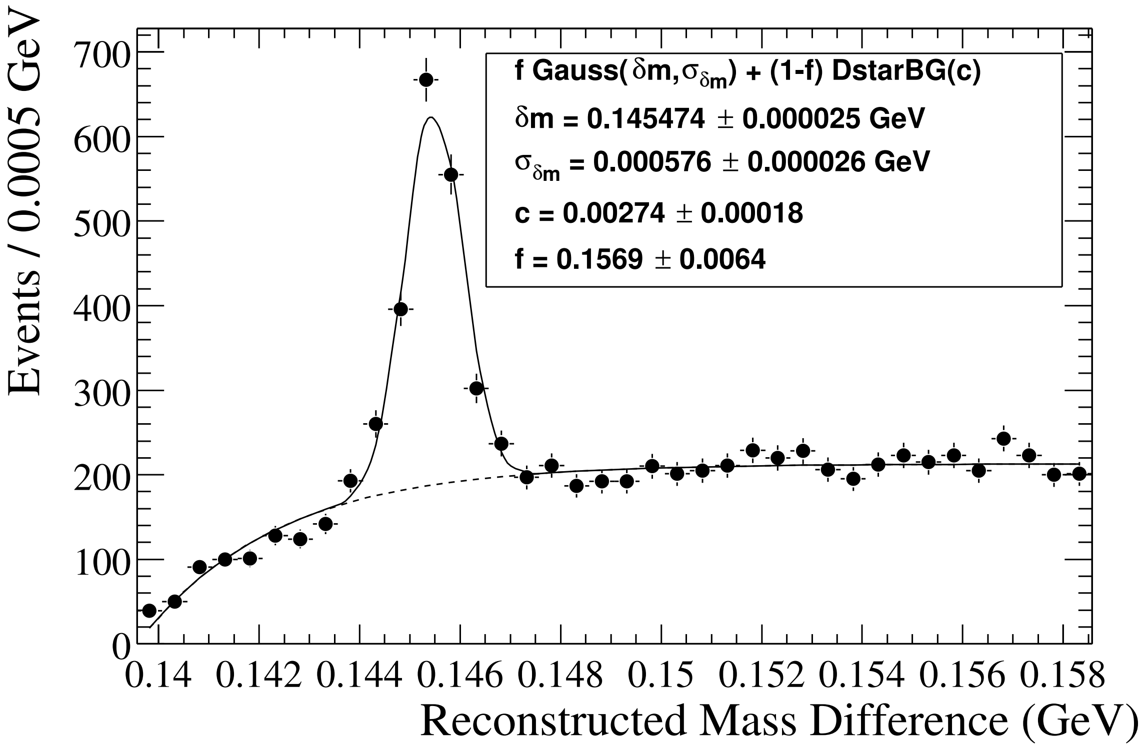


Figure 10: A fit to the mass difference between the reconstructed D^{*+} and D^0. The fit model is a sum of a single Gaussian and the `RooDstarBG` function. The values for δm and $\sigma_{\delta m}$ are blind.

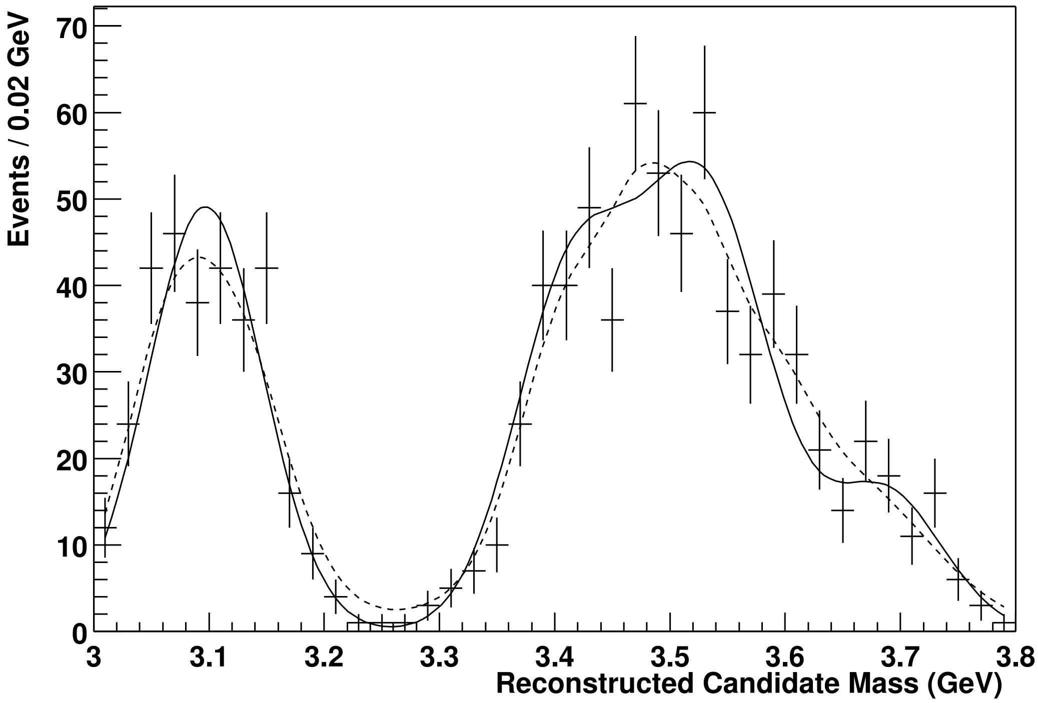


Figure 11: A comparison of the histogram of 1000 events (data points) with the underlying PDF used to generate these events (solid curve) and a non-parameteric KEYS model of these events (dashed curve).