AFit User Guide

Adrian Bevan[1], Fergus Wilson[2]
December 24, 2010

Abstract

AFit provides a high level user interface to RooFit and TMVA. Complicated maximum-likelihood fits can be set up using a text file (without the need to write a lot of code), and modified quickly to refine an analysis. There are a number of utilities to facilitate further analysis of the likelihood fit, such as toy MC, and plotting interfaces. The TMVAInterface provided allows a user to configure which classifiers to run with which variables. It is then possible to append RooDataSets with the output classifier MVAs for inclusion in a fit to the data. A number of examples are provded in the last section of this user guide.

Contents

[1]a.j.bevan@qmul.ac.uk, Queen Mary, University of London
[2]fergus.wilson@stfc.ac.uk, Rutherford Appleton Laboratory

1 Introduction to AFit

This package has been developed in order to add a layer of abstraction on top of RooFit [1] and simplify complex maximum likelihood fit based data analysis. The underlying function minimiser is MINUIT [2]. The aim of AFit is to provide a general fit framework that can be used in order to analyse data, but without having to write code in order to set the fit up. In order to do set up a general fit, you can configure AFit using a 'datacard'. In addition to this high level abstraction, it is also possible to use the individual wrapper classes to the RooFit Probability Density Functions (PDFs) when addressing simple problems. The default version of RooFit to use with AFit is the version bundled with ROOT [3].

AFit provides a higher level interface to RooFit that facilitates using a text configuration file to construct a complicated multidimensional likelihood fit model. There are also a number of tools and interfaces provided to simplify validation of the likelihood fit as well as performing analysis of the data. The rest of this document summarises the available PDFs (Section 2), different types of multi-dimensional component PDFs that can be constructed (Section 3), how to use the AFitMaster to construct a PDF model (Section 4), the various utilities available (Section 5), and working examples (Section 6). If you are already very familiar with the use of RooFit and maximum-likelihood fitting, you might consider starting to work through the examples section before reading all of the earlier sections of this user guide.

The main underlying statistical techniques used by RooFit and AFit are described in more detail in the following references [4, 5] as well as other books on statistics and data analysis.

1.1 Platform Requirements

The AFit package has been tested using

- gcc version 3.4.6 20060404
- The current beta version of AFit is being tested using ROOT version 5.26/00.
- Scientific Linux

2 PDFs

This section summarises the library of PDFs that are implemented in AFit. There are code snippets illustrating how to use the different PDFs described below. However for more complicated likelihood fit models, it is more practical to use the AFitMaster class as an interface to the PDF classes described here (See Section 4).

2.1 Parameter naming convention

The parameter names for all simple PDF components described in this section follow the same rule. The parameter name is constructed from two parts separated by a underscore ($_{)}$: (i) the PDF name, folllowed by (ii) the parameter variable name. For example, if we consider a Gaussian PDF with a PDF name "myGaussian", then the name of the mean of the Gaussian will be "myGaussian_mean". See Table 1 for a list of variable names for each PDF.

The following code shows how to make a Gaussian PDF and then change the value of the parameters.

```
RooRealVar x(``x'', ``'', -4.0, 4.0);

// The Gaussian PDF with default settings
AFitGaussian gausshape(x, ``mypdfname'');
// re-define default settings
gaussshape.setParameter(``mean'', 175.0);
gaussshape.setParameter(``width'', 3.0);

RooAbsPdf * gausPDF = gausshape.getPdf();
```

When building an AFitProdPdf the naming convention for the PDF follows the extended rule of (i) PDF component name followed by (ii) discriminating variable name, followed by (iii) parameter variable name. In this case, there will be an underscore between (i) and (ii). For example, the signal component for a one-dimensional fit in some variable x will have the PDF component name "signal". If this is described by a Gaussian distribution, then the mean of that distribution will have the parameter name "signal_xmean".

These conventions are imposed in order to help users configure their fits efficently.

2.2 Argus

This is the distribution used for a parametric description of a combinatorial background shape, as first used by the ARGUS Collaboration [6]:

$$\mathcal{P}(m\,; m_0, \xi) = \frac{1}{N} \cdot m\sqrt{1-(m/m_0)^2} \cdot \exp(\xi\,(1-(m/m_0)^2)) \cdot \theta(m < m_0) \qquad (1)$$

where $\theta(m < m_0) = 1$ and $\theta(m > m_0) = 0$. This PDF has a type 'argus' when using the AFitPdfFactory (see below). The following is an example of the code necessary to make an ARGUS PDF in AFit:

```
// the discriminating variable x is the beam constrained B meson mass.
RooRealVar x(``x'', ``'', 5.25, 5.29);

// The AUGUS PDF
AFitArgus argus(x, ``arguspdf'');
RooAbsPdf * argusPDF = argus.getPdf();
```

By default the ARGUS PDF will have an endpoint $m_0 = 5.29$ and slope $\xi = -50$, as shown in Figure 1.

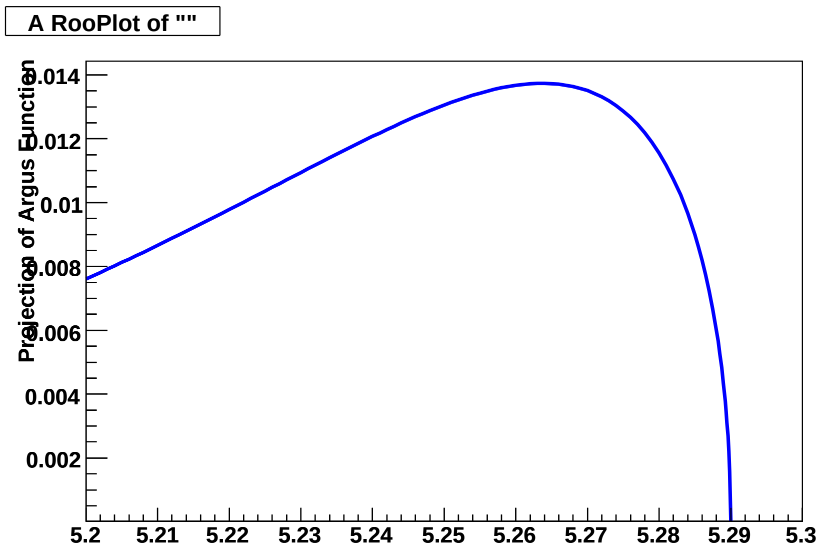

Figure 1: An example of the ARGUS PDF shape.

2.3 Breit Wigner

2.3.1 Non-relativistic Breit Wigner (or Cauchy) function

The non-relativistic Breit Wigner function (also called a Cauchy function) is given by

$$\mathcal{P}(x\,; m_0, \Gamma) = \frac{1}{(x - m_0^2) + (\Gamma/2)^2)} \tag{2}$$

and is shown in Figure 2. Here m_0 is the position of the peak and Γ is the width of the peak.

The following code snippet is sufficient to build a non-relativistic Breit-Wigner PDF (`bwPDF`). By default the mass and width of the PDF are 0.770 and 0.150, respectively. The variable names for the mass and width of the PDF are the name of the pdf, in this case `bwpdf`, followed by `m0` and `width`.

```
// x is the resonance mass (e.g.  m_rho)
RooRealVar x(``x'', ``'', 0.5, 1.0);
AFitBreitWigner bw(x, ``bwpdf'');
RooAbsPdf * bwPDF = bw.getPdf();
```

2.3.2 Relativistic Breit Wigner (with a Blatt-Weisskopf Form Factor)

The relativistic Breit Wigner form implemented in AFit is described in the following.

$$\mathcal{P}(m\,; m_0, \Gamma_0, J, R) = \frac{m^2}{(m^2 - m_0^2)^2 + m_0^2\Gamma^2(m)} \tag{3}$$

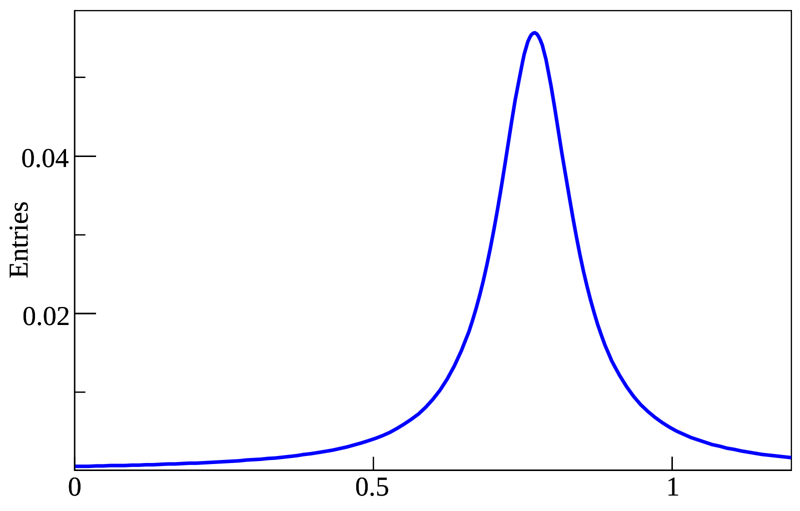

Figure 2: An example of the Breit Wigner PDF shape.

where the mass dependent width is given by

$$\Gamma(m) = \Gamma_0 \frac{m_0}{m} \left(\frac{k(m)}{k(m_0)} \right)^{2J+1} \frac{F(Rk(m))}{F(Rk(m_0))}, \tag{4}$$

$$k(m) = \frac{m}{2} \left(1 - \frac{(m_a + m_b)^2}{m^2} \right)^{1/2} \left(1 - \frac{(m_a - m_b)^2}{m^2} \right)^{1/2}, \tag{5}$$

and the functions F are the spin dependent Blatt-Weisskopf form factors,

$$F^{J=0}(x) = 1 \tag{6}$$

$$F^{J=1}(x) = \frac{1}{1+x^2} \tag{7}$$

$$F^{J=2}(x) = \frac{1}{9+3x^2+x^4} \tag{8}$$

In Eq. 4 m_0 is the mass of the resonance, Γ_0 is its width, J is its spin and R is the interaction radius. The event generator on BaBar, EvtGen, uses a range parameter, of $3.0GeV^{-1} \simeq 0.6fm$. The parameters m_a and m_b are the masses of the daughters of the decaying resonance.

The corresponding distribution for a ρ meson described by this PDF is shown in Fig. 3, where $m_0 = 0.77\mathrm{GeV/c}^2$, $\Gamma_0 = 0.15\mathrm{GeV}$, $J = 1$, $R = 3.0GeV^{-1}$, and $m_a = m_b = 0.135\mathrm{GeV/c}^2$. The allowed values of J are 0, 1, and 2.

The following code snippet illustrates how to instantiate an AFitRelBreitWigner.

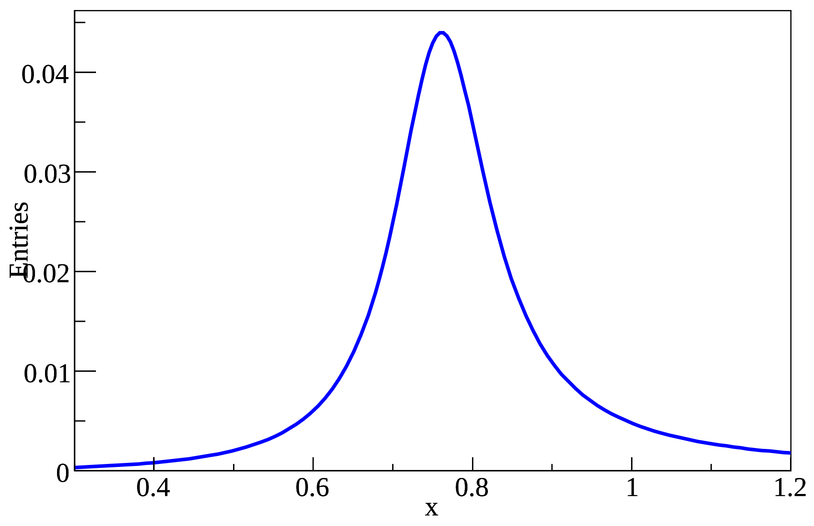

Figure 3: An example of the relativistic Breit Wigner PDF shape for a ρ meson.

```
RooRealVar x(``x'', ``'', 0.0, 1.0);
AFitRelBreitWigner bw(x, ``bwpdf'');
RooAbsPdf * bwPDF = bw.getPdf();
```

2.4 Bukin Function

The Bukin function is given by

$$\mathcal{P}(x\,; x_p, \sigma_p, \xi, \rho) = A_p \exp\left[\frac{\xi\sqrt{\xi^2+1}(x-x_1)\sqrt{2\ln 2}}{\sigma_p\left(\sqrt{\xi^2+1}-\xi\right)^2 \ln\left(\sqrt{\xi^2+1}+\xi\right)} + \rho\left(\frac{x-x_i}{x_p-x_i}\right)^2 - \ln 2\right], \quad (9)$$

where $\rho = \rho_1$ and $x_i = x_1$ for $x < x_1$, and $\rho = rho_2$ and $x_i = x_2$ when $x \geq x_2$. This function is shown in Figure 4 for $\xi = -0.2$, $x_p = 0.5$, $\sigma_p = 0.1$, $\rho_1 = 0.1$, and $\rho_2 = 0.2$.

$$x_{1,2} = x_p + \sigma_p\sqrt{2\ln 2}\left(\frac{\xi}{\sqrt{\xi+1}} \mp 1\right) \quad (10)$$

The parameters x_p and σ_p are the peak position and width (FWHM/2.35), and ξ is an asymmetry parameter. The numerical integral of the Bukin function does not always converge, so you should check to make sure that the PDF is correctly evaluated (by plotting the fitted result) if you use this PFD.

The following code snippet will make a bukin pdf with the same PDF parameters as used in Figure 4.

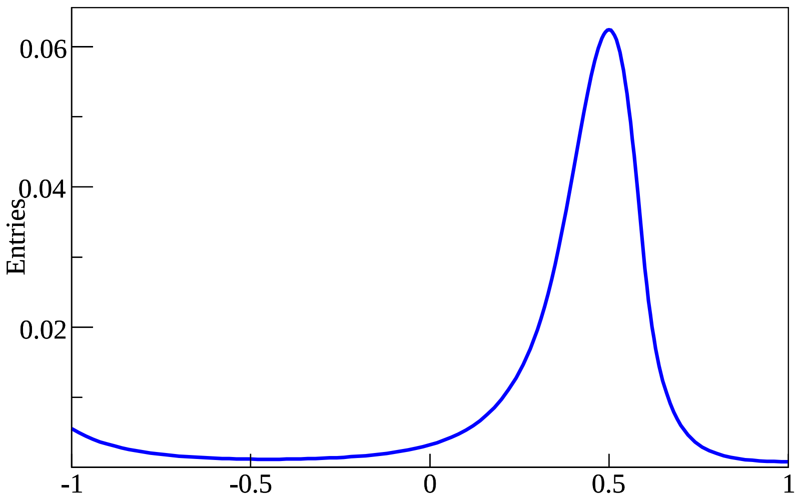

Figure 4: An example of the Bukin PDF shape.

```
RooRealVar x(‘‘x’’, ‘‘’’, 0.0, 1.0);
AFitBukin bukin(x, ‘‘bpdf’’);
RooAbsPdf * bukinPDF = bukin.getPdf();
```

2.5 Chebychev Polynomial

Chebychev Polynomial distribution: The Cheby shape is given by

$$\mathcal{P}(x, p_i) = 1 + \sum_{i=1}^{n} p_i T_i(x), \tag{11}$$

where the parameters p_i are coefficints of the functions T_i, and the T_i are define elsewhere[3]. The RooFit user guide advises that one should use a Chebychev Polynomial over a polynomial wherever possible, as the coefficients of a Chebychev Polynomial have smaller correlations than the coefficients of a polynomial. The corollory of this is that fits behave better when using a Chebychev Polynomial than when using the equivalent polynomial. The following code snippet will make a Chebychev Polynomial of order 3:

```
Int_t iOrder = 3;
AFitCheby chebybld(x, ‘‘pdf’’, iOrder);
RooAbsPdf * cheby = chebybld.getPdf();
```

[3]For example see: http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html

2.6 Crystal Ball

Crystal Ball (CB) distribution: The CB shape is a Gaussian with an exponential tail as defined in [7]. The functional form implemented in RooFit is given by

$$\mathcal{P}(m; m_0, \sigma, \alpha, n) = \frac{1}{N} \times e^{-(m-m_0)^2/(2\sigma^2)}\,; m > m_0 - \alpha\sigma, \tag{12}$$

$$= \frac{1}{N} \times \frac{(n/\alpha)^n \exp(-\alpha^2/2)}{((m_0 - m)/\sigma + n/\alpha - \alpha)^n}\,; m \leq m_0 - \alpha\sigma, \tag{13}$$

where we use the abbreviation: $alpha = \alpha$, $n = n$, $Mean = m_0$ and $resn = \sigma$.

This PDF has a type 'cbshape' when using the AFitPdfFactory (see below). The following is an example of the code necessary to make a Crystal Ball PDF in AFit, and Figure 5 shows the corresponding PDF distribution.

```
RooRealVar x(``x'', ``'', -5.0, 5.0);

// The Crystal Ball PDF
AFitCBShape cbshape(x, ``mypdfname'');
RooAbsPdf * cbPDF = cbshape.getPdf();
```

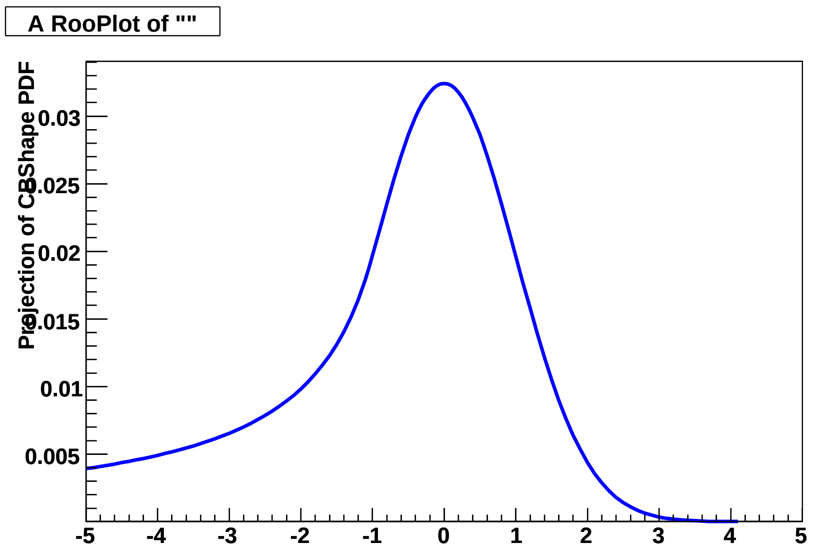

Figure 5: An example of the Crystal Ball PDF shape.

2.7 Decay Models

2.7.1 The Decay Model

The `AFitDecay` class is the wrapper for the `RooDecay` class. This is used to model lifetime decay of non-mixing particles as a function of Δt. The functional form of this PDF is

$$\mathcal{P}(\Delta t, \sigma(\Delta t)) \quad = \quad \frac{e^{-|\Delta t|/\tau}}{\tau} \oplus \mathcal{R}(\Delta t, \sigma(\Delta t)).$$

where τ is the lifetime of the particle. The resolution function $\mathcal{R}$ is convolved with the physical time-dependence as indicated. Figure 6 illustrates the shape of this PDF.

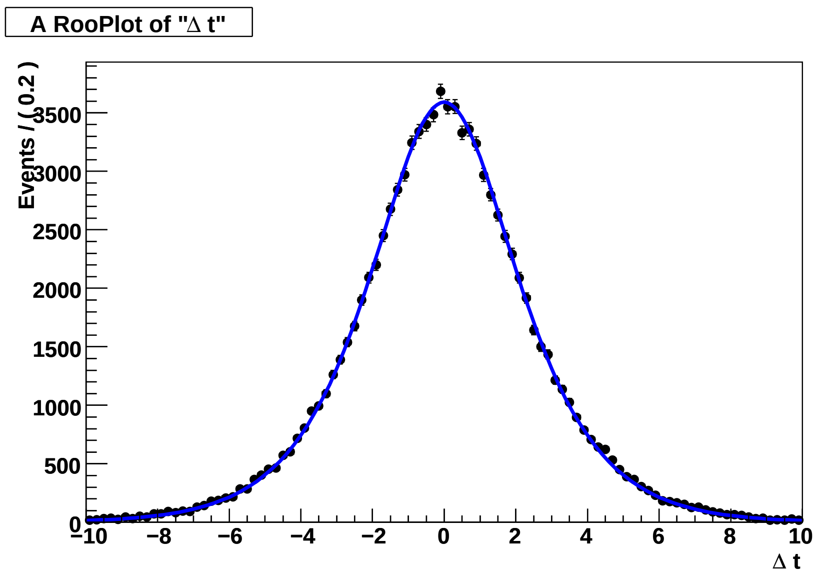

Figure 6: An example of the Decay PDF shape.

NOTE: When using this model, it is imperative that the RooAbsPdf of the resolution function you intend to use is instantiated before the Decay model's RooAbsPdf.

2.7.2 The BDecay Model

The `AFitBDecay` class is the wrapper for the `RooBDecay` class. This is used to model lifetime decay of non-mixing particles as a function of Δt. The functional form of this PDF is the most general description of B decay time distribution with effects of CP violation, mixing and life time differences. Dilution is not explicitly included in this pdf. See the RooFit documentation for more information.

NOTE: When using this model, it is imperative that the RooAbsPdf of the resolution function you intend to use is instantiated before the BCPGenDecay model's RooAbsPdf.

2.7.3 The BCPGenDecay Model

The `AFitBCPGenDecay` class is the wrapper for the time-dependent CP asymmetry `RooBCPGenDecay` class. This is used to measure CP asymmetries in $\Upsilon(4S) \rightarrow B^0\overline{B}^0$ decays at BaBar. Section 6.5 illustrates how to use this class. The functional form of this PDF is

$$\begin{aligned}\mathcal{P}_{\pm}(\Delta t, \sigma(\Delta t)) &= \frac{e^{-|\Delta t|/\tau_{B^0}}}{4\tau_{B^0}}\left((1 \mp \Delta\omega) \pm (1-2\omega) \times [S_f \sin(\Delta m_d \Delta t) - C_f \cos(\Delta m_d \Delta t)]\right) \\ &\quad \oplus \mathcal{R}(\Delta t, \sigma(\Delta t)).\end{aligned}$$

where $\mathcal{P}_{+(-)}$ describes B^0 ($\overline{B}^0$) tagged events, τ_{B^0} is the B^0 lifetime, ω is the mistag probability, $\Delta\omega$ is the mistag probability difference between B^0 and $\overline{B}^0$ tagged events, Δm_d is the $B^0 - \overline{B}^0$ mixing frequency, and S and C are the CP asymmetry parameters. The resolution function $\mathcal{R}$ is convolved with the physical time-dependence as indicated.

It is possible to blind S and C by setting the blindingState to blind, ensuring that the blindStringS and blindStringC variables have been set appropriately.

NOTE: When using this model, it is imperative that the RooAbsPdf of the resolution function you intend to use is instantiated before the BCPGenDecay model's RooAbsPdf.

2.8 Disappearance PDF

The neutrino disappearance probability is given by

$$\mathcal{P}(E; \theta, A, \Delta m^2, L) = 1 - \sin^2(2\theta)\sin^2(A\Delta m^2 L/E), \tag{14}$$

where θ is the mixing angle, $A \simeq 1.267$ given current data, L is the baseline of the neutrino experiment (km), and E is the neutrino energy (GeV). The class AFitDisapperance is the implementation of this PDF. The corresponding distribution produced by this class is shown in Figure 7 for $L = 265$, $A = 1.267$, and $\Delta m^2 = 0.0024$. The observed PDF for low neutrino energy (< 0.1 GeV) is an artifact of the granularity of the RooCurve used to create the plot. The numerical PDF is reproduced correctly.

2.9 Exponential

The exponential function defined as

$$\mathcal{P}(x; \gamma) = e^{\gamma x}, \tag{15}$$

where γ is the slope of the exponential. This PDF has a type 'exponential' when using the AFitPdfFactory. If the category fitLifetime has the value 'true', then the slope of the exponential is replaced by $-1/\gamma$, where γ is fitted as a lifetime and the PDF becomes:

$$\mathcal{P}(x; \gamma) = e^{-x/\tau}, \tag{16}$$

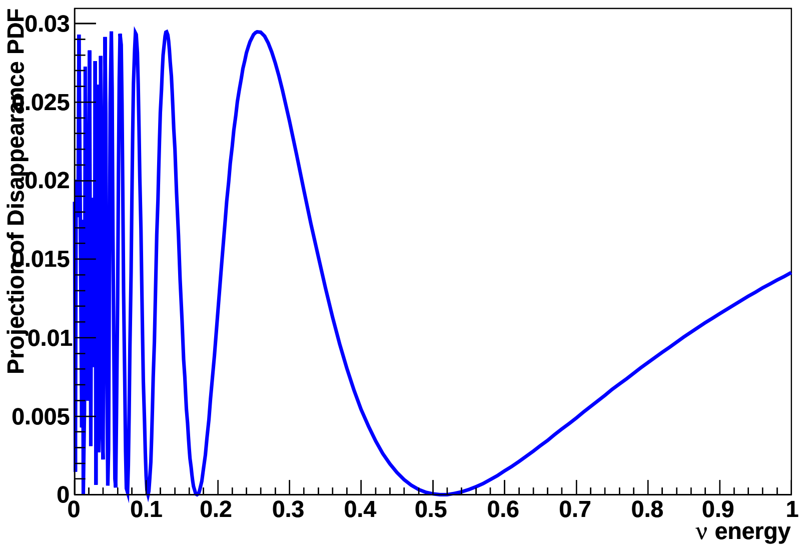

Figure 7: An example of the neutrino disappearance PDF shape. The irregular oscillation amplitude visible for low ν energy is a plotting artifact.

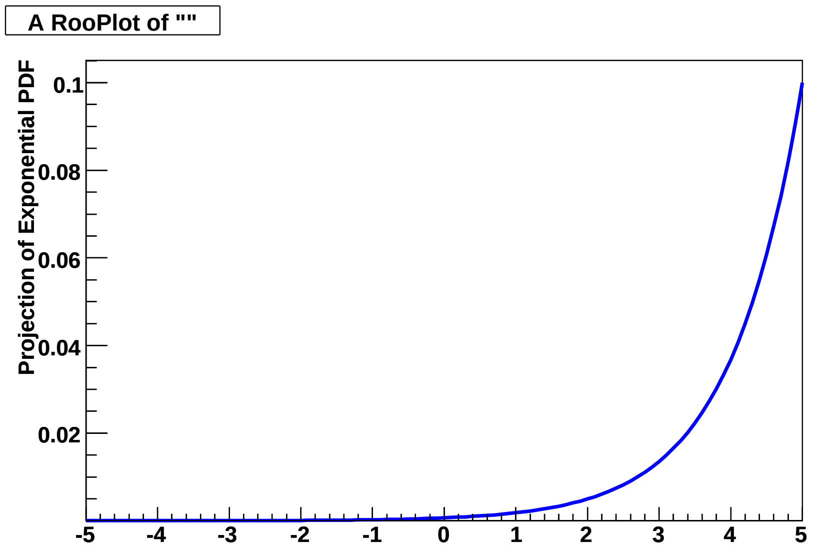

Figure 8: An example of the exponential PDF shape.

The fitLifetime category has the name $< pdfname >$ _ $< variablename > fitLifetime$. This option is useful when trying to determine the lifetime of exponentially decaying sample of data. Figure 8 shows an example of the exponential function PDF.

The following code snippet will make an exponential PDF with a decay constant of 1.0.

```
RooRealVar x(‘‘x’’, ‘‘’’, 0.0, 20.0);

// The Exponential PDF
AFitExponential expshape(x, ‘‘mypdfname’’);
RooAbsPdf * expPDF = expshape.getPdf();
```

2.10 Flatte Function

An example of the Flatte function distribution is shown in Fig. 9. More details will be added in due course.

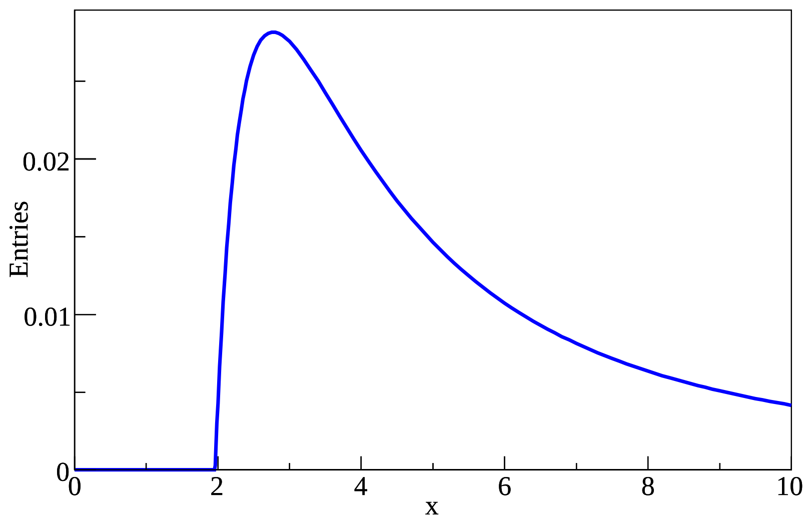

Figure 9: An example of the Flatte PDF shape.

The following code snippet is an illustration of how to construct an AFitFlatte pdf.

```
RooRealVar x(‘‘x’’, ‘‘’’, 0, 10.0);

AFitFlatte flatteshape(x, ‘‘mypdfname’’);
RooAbsPdf * flattePDF = flatteshape.getPdf();
```

2.11 Gaussian

This is defined by a mean and width (μ and σ) and is given by

$$\mathcal{P}(x\,;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-[x-\mu]^2/2\sigma^2\right) \tag{17}$$

The Gaussian function implemented in AFit, allows you to choose to have different widths above and below the mean value.

This PDF has a type 'gaussian' when using the AFitPdfFactory, and Figure 10 shows an example of the Gaussian PDF shape.

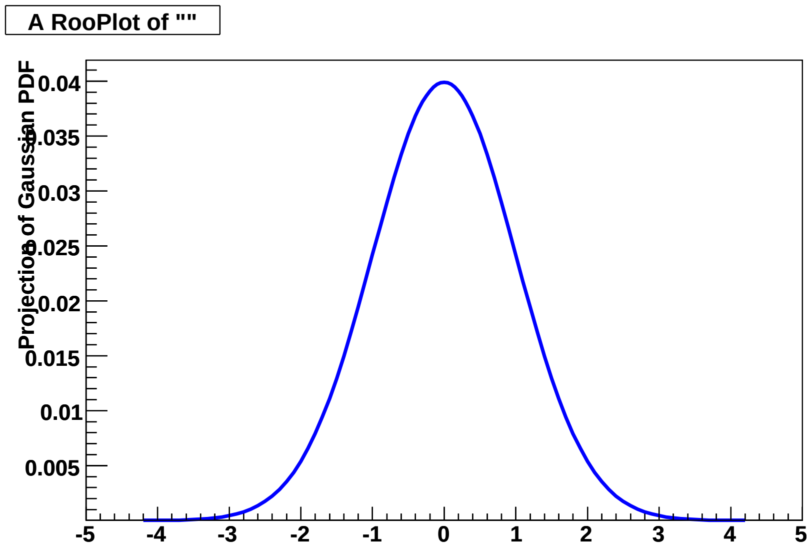

Figure 10: An example of the Gaussian PDF shape.

The following code snippet will make a Gaussian PDF with a mean of 0.0 and a width of 1.0.

```
RooRealVar x(‘‘x’’, ‘‘’’, -4.0, 4.0);

// The Gaussian PDF
AFitGaussian gausshape(x, ‘‘mypdfname’’);
RooAbsPdf * gausPDF = gausshape.getPdf();
```

2.12 Generic PDF with functional form $f(x)$

This PDF has type 'generic' when using the AFitPdfFactory. The functional form of a Generic PDF is defined by a string input into the constructor of the class. For example, in order to make a generic PDF for the function $f(x) = x^2 + x + 1$, one can do the following:

```
RooRealVar x(''x'', '''', -5.0, 5.0);

AFitGeneric genshape(x, ''mypdfname'', ''@0*@0+@0+1'');
RooAbsPdf * genericPdf = genshape.getPdf();
```

The corresponding shape of this generic PDF example is shown in Figure 11.

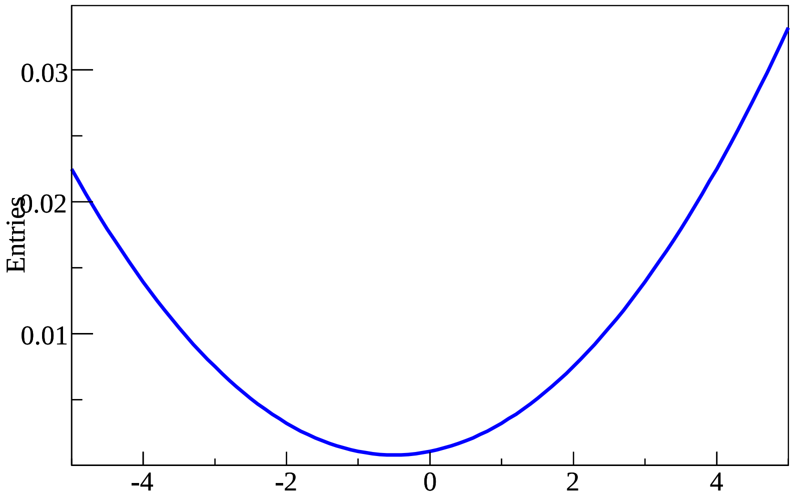

Figure 11: An example of the generic PDF shape for the function $f(x) = x^2 + x + 1$.

2.13 Gounaris-Sakurai lineshape PDF

This is similar in shape to the relativistic Breit-Wigner described in Sec. 2.3.2. Gounaris-Sakurai (GS) distribution is a model of the P-wave $\pi\pi$ scattering amplitude [8]: The parameters mean and width of the resonance are m_0 and Γ_0. The GS PDF used is for $|A|^2$ and is defined as

$$GS(m\,; m_0, \Gamma_0, J, R) = \frac{(1 + d.\Gamma_0/m_0)^2}{(m^2 - m_0^2 - f(m^2))^2 + m_0^2\Gamma^2(m)}, \tag{18}$$

where

$$\begin{aligned}
f(s) &= \Gamma_0 \frac{m_0^2}{k^3(m_0)} \left[k^2(m)[h(s) - h(m_0^2)] + (m_0^2 - s)k^2(m_0)dh/ds|_{s=m_0^2}\right], & (19)\\
h(s) &= \frac{2k(m)}{\pi\sqrt{(s)}} \ln\left(\frac{\sqrt{s} + 2k(m)}{2m_\pi}\right), & (20)\\
dh/ds|_{s=m_0^2} &= h(m_0^2)\left[(8k^2(m_0))^{-1} - (2m_0^2)^{-1}\right] + (2\pi m_0^2)^{-1}, & (21)
\end{aligned}$$

$$d = \frac{3m_\pi^2}{\pi k^2(m_0)} \ln\left(\frac{m_0 + 2k(m_0)}{2m_\pi}\right) + \frac{m_0}{2\pi k(m_0)} - \frac{m_\pi^2 m_0}{\pi k^3(m_0)}, \tag{22}$$

$$\Gamma(m) = \Gamma_0 \frac{m_0}{m}\left(\frac{k(m)}{k(m_0)}\right)^{2J+1}, \tag{23}$$

$$\tag{24}$$

$\Gamma(m)$ is the mass dependent width of Eq. 23, $s = m^2$, and $k(\sqrt{s})$ is defined in Eq. 5.

2.14 Helicity PDF

This PDF has type 'helicity' when using the AFitPdfFactory. The functional form of the helicity is depends on the value of the type string specified prior to building the PDF and this PDF is implemented in order to study angular correlations in components of type `vvpolarisation`. For the helicity angle θ, the longitudinally polarised part of the `vvpolarisation` component should have a $\cos\theta$ distribution of the form `xsqr`, and the transversely polarised part of the `vvpolarisation` component should have a $\cos\theta$ distribution of the form `1-xsqr`. In practice any PDF of this form should be modulated by an acceptance function.

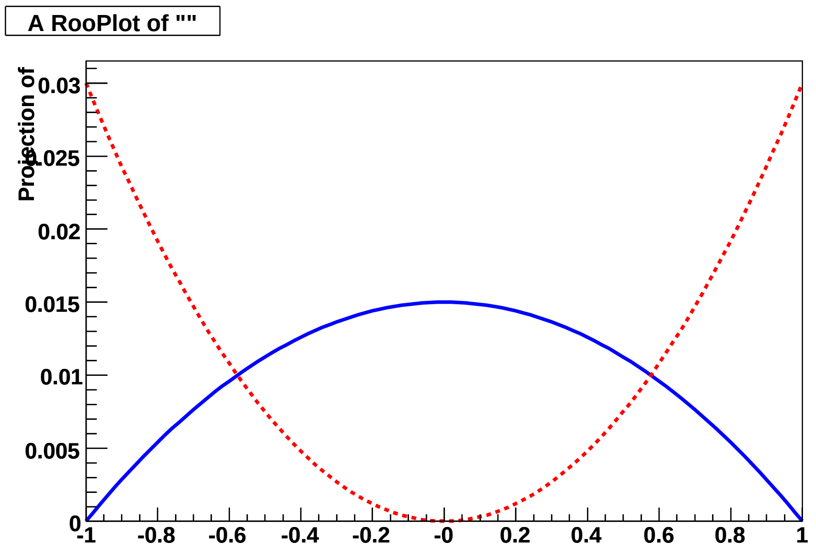

Figure 12: An example of the helicity PDF shape for (dashed) longitudinal and (solid) transverse polarisation forms.

The following code snippet will make a helicity PDF shape with an `xsqr` distribution:

```
AFitHelicity hbld(x, ‘‘pdf’’);
RooAbsPdf * h = hbld.getPdf();
```

2.15 Histogram (non-parametric PDF)

It is possible to construct a non-parametric PDF based on an input histogram using the RooHistPdf class. The AFit wrapper of this class is AFitHist. The value of the PDF for a given value of x corresponds to the normalised bin content of the histogram used to define the PDF (unless interpolation is used between adjacent bins).

In order to build a Histogram PDF for a discriminating variable x, you need to enter the following

```
AFitHist pdfbld(x, ‘‘pdf’’);
pdfbld.datafile.setVal(‘‘myfile.root’’);
pdfbld.histname.setVal(‘‘hist’’);
pdfbld.order.setVal(2);
RooAbsPdf * pdf = pdfbld.getPdf();
```

where the histogram needed to define the PDF is called `hist` and can be found in the ROOT file `myfile.root`. An assertion is made if the specified histogram does not exist. Figure 13 shows an example of this type of PDF given an input histogram. The data in the histogram are the result of randomly filling the histogram according to a Gaussian distribution with a mean of 0.5 and width of 0.1.

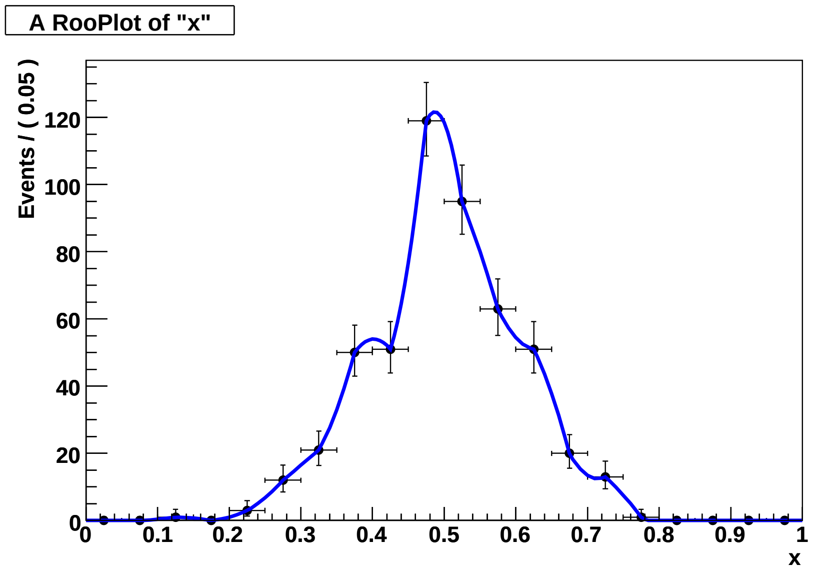

Figure 13: An example of the Histogram PDF shape (solid), compared with the original data (points).

2.16 KEYS (non-parametric PDF)

The KEYS algorithm (Kernal Estimation of Your Shapes) [9] can be used to obtain a smoothed non-parametric PDF representation of a sample of data or Monte Carlo simulated data. The use of the KEYS PDF is similar to that of the histogram PDF described above. The main difference is that fits using a KEYS pdf will be a lot slower than those using a histogram PDF. The reason for this is that a Gaussian kernel is computed for each event in the data set used to construct a KEYS PDF, and the PDF is evaluated at each point by computing a sum over all events. In practice it is usually beneficial to compute a KEYS PDF once and then store the output shape as a histogram for use in all later fitting and validations.

Figure 14 shows an example of this type of PDF given an input histogram. The data in the figure are the result of randomly filling the histogram according to a Gaussian distribution with a mean of 0.5 and width of 0.1.

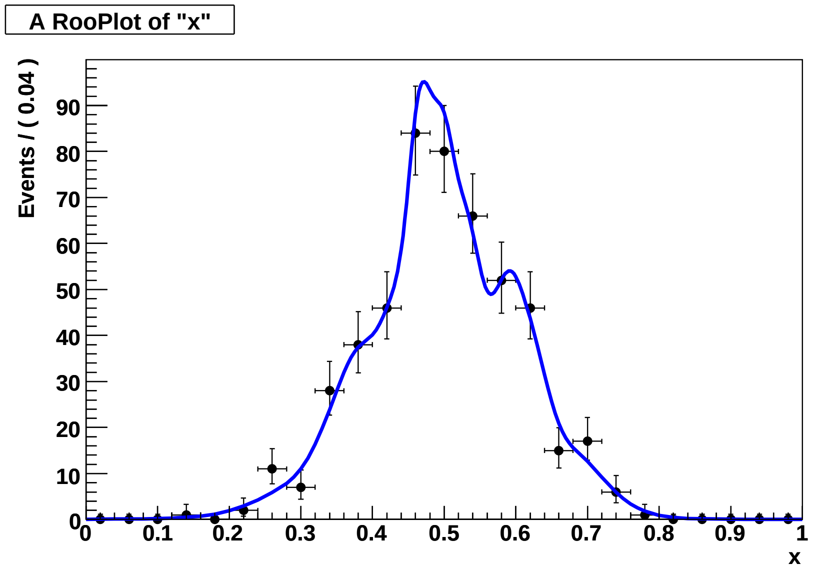

Figure 14: An example of the KEYS PDF shape (solid), compared with the original data (points).

In order to build a KEYS PDF for a discriminating variable x, you need to enter the following

```
AFitKeys pdfbld(x, ‘‘pdf’’);
pdfbld.setParameter(‘‘file’’, ‘‘myfile.root’’);
pdfbld.setParameter(‘‘tree’’, ‘‘treename’’);
pdfbld.setParameter(‘‘rho’’,1);
pdfbld.setParameter(‘‘mirror’’, ‘‘NoMirror’’);
RooAbsPdf * pdf = pdfbld.getPdf();
```

where datafile sets the ROOT file containing the TTree treename used to construct the PDF, ρ

is a smoothing parameter, and mirror is a RooCategory that defines how edge effects are dealt with by RooKeysPdf. The allowed options for mirror are: NoMirror, MirrorLeft, MirrorRight, MirrorBoth, MirrorAsymLeft, MirrorAsymLeftRight, MirrorAsymRight, MirrorLeftAsymRight, MirrorAsymBoth. The default is $\rho = 1$ and mirror set to NoMirror.

2.17 Landau

This PDF has a type 'landau' when using the AFitPdfFactory, and Figure 15 shows an example of the Landau PDF shape. The functional form of the Landau distribution is given in Ref. [10] and this is often used to describe the fluctuations in energy loss of a charged particle passing through a thin layer of material.

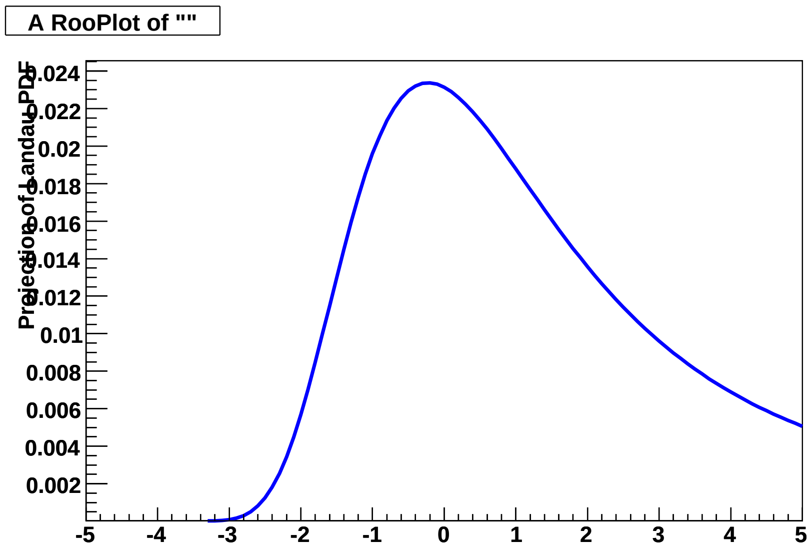

Figure 15: An example of the Landau PDF shape.

The following code snippet will construct a Landau PDF with a mean of 0.0, and a width of 1.0.

```
AFitLandau landaubld(x, ‘‘pdf’’);
RooAbsPdf * landau = landaubld.getPdf();
```

2.18 Novosibirsk

This PDF has a type 'novosibirsk' when using the AFitPdfFactory, and Figure 16 shows an example of the Novosibirsk PDF shape. The functional form of the Novosibirsk function is

$$P(x) \quad = \quad e^{-0.5^{(\ln q_y)^2/\Lambda^2 + \Lambda^2}}, \tag{25}$$

$$q_y \quad = \quad 1 + \Lambda(x - x_0)/\sigma \times \frac{\sinh(\Lambda\sqrt{\ln 4})}{\Lambda\sqrt{\ln 4}}, \tag{26}$$

x_0 is the peak position, σ is the width of the peak, and Λ is a parameter describing the tail of the distribution.

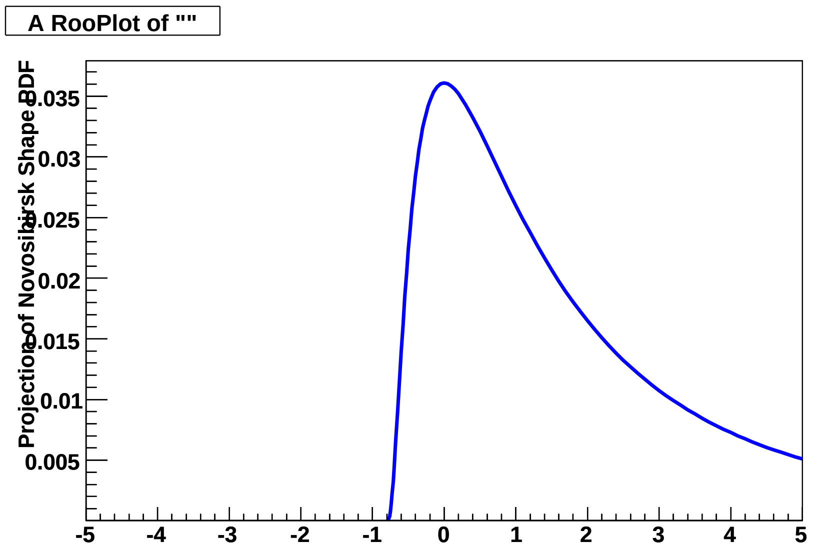

Figure 16: An example of the Novosibirsk PDF shape.

The following code snippet will construct a Novosibirsk PDF with a mean of 0.0, a width of 1.0, and a tail parameter of 1.0.

```
AFitNovosibirskShape novbld(x, ‘‘pdf’’);
RooAbsPdf * nov = novbld.getPdf();
```

2.19 Polynomial

The polynomial is defined as

$$\mathcal{P}(x; p_i) = \sum_{i=1}^{N} p_i x^i, \tag{27}$$

where the p_i are coefficients. This PDF has a type ’polynomial’ when using the AFitPdfFactory, and Figure 17 shows an example of the polynomial PDF shape.

The following code snippet will construct a cubic polynomial function (this is specified by the `iOrder` argument). By default the polynomial parameters are set to 0.0, so you need to modify the parameters either via a configuration file, or by accessing the RooRealVar’s directly in order to make a non-trivial shape.

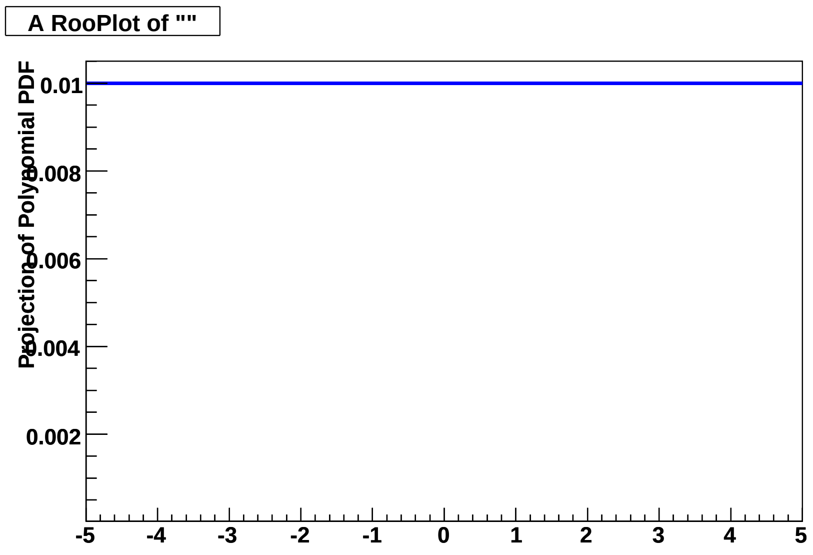

Figure 17: An example of the polynomial PDF shape.

```
Int_t iOrder = 3;
AFitPoly polybld(x, ``pdf'', iOrder);
RooAbsPdf * poly = polybld.getPdf();
```

2.20 Parametric Step Function

The Parametric Step Function (PSF) is a PDF where data are binned in one dimension, and the parameters of the PDF are the fractions of probability in each bin. As binning can be non-uniform by definition, the fitted fractions are, in general, not the heights of each bin when plotted, but also depend on the bin width. Figure 18 shows an example of using the PSF pdf.

The PSF pdf has N bins, so there are N coefficients ("pdfname_PSFcoef_i") and N+1 bin boundaries ("pdfname_PSFlimit_i") that have to be set in order to make this PDF. The following code snippet will construct a parametric step function PDF similar to the one shown in the Figure.

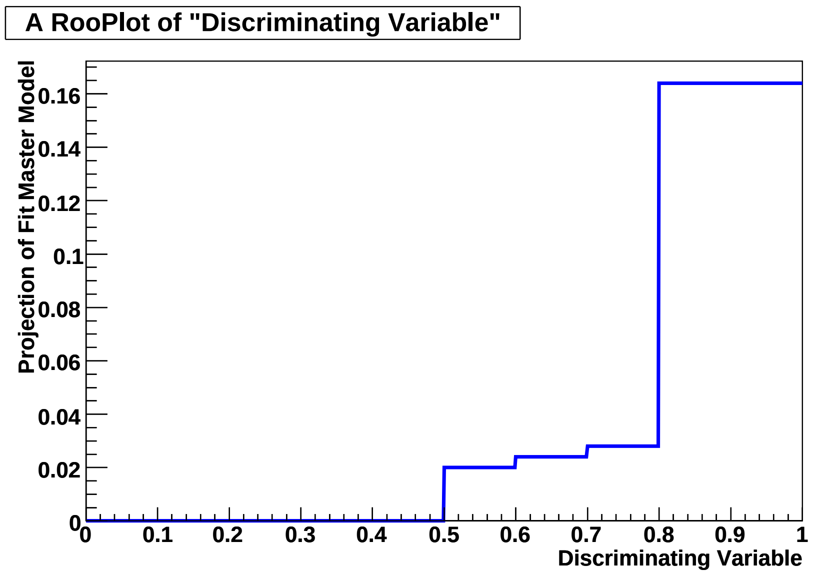

Figure 18: An example of the Parametric Step Function shape.

```
// Instantiate the discriminating variable
RooRealVar x("x", "", 0.0, 1.0);

// Set the relative weights of each bin
RooRealVar n0("n0", "", 0.1);
RooRealVar n1("n1", "", 0.2);
RooRealVar n2("n2", "", 0.3);
RooRealVar n3("n3", "", 0.4);
RooRealVar n4("n4", "", 0.1);

RooArgList coefList;
coefList.add(n0);
coefList.add(n1);
coefList.add(n2);
coefList.add(n3);
coefList.add(n4);

// Set the bin boundaries - there are N+1 boundaries
// Note that the first and last boundary match the limits
// of the discriminating variable x.
TArrayD limits;
limits.Set(6);
limits[0] = 0.0;
limits[1] = 0.5;
limits[2] = 0.6;
limits[3] = 0.7;
limits[4] = 0.9;
limits[5] = 1.0;

RooParametricStepFunction pdf("pdf", "ParametricStepFunction PDF",
                                    x, coefList, limits, 5);
```

2.21 Resolution

The resolution function implemented in AFit is a triple Gaussian function of the form

$$\begin{aligned}\mathcal{P}(x;p_i) &= f_{core}G_{core}(x,\sigma(x),\mu_{core},\sigma_{core}) + f_{tail}G_{tail}(x,\sigma(x),\mu_{tail},\sigma_{tail}) \\ &\quad +(1-f_{core}-f_{tail})G_{outlier}(x,\mu_{outlier},\sigma_{outlier}) \qquad (28)\end{aligned}$$

where x is the discriminating variable, $\sigma(x)$ is the uncertainty on the discriminating variable, and the μ_i and σ_i are the means and widths of the i^{th} Gaussian, where $i = core, tail, outlier$. It is possible to multiply the mean and with of the core and tail Gaussians by $\sigma(x)$, however the default behaviour is not to do this.

In order to scale the mean and width parameters by the error on x, the following parameters should be set to `yes`

```
scaleCoreMean
scaleTailMean
scaleCoreWidth
scaleTailWidth
```

An example of the resolution function PDF for the proper-time difference distribution (Δt) in BaBar is shown in Figure 19 for $B^0 \to \pi^+\pi^-$ Monte Carlo simulated data.

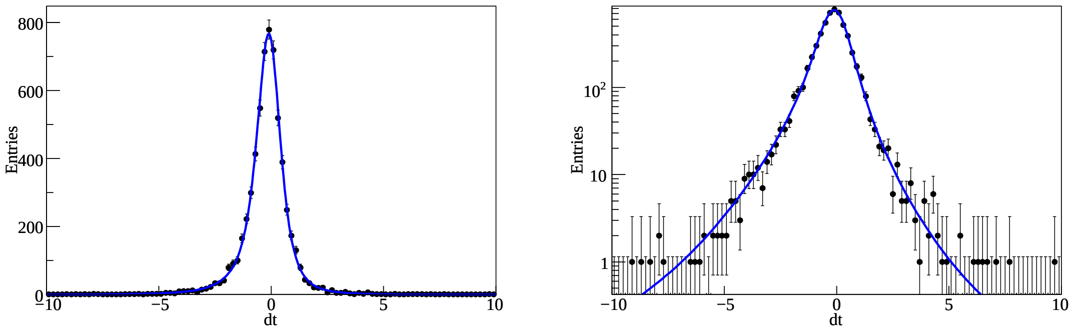

Figure 19: An example of the resolution function PDF shape. The figure on the right shows the same plot on a log scale.

A resolution function PDF can be constructed using the following steps

```
// the discriminating variable is deltat and the conditional variable is
// is the error on deltat:  deltatErr
RooRealVar deltat(``reso_dt'', ``dt'', -10.0, 10.0);
RooRealVar deltatErr(``deltaterr'', ``sdt'', 1.2, 0.0, 2.50);

AFitResolution resoBld(&deltat, &deltatErr, ``Reso'');
RooAbsPdf * pdf = resoBld.getPdf();
```

and there is a detailed example of how to use the AFitResolution class in Section 6.3

2.22 Sigmoid

This PDF has a type 'sigmoid' when using the AFitPdfFactory. The function implemented is

$$\mathcal{P}(x; a, b) = \frac{1}{1 + e^{a(x+b)}}, \tag{29}$$

where the coefficients a and b are an exponent scale factor, and x-offset respectively. Figure 20 shows an example of the Sigmoid PDF shape.

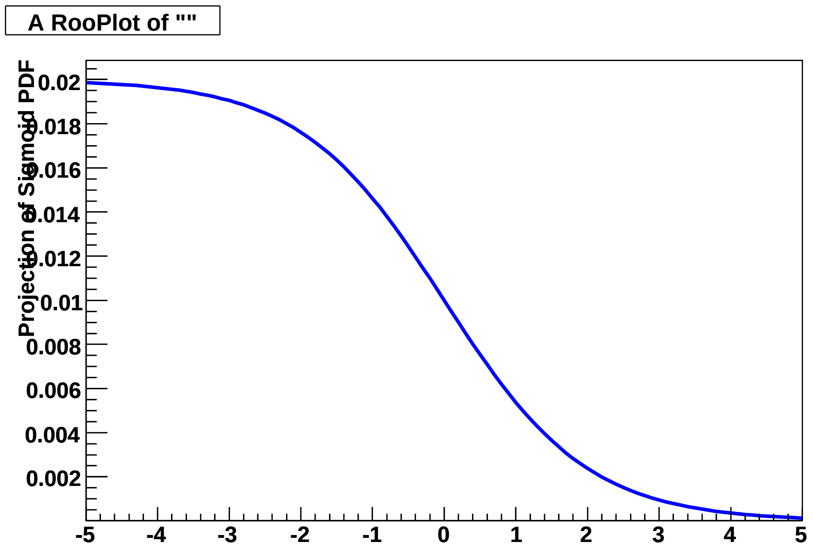

Figure 20: An example of the sigmoid PDF shape.

The following code snippet will make a sigmoid function PDF with $a = 1.0$, and $b = 0.0$.

```
AFitSigmoid sigbld(x, ``pdf'');
RooAbsPdf * sigmoid = sigbld.getPdf();
```

2.23 Step Function

It can be useful to impose a sharp cut-off to a PDF. In such cases it is useful use a step function PDF where

$$\mathcal{P}(x) = 1, \tag{30}$$

between x_a and x_b, and $\mathcal{P}(x) = 0$ elsewhere. Similarly it can also be useful to veto a region of x using the complement of the step function. Figure 21 shows an example of the step and veto functions where $x_a = 2$ and $x_b = 3$.

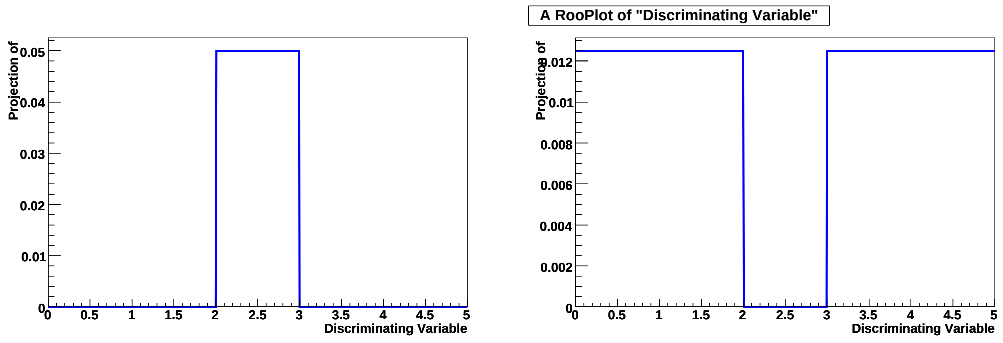

Figure 21: An example of (left) a step and (right) a veto function.

The following code snippet will make a step function PDF with a step between $x = 0.0$, and $x = 1.0$.

```
RooRealVar x(‘‘x’’, ‘‘’’, -1.0, 2.0);
AFitStepFunction stepbld(x, ‘‘pdf’’);
RooAbsPdf * step = stepbld.getPdf();
```

2.24 Voigtian

This PDF has a type 'voigtian' when using the AFitPdfFactory. Figure 22 shows an example of the Voigtian PDF shape. This function is defined as the convolution of a Gaussian and a Breit-Wigner distribution. It is useful when trying to describe mass peaks with the particle width Γ is comparable to the detector resolution. The functional form of this PDF is

$$\mathcal{P}(x; m, \Gamma, \sigma) = \frac{1}{N} \int_{-\infty}^{+\infty} G(x'; 0, \sigma) BW(x - x'; m, \Gamma) dx'. \tag{31}$$

where G is the Gaussian and BW is the Breit-Wigner.

The following code snippet will make a Voigtian PDF shape with a mean of 0.0, a width of 1.0, and a σ of 1.0:

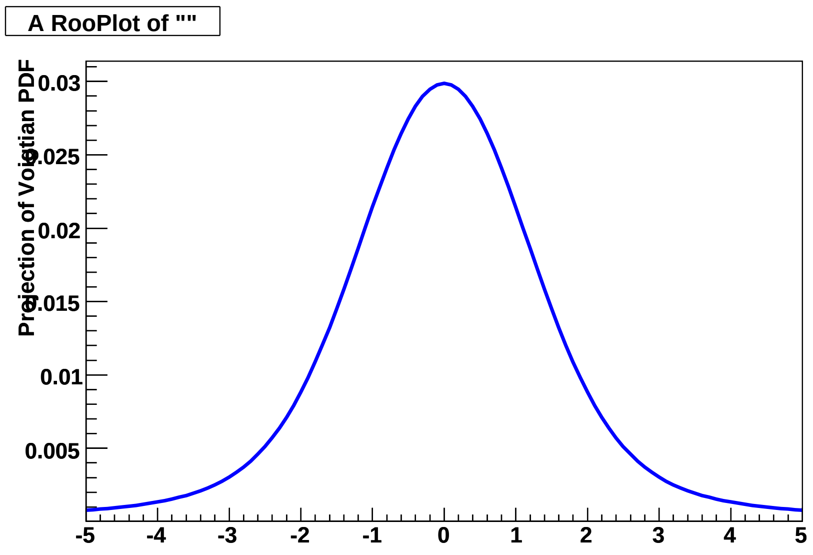

Figure 22: An example of the Voigtian PDF shape.

```
AFitVoigtianShape vbld(x, ‘‘pdf’’);
RooAbsPdf * v = vbld.getPdf();
```

2.25 Composite Add PDF

Composite PDFs can be constructed from their individual components using an AFitAddPdf. Each component is added with a relative fraction as follows

$$\mathcal{P}(x; f_i) = f_1 PDF_1(x) + f_2 PDF_2(x) + \ldots (1 - f_1 - f_2 \ldots - f_{N-1}) PDF_N(x). \qquad (32)$$

This PDF has a type ’add:x,y,...’ when using the AFitPdfFactory. The comma separated variables after the colon specify the different PDF types to be added together.

2.26 Composite Multiply PDF

This PDF has a type ’multiply’ followed by a colon ’:’ and a comma separated list of PDF types to instantiate and multiply together. The functional form of this PDF is given by

$$P(x) \quad = \quad P_1(x) * P_2(x) * \ldots * P_n(x) \qquad (33)$$

The multiplication is implemented by using a RooGenericPDF, and the limit of the number of PDFs that can be multiplied together is governed by the computing resources. Figure 23 shows a helicity PDF multiplied by a polynomial.

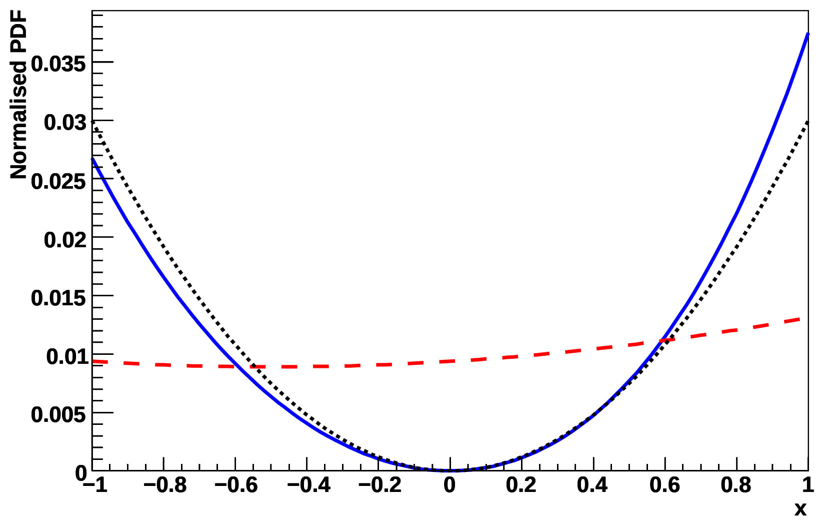

Figure 23: An example of the Multiply PDF shape (solid) for a helicity distribution (dotted), multiplied by a polynomial (dashed).

2.27 Multi-dimensional PDFs

Multi-dimensional PDFs can be constructed from the product of their individual components using an AFitProdPdf:

$$\mathcal{P}(\underline{x}) = PDF_1(x_1) \times PDF_2(x_2) \times \ldots \times PDF_N(x_N). \tag{34}$$

Multidimensional PDFs can be created using the AFitPdfFactory member function `makePdf(TString name, RooArgList &discVarList)`, where the first argument is the name of the PDF, and the second argument is a RooArgList of discriminating variables.

2.28 PDF summary

Table 1 summarises the different types of PDF available in AFit. This table shows the name of the function, the abbreviation of the function name that is used in the AFitPdfFactory, and the list of parameter names required by the pdf.

Table 1: The different types of PDF available in AFit. ‡denotes a conditional PDF that can be made using the PDF factory method `makeConditionalPdf`.

PDF	Pdf Factory Label	Variable Names
Argus	argus	endpt, xi, power
Breit-Wigner	breitwigner	m0, Gamma
Relativistic Breit-Wigner	relbreitwigner	mass, width, radius, spin, mass_a, mass_b
Bukin	bukin	Xp, sigp, xi, rho1, rho2
Chebychev Polynomial	chebyN	pi
Crystal Ball	cbshape	mean, width, alpha, n
Decay	decay	tau
BCPDecay	cpdecay	tau, dm, S, C, avgMistag, delMistag, mu
BDecay	bdecay	tau, dm, f0, f1, f2, f3, dgamma
Disappearance	disappearance	theta, amp, L, deltamsqr
Exponential	exponential	constant
Gaussian	gaussian	mean, width
Asymmetric Gaussian	agaussian	mean, widthL, widthR
Generic PDF	generic	...
Gounaris-Sakurai	gounarissakurai	mass, width, radius, spin, mass_a, mass_b
Helicity	helicity	...
Histogram	1dhist	...
KEYS	1dkeys	...
Landau	landau	mean, sigma
Novosibirsk	novosibirsk	peak, width, tail
Polynomial	polyN	pi
PSF	psf	_PSFcoef_N, _PSFlimit_N
Resolution‡	resolution	sigMean$i$, sigSig$i$, coreFrac, tailFrac scaleCoreMean, scaleCoreWidth, scaleTailMean, scaleTailWidth, useTruthModel
Sigmoid	sigmoid	a, b
Step/Veto	step	*limits*, *veto*
Voigtian	voigtian	mean, width, sigma, Algorithm
Composite Add PDF	add:x,y,...	
Composite Multiply PDF	multiply:x,y,...	
Multi-dimensional PDFs	–	–

3 Fit Components

A fit component is a multi-dimensional PDF defined for the set of variables $\underline{x}$. The types of fit component can be built are listed below and summarised in Table 2.

3.1 The default fit component

The default fit component is a product of one dimensional PDFs which can be expressed as:

$$PDF(\underline{x}) = PDF_1(x_1) \times PDF_2(x_2) \times \ldots \times PDF_N(x_N). \tag{35}$$

The configuration file excerpt for a default component looks like:

```
[signal]
signal_x_type = gaussian
signal_y_type = gaussian
signal_q_type = gaussian
```

where the term in square brackets is the component name. This is followed by a list of variables with the naming convention of the component name, followed by the discriminating variable name followed by `'_type'`. The value assigned to this variable has to be one of the PDF types listed in Table 1.

3.2 Scalar to Vector Vector decays (vvpolarisation)

The decay of a scalar particle to two vector particles results in the final state being a superposition of three amplitudes. One of these corresponds to the longitudinal polarisation, and the other two are transverse polarisations. In the vvpolarisation component model, the longitudinal and transverse polarisations are combined by the fraction of longitudinally polarised events f_L. In order to use the physical value of f_L in a fit, one has to specify the reconstruction efficiency for the longitudinal and transverse events. So the PDF is given by

$$\mathcal{P}(\underline{x}) = f_L^{eff}\mathcal{P}_L(\underline{x}) + (1 - f_L^{eff})\mathcal{P}_T(\underline{x}). \tag{36}$$

where $\mathcal{P}_L(\underline{x})$ is the PDF for the longitudinal polarisation, $\mathcal{P}_T(\underline{x})$ is the PDF for the transverse polarisation, and f_L^{eff} is the observed fractional difference between the two polarisations. The effective parameter f_L^{eff} is related to the physical parameter f_L via

$$f_L^{eff} = \frac{f_L}{(1 - f_L)\epsilon_T/\epsilon_L + f_L} \tag{37}$$

where ϵ_L and ϵ_T are the efficiencies for the longitudinal and transverse polarisations.

The configuration file excerpt for a vvpolarisation component looks like:

Table 2: The different types of fit component available.

Component Name	Description
default	A product of one dimensional PDFs in $\underline{x}$
vvpolarisation	PDFs for longitudinal and transfers components coupled by f_L
composite	a weighted sum of components

```
[vvsignal]
vvsignal_polarisationfrac = 1.0 +/- 0.1 L(0.0 -1.0)
vvsignal_effLong = 0.30 C
vvsignal_effTran = 0.40 C

[vvsignal_long]
vvsignal_long_x_type = gaussian
vvsignal_long_y_type = gaussian
vvsignal_long_q_type = landau

[vvsignal_tran]
vvsignal_tran_x_type = gaussian
vvsignal_tran_y_type = gaussian
vvsignal_tran_q_type = landau
```

where the term in square brackets is the component name. For the `vvsignal` component, the only parameters that need to be configured are the value of f_L which is given by `vvsignal_polarisationfrac`, and the efficiencies for the longitudinal and transverse polarisations (given by `vvsignal_effLong` and `vvsignal_effTran`, respectively) in this example. The longitudinal and transverse polarisations are identified with the other two component names. Each of these polarisations is modelled by a default PDF type as described above.

3.3 Composite component model (composite)

This component model is the sum of several indivudual components, added with a given weighting. The total pdf can be written in terms of each of the components $\mathcal{C}_i$ as:

$$PDF(\underline{x}) = (1 - f_1 - \ldots - f_n)\mathcal{C}_0(\underline{x}) + f_1\mathcal{C}_1(\underline{x}) + \ldots + f_n\mathcal{C}_n(\underline{x}). \quad (38)$$

where the f_i are the fractions of the components with i from 1 through to n. The fraction of the the zeroth component is given by one minus the sum of the f_i.

4 Building a Fit model

All applications in AFit start from a fit model that is constructed by the AFitMaster class. This class is responsible for reading a configuration file that specifies the lists of discriminating variables, and fit components, as well as the individual PDF types for each discriminating variable used in every fit component. Once this has been done, the fit model will be constructed. The rest of this section describes additional functionality provided by this class. Several examples are described in Section 6.

4.1 The *makePdf* function

The *makePdf* member function of AFitMaster is used to read in the configuration file specified in the constructor, and build a PDF according to the configuration.

4.2 The *makeSimPdf* function

The *makeSimPdf* member function of AFitMaster is used to read in the configuration file specified in the constructor, and build a PDF according to the configuration. The pdf constructed using *makePdf* is split according to rules specified for one or more RooCategory variables. Examples of PDFs made using this function include time-dependent CP asymmetry measurement PDFs where the PDF is split according to physics flavour tag categories. Section 6.4 describes an example of how to make a model that is split by several categories.

4.3 The *makeConditionalPdf* function

The *makeConditionalPdf* member function of AFitMaster is used to read in the configuration file specified in the constructor, and build a conditional PDF according to the configuration.

4.4 The *fitParameters* function

The *fitParameters* member function of AFitMaster, reads in reference data specified in the configuration file for a given fit component, and fits PDFs to the reference data for that fit component. The data files to use when fitting parameters are specified in the configuration file block `[PDF Param Files]`. The following example shows a signal data file being specified as an ascii file called signal.txt, with the discriminating variables as uncorrelated. The keyword 'data' in the comma separated list is the name of the data to be read (so this is a dummy variable for an ascii file).

```
[PDF Param Files]
signal_referencedata = signal.txt,ascii,data,uncorrelated
```

If discriminating variables in the fit are specified as uncorrelated in the configuration file, then the PDF parameters for each discriminating variable will be fitted separately for a fit component. Otherwise all parameters for all PDFs in a component will be fitted simultaneously.

When finished fitting, the a new configuration file will be written to the *out* stream specified in the *fitParameters* member function call. The following code snippet illustrates how to use this function.

```
AFitMaster master(‘‘mydatacard.txt’’);
RooAbsPdf * pdf = master.getPdf();
ofstream out (‘‘outputdatacard.txt’’);
master.fitParameters(out);
out.close();
```

Note: in order to use this function, the data card used in the AFitMaster constructor should fully specify the PDF, taking note of which shape parameters need to be varied and which are to be held constant. If you need to create a data card before fitting the data, you can use the writeDataCard function described in Section 4.6 to obtain a template for modification.

4.5 The *persist* function

This function will Write out the following

- A list of the discriminating variables used in the fit model.
- A list of the component types used in the fit model.
- A list of the component coefficients used in the fit model.
- A list of the options specified in the [FitConfiguration] block.
- The fit results obtained when fitting the PDF shapes (these will be present only if the *fitOptions* specified in the [FitConfiguration] block used contains 'r').

to the file specified by *ofileName*. If this file exists, it will be overwritten, otherwise a new file will be created. There is only one argument to this member function, and it is used as follows:

```
master.persist(‘‘myfile.root’’);
```

4.6 The *writeDataCard* function

When setting up a fit model there can be a very large number of fit parameters that need to be defined in the configuration file (or data card) that describes your fit. The *writeDataCard* function aims to simplify the process of writing a skeleton data card. If you are able to define the variables that go into your fit, as well as the fit components (signal and backgrounds), the component types, the yields[4] and the functional forms of the distributions used for the

[4]Note that each category yield can be blinded using an independent blinding string and scale factor.

component PDFs, then one can write out a new data card that has all of the aforementioned information included, in addition to the the PDF parameters that will need to be defined by the fit model.

For example if you consider the 2D fit model of the effective mass M_{ES} and energy difference ΔE of a B meson (See `examples/testAFitProjectionPlot.cc` and `examples/mes_de_model.txt`), then it is sufficient to specify the following

```
[FitConfiguration]
// specify the variables to use in the fit
variables = bMes,bDeltaE
// specify the names of the signal and background components
components = signal,continuum,Bbg0
fitOptions = etrmh

// set the limits and initial values of the variables used
bMes = 5.2700 +/- 0 L(5.25 - 5.29) B(30)
bDeltaE = 0.0000 +/- 0 L(-0.3 - 0.3) B(30)

// set the component types
signal = default
continuum = default
Bbg0 = default

// give initial values for the yields of each component
signalYield = 500.00 +/- 10.000 L(-100 - 10000)
continuumYield = 2000.00 +/- 10.000 L(-100 - 10000)
Bbg0Yield = 50.000 +/- 10.000 L(-100 - 10000)

// define the shapes used for the signal $M_{ES}$ and $\Delta E$ PDFs
[signal]
signal_bMes_type = gaussian
signal_bDeltaE_type = landau

// define the shapes used for background $M_{ES}$ and $\Delta E$ PDFs
[continuum]
continuum_bMes_type = argus
continuum_bDeltaE_type = poly2

// define the shapes used for background $M_{ES}$ and $\Delta E$ PDFs
[Bbg0]
Bbg0_bMes_type = argus
Bbg0_bDeltaE_type = poly2
```

With this information specified in a configuration file you can generate a configuration that includes PDF parameters by running the following commands:

```
AFitMaster master(‘‘my_model_configuration.txt’’);
RooAbsPdf * pdf = master.getPdf();
master.writeDataCard(cout);
```

where `cout` can be replaced by any ostream object for example a text file you've just opened.

4.7 The *fitData* function

This member function is used in order to fit the data to the total PDF model build from the AFitMaster. If the *fitOptions* variable in the configuration file's [*FitConfiguration*] block specifies 'r', this function will return a pointer to a RooFitResult that should be non-zero.

4.8 Blinding yields

The fit yields for components are by default unblind. However AFit is written so that any of the yields may be blinded using the appropriate syntax in the configuration file. Each yield has its own blinding string, blinding state and a scale factor used to compute the blinding offset. For example, if we consider the signal fit component, its yield has the name `signalYield`. The blinding parameters associated with `signalYield` are `signalYield_BlindingType`, `signalYield_BlindingString`, and `signalYield_BlindingScale`. The possible values of the `BlindingType` are blind and unblind (all lowercase). The blinding string is any user defined string, and the scale is used to tune the size of the random offset. The `RooUnblindOffset` class is used to implement blind yields in AFit.

4.9 Replacing Variables In A PDF

Often we want to construct a complicated PDF where several shapes have a common parameter, for example a kinematic endpoint in M_{ES} as described by an ARGUS PDF shape. The data are all constrained by the same endpoint, and so in principle should have the same `RooRealVar` parameter assigned as this common parameter for all components that are described by an ARGUS PDF in the fit. Normally this will not be the case: each ARGUS PDF will have its own endpoint.

It is possible to override this behaviour by substituting parameters when the PDF is being built. The order in which this is done is important - the PDF you take the new parameter from must have been built before the PDF you are currently trying to modify (so this has to appear earlier in the list of components, or variables). If this is satisfied, then you can assign a parameter substitution by adding the following line to the appropriate section describing your PDF in the data card:

```
mypdfname_variablesToReplace = oldVarA:newVarA,oldVarB:newVarB,...
```

where the comma separated list provides pairs of old variables that will be replaced by new ones.

4.10 Computing systematic uncertainties

The systematic uncertainties on a fit result from a constant parameter p in the likelihood model is given by the shift on the nominal fitted value when p is varied by $\pm\sigma(p)$. The `computeSystematicError` function within `AFitMaster` can be used to compute such an uncertatinty. When this function is called, three fits to the data are performed (i) the nominal fit (ii) a fit with p set to $p+\sigma(p)$, and (iii) a fit with p set to $p-\sigma(p)$. The shifts in all parameters allowed to vary in the fit are reported in a specified text file.

For example, using the rare B decay model one can compute the systematic uncertainty from a generated toy Monte Carlo simulated data sample using the following

```
AFitMaster master(``AFit/example/rareBdecay.txt'');
RooAbsPdf * pdf = master.getPdf();
RooArgSet &varsToGen = master.getDiscVarSet();
AFitToy toy;
RooDataSet * data = toy.generateToySample(pdf, varsToGen, 1000, 0);
master.computeSystematicError(data, ``continuum_bMesxi'', ``testSyst.txt'');
```

The output file from this test is the following:

```
BbgOYield = -76.775 +/- 305.881 systematic error shifts = +0 -0.251594
continuumYield = 915.797 +/- 171.32 systematic error shifts = +0 -0.192007
signalYield = 161.237 +/- 14.9639 systematic error shifts = +0.12581 -0.12946
```

from which one can read off the systematic uncertainty of interest from the last two columns of data. If the systematic shift is one-sided (as in the case of `BbgOYield` and `continuumYield` above) then the larger of the two shifts is taken as the systematic uncertainty, otherwise the asymmetric error is reported. The output file is written in such a way that it can easily be parsed by scripts.

5 Utilities

5.1 Statistical analysis

5.1.1 Pearson Correlation Coefficients

The Pearson correlation coefficient $\rho = \sigma_{x,y}/\sigma_x\sigma_y$ between variables in a TTree can be computed using the `correlation` member function of the `AFitStatTools` class. This function requires that the branches of the TTree are of type `Double_t`, and the correlation coefficients are calculated for all possible combinations of a comma separated list of variables. For example, given a pointer to the TTree `tree`, one can compute the correlations between the variables `mass` and `energy` using

```
AFitStatTools st;
st.correlation(tree, ‘‘mass,energy’’);
```

or

```
AFitStatTools st;
st.pearsons_correlation(tree, ‘‘mass,energy’’);
```

the output of this will look something like

```
Results from AFitStatTools::correlation for the variables
      mass,energy
Correlation matrix follows:
1            -0.0157139
-0.0157139            1
```

where the order of the columns and rows is the same as the order of variables in the comma separated list.

5.1.2 Spearmans Rank Correlation Coefficients

The matrix of Spearmans rank correlation coefficients r_s can be calculated in a similar way using

```
AFitStatTools st;
st.correlation(tree, ‘‘mass,energy’’, kTRUE);
```

or

```
AFitStatTools st;
st.spearmans_rank_correlation(tree, ‘‘mass,energy’’);
```

where the correlation r_s is given by

$$r_s = 1 - \frac{6 \sum_i d_i^2}{n(n^2 - 1)} \tag{39}$$

where n is the number of data points, the d_i are difference between the integer ranks of the i^{th} event for the two variables x and y and this quantity is summed over all events.

5.1.3 Miscellaneous

The `AFitStatTools` class has member functions that perform a number of different calculations, including

- Binomial error
- χ^2 sum
- χ^2 probability

5.2 Projection Plots

The class AFitProjectionPlot can be used in order to make projections of a PDF on a RooPlot. If a RooDataSet is provided, the data will be plotted on the RooPlot as well as the projection over the PDF. This class can also be used in order to compute a pull plot from the difference between data points and the PDF curve.

The following is an example of using this plot class to make a plot of a Gaussian PDF for the variable `x` where a generated data set of 1000 events is plotted on the RooPlot in addition to the pdf. Figure 24 shows the result of running this macro.

```
RooRealVar x(‘‘x’’, ‘‘’’, 0.0, -5, 5);
AFitPdfFactory fact;
AFitAbsPdfBuilder * bld = (AFitAbsPdfBuilder *)fact.makePdf(‘‘pdf’’,
                                                  ‘‘gaussian’’, x);
RooAbsPdf * pdf = (RooAbsPdf*)bld->getPdf();

AFitProjectionPlot plotter;

RooDataSet * data = pdf->generate(RooArgSet(x), 1000);
RooPlot * frame = plotter.makePlot(x, data, pdf);
frame->Draw();
```

Section 6.3 describes an example that uses a conditional variable in the PDF. It is necessary to provide prototype data sets for all conditional variables when projecting such a PDF.

5.2.1 Example: Enhancing the signal for a 2D fit model

If one has a more complicated model for example a 2D fit to a sample of data, for example the effective mass M_{ES} and energy difference ΔE of a B meson, it is possible to enhance the signal by cutting on variables not projected in the fit to data. An example of this would be the following

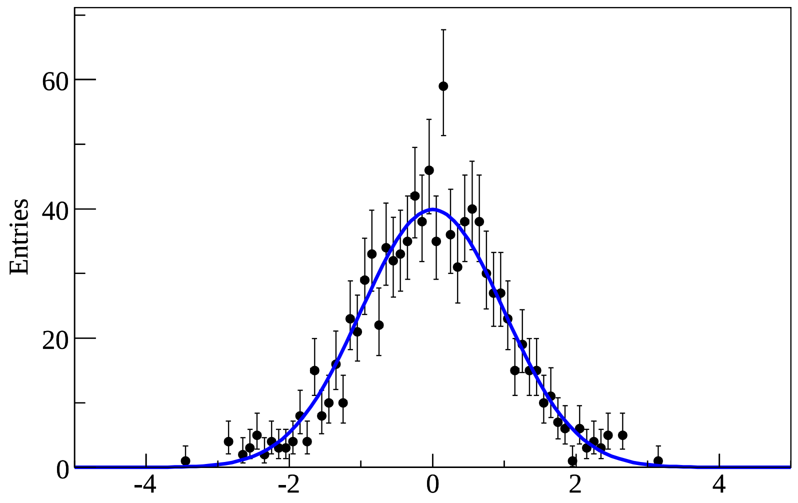

Figure 24: An example of using the AFitProjectionPlot class to plot a Gaussian PDF and generated data set.

s

```
RooPlot * frame = plotter.makePlot(‘‘mes>5.27’’, *DeltaE, data, pdf);
```

where the RooDataSet passed to the makePlot function will automatically have the cut applied to it in order to produce the plot with a correct normalization. One can see the effect of making such a cut by comparing the distribution of data before and after applying it. This particular example is available in `examples/testAFitProjectionPlot.cc` and Figure 25 shows the distribution of generated data before and after applying the cut of $M_{ES} > 5.27$ to it.

5.2.2 Example: Plotting an asymmetry

There is an interface method for plotting asymmetries on data using the `AFitProjectionPlot` class. For example, one can make an asymmetry plot as a function of Δt `deltat` for a time-dependent CP analysis where there is a flavour tag variable `tag`, a data sample `data`, a prototype with conditional variables `proto` that includes the flavour tag using the function call

```
RooPlot * frame = plotter.makeAsymmetryPlot(deltat, tag, data, pdf, proto);
```

where the end result should look similar to Figure 26.

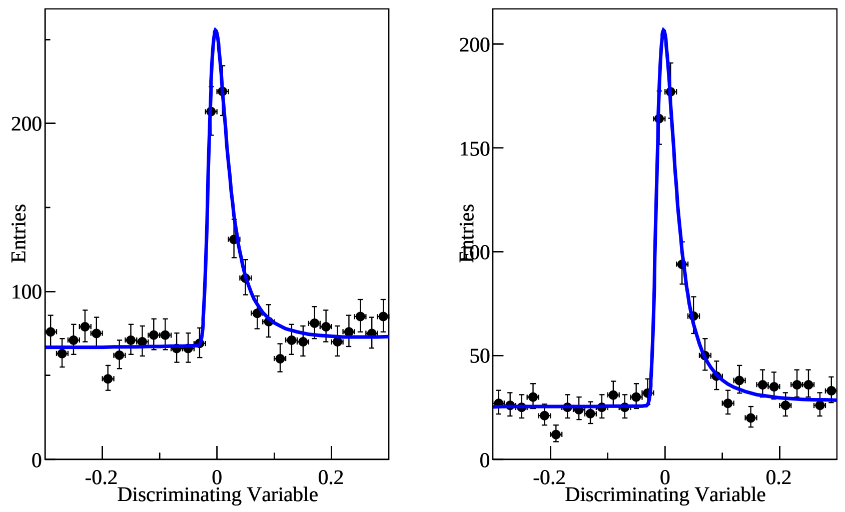

Figure 25: An example of using the AFitProjectionPlot to make signal enhanced projections: (left) before and (right) after cutting on M_{ES}.

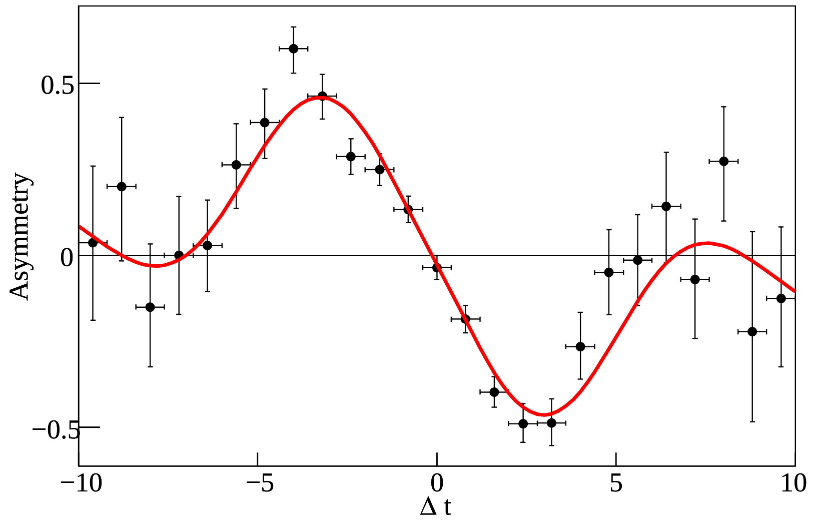

Figure 26: An example of using the AFitProjectionPlot to make a Δt asymmetry plot.

5.2.3 Pull Plots

Given a RooPlot that has both a data set and a curve plotted on it, you can produce a pull plot, that is a plot of the difference between the curve and the data points, normalized to the error on the data by doing the following

```
RooHist * datahist = frame.getHist(``TotalModelData_plot__bMes'');
RooCurve * curve = frame.getCurve(``TotalModelProjected'');
RooHist * PullPlot = plotter.makePullPlot(datahist, curve);
PullPlot.Draw(``A*'');
```

which will result in a pull plot like Figure 27 being draw.

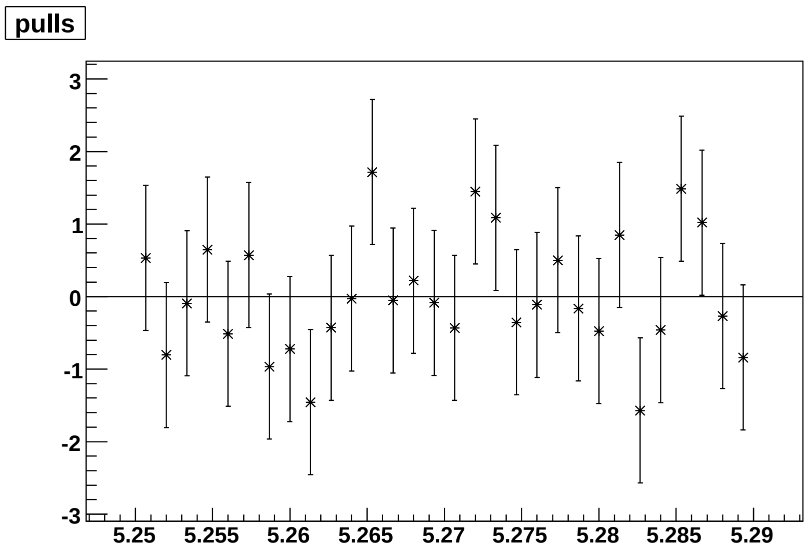

Figure 27: An example of using the AFitProjectionPlot class to make a pull plot.

5.2.4 Likelihood Projection Plots

One way to enhance the signal content of a data sample when making a projection, is to cut on a likelihood ratio of the projected variables (this ratio does not use information from the plotted variable). The likelihood ratio computed in `AFitProjectionPlot` is the $\log_e$ of the ratio of signal to total likelihoods. The following code snippet illustrates how to make a liklihood ratio projection plot:

```
AFitProjectionPlot plotter;
RooPlot * llrframe = plotter.makeLRProjectionFrame(pdf, data, varToPlot, sigCompName);
RooPlot * projframe = plotter.makeLRProjection(pdf, data, varToPlot, sigCompName, cutVal);
```

where `pdf` is the total PDF, data is the data set to use when making the projection, varToPlot is a TString whose value is the variable to plot, and sigCompName is the component 'signal' to enhance. The argument cutVal is the minimum value of the likelihood ratio for projected events. Figure 28 shows the likelihood ratio distribution, and the corresponding before/after signal enhancement projections of the data. Both the total and signal PDFs are plotted on the `RooPlot` of the projection made using the `makeLRProjection` function.

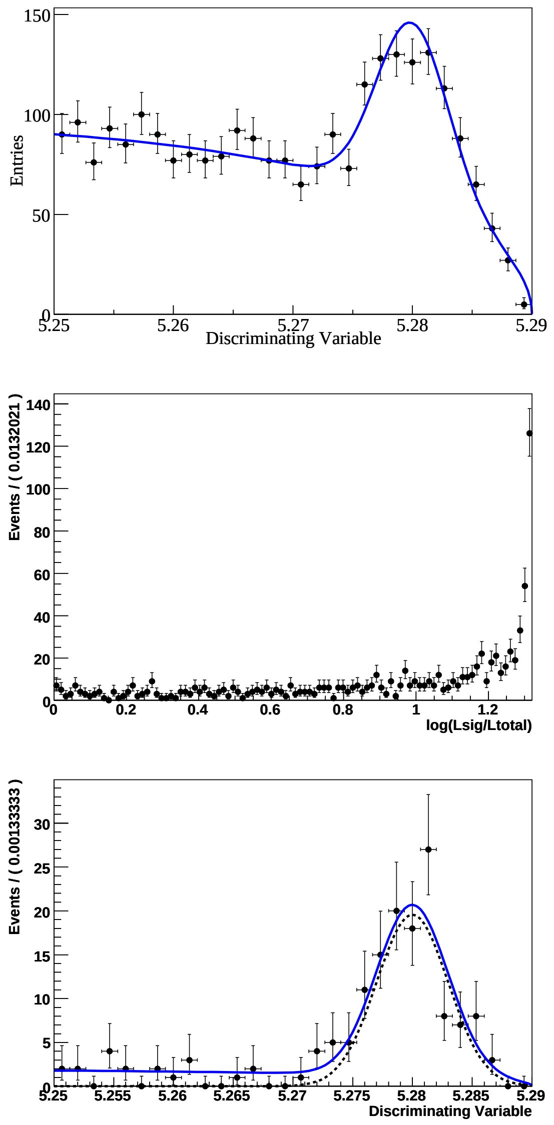

Figure 28: The distribution of (top) the projection of m_{ES}, (middle) likelihood ratio (bottom) the projection after cutting on the likelihood ratio to enhance signal.

5.3 Likelihood Ratio Plots

A test that can be made between the a complicated PDF and a reference data sample is to compute the ratio of the likelihood of an event to be signal $\mathcal{L}_{sig}$ normalised to the total likelihood of an event to either signal or background $\mathcal{L}_{sig} + \mathcal{L}_{bg}$. The class AFitLRPlot computes this ratio for a sample of simulated data generated from the PDF (toy data) and compares this to a reference sample of data.

The likelihood for this example is one dimensional: the discriminating variable is the B meson mass. The signal PDF is a Gaussian distribution centred at 5.28 GeV/c^2 with a width of 10 MeV/c^2, and the background distribution is described by an Argus function (see below). Figure 29 shows the likelihood ratio

$$R = \frac{\mathcal{L}_{sig}}{\mathcal{L}_{sig} + \mathcal{L}_{bg}}, \tag{40}$$

for Monte Carlo data (points) and the shaded regions correspond to the simulated data generated from the PDF. If the PDF describes the data properly, the data and toy data distributions will agree and the signal should peak near $R = 1$, and the background should peak near $R = 0$. The Figure clearly shows a background component that peaks near $R = 1$ which indicates that there is no way to distinguish between the signal and part of the background using that particular fit configuration.

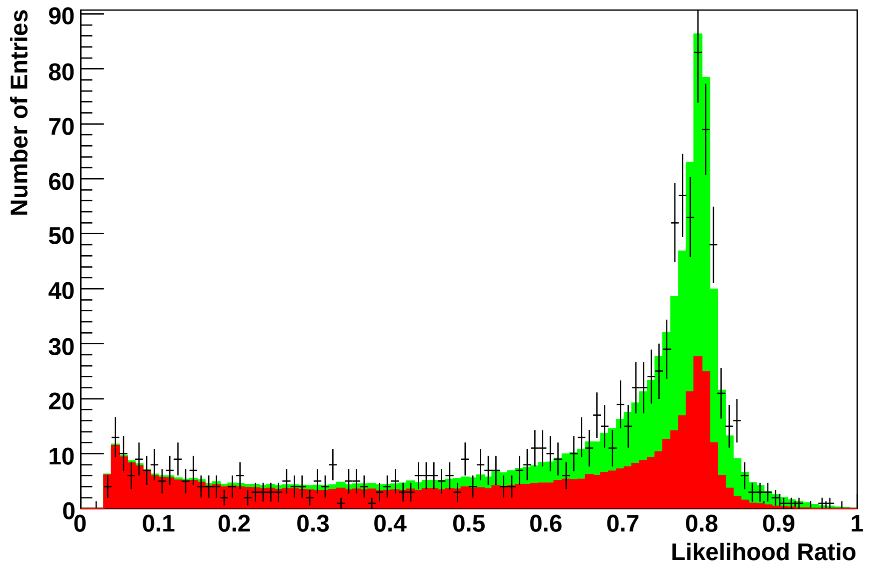

Figure 29: The likelihood ratio R for a sample of Monte Carlo compared to a PDF. The signal component of the PDF is shown in green and the background component is shown in red.

The code required to make the plot of Figure 29 available in *example/likelihoddRatio.cc* and is shown here::

```
// the discriminating variable x is the beam constrained B meson mass.
RooRealVar x(‘‘x’’, ‘‘mass (GeV)’’, 5.25, 5.29);

// The background PDF
AFitArgus argus(x, ‘‘arguspdf’’);
RooAbsPdf * argusPDF = argus.getPdf();

// The signal PDF
AFitGaussian gauss(x, ‘‘gausspdf’’);
gauss.setParameter(‘‘mean’’, 5.28);
gauss.setParameter(‘‘width’’, 0.01);
RooAbsPdf * gaussPDF = gauss.getPdf();

// set the fraction of signal at 50%, so there is 90% background
// and generate 1000 toy events from the composite PDF.
RooRealVar sigfrac(‘‘sigfrac’’, ‘‘fraction’’, 0.5);
RooAddPdf pdf(‘‘pdf’’, ‘‘’’, RooArgList(*gaussPDF, *argusPDF), sigfrac);
Int_t nToGen = 1000;
RooDataSet *data = (RooDataSet*)pdf.generate(RooArgSet(x), nToGen);

Double_t nSig = sigfrac.getVal()*nToGen;
Double_t nBG = nToGen-nSig;

// compute the likelihood ratio
AFitLRPlot lrplot;
lrplot.makePlot(gaussPDF, argusPDF, new RooArgSet(x), data, nSig, nBG);
```

5.4 TMVA Interface

The class AFitTMVAInterface is a utility to facilitate the computation of an MVA within an analysis framework. See Ref. [11] for more information on the TMVA package. The TMVA training options are steered from a text file (or the same PDF data card that you use for your fit). The text file block is denoted by `[TMVAInterface]`. The following is an example configuration used for computing a Fisher and training a Boosted Decision Tree with TMVA:

```
[TMVAInterface]
sigFile = tmva_fsig.root
bgFile = tmva_fbg.root
dataName = data
outputFileName = tmva_out.root
trainingMethod = Fisher,BDT
variables = a:F,b:F
```

The signal and background files are specified by assigning values to `sigFile` and `bgFile`. It is assumed that the input data come from a RooDataSet with a name given by the value of `dataName`. The output file has the default value of `tmva_out.root`. More than one training method can be booked for the variables going into the MVA. This is done by assigning the appropriate method to a comma-separated list of methods. The variables to train are specified in a comma separated list with pairs of data: `variable_name:Type`. TMVA recognises types of `I` and `F` to distinguish between different integer variables (int, long int etc.), and floating point variables (float, double). Once finished training MVAs, TMVA writes output to a sub-directory called `weights`.

The known MVA types are: Cuts, Likelihood, HMatrix, Fisher, CFMlpANN, TMlpANN, BDT, RuleFit, SVM, MLP. See the TMVA User Guide for more information on these [11]. Other options that are recognised by AFit are described below:

- factoryOptions – This variable can be set to any of the options passed to the TMVA factory on instantiation.
- trainingOptions – This variable can be set to define the options used when calling `factory.PrepareTraining`
- preselectionCut – This variable can be set to define the preselection cut used when calling `factory.PrepareTrainingAndTestTree`.
- analysisprefix – This is the name of the TMVA::Factory instance used (also the weight file prefix name).

Once the training configuration file has been prepared, this can be used by typing the following:

```
AFitTMVAInterface a(‘‘myConfigurationFile.txt’’);
a.trainMethods();
```

TMVA will run the specified classifiers with the specified variables found in the source files. Once you've inspected the output of TMVA, and decided which classifier(s) you want to consider using in your fit, you can compute classifiers for RooDataSets using the `AFitTMVAInterface::runReader` member function. This takes a RooDataSet as an argument, and will add columns to the data set for each classifier specified in the configuration file. The new RooDataSet is passed back to the user for storage. In the case that there are several files that you wish to process, you can use the overloaded function of the same name that takes two string arguments (the first string is the initial file, and the second string is the target file).

5.5 Toy Monte Carlo Validation of the likelihood

The `AFitToy` class has been implemented in order to simplify the process of running toy Monte Carlo validations of the likelihood function. The process of running a toy Monte Carlo study involves

1. Setting the random number seed for use in generation, this ensures that one is able to reproduce exactly the same sample of data using the same sequence of random numbers each time. One will have to use different seeds for different toy Monte Carlo samples.

2. Determining the number of events to generate. If one analyses an ensemble of toys, it is necessary to generate a mean number of events according to a Poissonian distribution.

3. Generating the simulated data sample: there are several ways of doing this depending on the type of toy to be performed, and the preferred context.

4. Fitting the simulated data sample.

5. Persisting the results of the fit for further analysis.

Given a pdf and a RooArgSet of discriminating variables the `AFitToy` class can be used to generate toy Monte Carlo samples of events as follows:

```
AFitMaster master(‘‘AFit/example/MuonLifetime.txt’’);
RooAbsPdf * pdf = master.getPdf();

// Get the time variable from the AFitMaster.
RooArgSet * compSet = pdf->getComponents();
RooArgSet * parSet = pdf->getParameters(compSet);
RooRealVar * t = (RooRealVar*)parSet->find(‘‘t’’);

// make the interface used to run toys
AFitToy toy;
for(int i=0; i < 10; i++){
 // generate the toy data
  toy.setSeed(i);
  RooDataSet * data = toy.generateToySample(pdf, RooArgSet(*t), 1000, 0);

 // fit the toy data
  RooFitResult * r = pdf.fitTo(data, ‘‘etrm’’);
}
```

where the muon lifetime fit example has been used for this toy (see Section 6.1). The arguments to `generateToySample` are a pointer to the RooAbsPdf to fit, the set of variables to generate, the number of events to generate, and the prototype to use for any conditional variables that you want to generate. The total number of events generated in each toy sample will have a Poisson mean of 1000. This default behaviour can be switched off by calling the `toy.setPoisson(kFALSE);` before generating. If one does this, then each toy will have exactly 100 events generated.

There are more sophisticated toy generation functions to call. The highest level one is illustrated in the following muon lifetime fit example:

```
AFitToy toy;
toy.setOutputDir(``toy/'');
toy.generateToys(``macros/MuonLifetime.txt'', ``fpr'', 1, 100, 0);
```

The AFitToy instance uses the input configuration file to build the likelihood fit model. The second argument "fpr" determines what is done with the generated data (described below). Using the likelihood, toys with initial random number seeds corresponding to the toy number (1 through 100, the third and fourth arguments to the generateToys member function). The final argument is a prototype that can be used when generating toy samples for any likelihoods that depend on conditional variables.

The options "fpr" determine the following

- f – Fit the generated data.
- p – Persist the generated data in a root file. The file name is *iSeed_toydata.root*, and the data are saved as a RooDataSet (*data*).
- r – Persist the results in a root file. The file name is *iSeed_toyresults.root*, and the RooFitResult ($fitResult$) is saved to the file, along with a TTree (*resultdata*) that contains the fit result information.

In order to inspect the results of an ensemble of toy MC experiments, it is easy to chain together the *resultdata* TTrees obtained using the following

```
TChain chain(``resultdata'');
toy.setOutputDir(``toy/'');
toy.chainResults(chain, 1, 100);
```

If you want to make sure that fits converged ok with $status = 0$, then the last line should be replaced by `toy.chainResults(chain, 1, 100, kTRUE);`. The `chainResults` member function checks to see if a file exists before trying to add it to a chain, and any zombie files are skipped automatically.

5.6 Toy Monte Carlo Validation: Embedded Toys

The previous section summarised tools used to run Toy Monte Carlo validation studies where the likelihood is used to generate an ensemble of data samples to fit back. Any deviation from the input results would be a result of the intrinsic bias of the fit. In defining the likelihood, we make assumptions about many things including the correlations between discriminating variables in the fit. These assumptions can be tested by embedding simulated data samples obtained from the Full Monte Carlo simulation for an experiment. The AFitToy utility has several member functions to facilitate performing such a toy. These are `generateEmbeddedToySample`, and `generateEmbeddedToys`. Given a large data sample of events from a Full Monte Carlo simulation, one can create a number of subsamples with a predetermined number of events using the following:

```
toy.generateEmbeddedToys(sourcedata, 1000, 1, 10);
```

where this example used 1000 events per sample, and generates 10 samples with numbers 1 through 10. The output data files are written to the directory specified by `setOutputDir` and have file names that are `<iToy>_embtoydata.root`. Once several files have been generated (for example signal and background), they can be merged together using the `mergeFiles`[5] member function.

s

6 Examples

All example macros and configurations can be found in the `AFit/example` directory. These examples are ready to run from the same directory that contains the AFit package. This assumes that once you have compiled AFit, that you then load the shared library into root prior to running these examples.

6.1 Fitting the muon lifetime

The muon lifetime can be fitted from data taken with a simple experiment where slow moving muons from cosmic rays are trapped in a scintillator, and subsequently decay [12]. This process results in to light pulses detected by a photomultiplier tube, and the different in time t between the two pulses is recorded. The start time of the clock is the time when the muon enters the detector, and the stop time is that when the muon decays into an electron and two neutrinos. In addition to this signal process, there is a background which is assumed to be uniform in t. So the PDF for this problem is given by

$$\mathcal{P} = N_{\mathrm{signal}} e^{-t/\tau_\mu} + N_{\mathrm{background}}. \tag{41}$$

This is a one dimensional problem with t being the sole discriminating variable. There are two components: signal and background. The configuration file for this example is `MuonLifetime.txt`. So the signal component is described by an exponential function, using a lifetime rather than a constant as the parameter to be determined from data, and the background component is described by a polynomial of order one with a coefficient of zero. The configuration file for this example can be found in `MuonLifetime.txt`.

[5]This member function takes a pointer to a TObjArray as an argument for the list of input files to merge. This TObjArray can be made from a comma separated list of files in a TString using `TString::Tokenize(',')`.

If you run the `MuonLifetime.cc` example on the sample of data provided, you will obtain the result $\tau_\mu = 2.09 \pm 0.02\ \mu s$ which accounts for the interaction of μ^- in matter. The distribution of the data and the fitted PDF are shown in Figure 30.

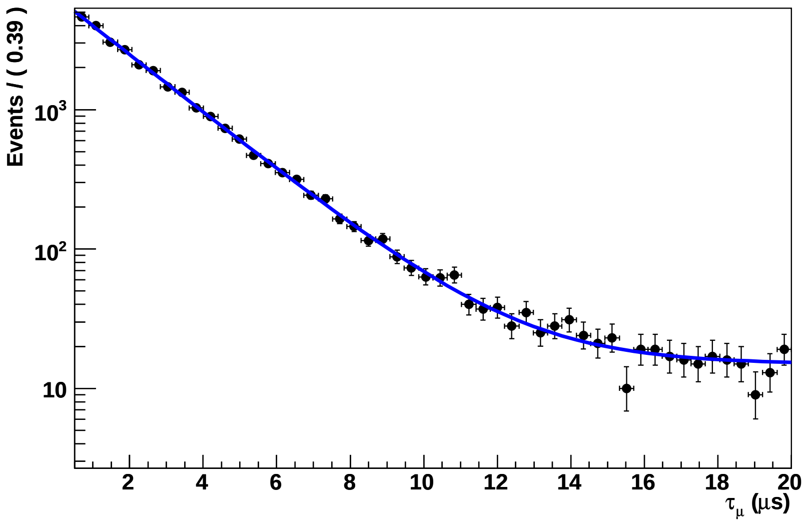

Figure 30: Fitting the μ lifetime using the `MuonLifetime.cc` example.

The fit configuration of this example can be found in `examples/MuonLifetime.txt`. The first line after `[FitConfiguration]` specifies the variable name `t` for the time difference between the two signals from the photomultiplier tube.

```
[FitConfiguration]
variables = t
```

Each fit component (these are `single`, `background` listed in the configuration file option `components` needs to have its form defined (see `signal` etc listed below. The names are defined by the comma separated values in the `components` list.

```
components = signal,background
```

The fit range and number of bins used when plotting the discriminating variables is also defined.

```
t = 0.5 +/- 0.01 L(0.5 - 20.0) B(50)
```

The type of each fit component is specified, see Section 3 for a list of possible fit component options.

```
signal = default
background = default
```

The final part of the `FitConfiguration` block specifies the fit yields, and allowed ranges for each of the fit components.

```
signalYield = 30000 +/- 10.000 L(-100 - 1e6)
backgroundYield = 100 +/- 100.000 L(-1000 - 1e4)
```

Each fit component specified has to have the functional form of the PDFs defined. The available options are listed in Table 1. For example the `signal` PDF for this example has an exponential function for the t distribution,

```
[signal]
signal_t_type = exp
```

The parameters of the exponential function are set by the following

```
[signal_t]
signal_t_constant = 2.2 +/- 0.1 L(-4.0 - 4.0)
signal_t_fitLifetime = true
```

where the block name is derived from the component name and variable name (this is just $<$ *component name* $>$ _ $<$ *variable name* $>$), and the PDF parameter names are prefixed by the same string. In this particular example the option `signal_tfitLifetime` is set to `true`, so that the functional form of the exponential is $e^{-t/constant}$ where *constant* corresponds to the lifetime (see section 2.9 for details). The `background` PDF is defined in an analogous way.

6.2 Simple rare B decay search at BaBar or Belle

Two kinematic variables can be used to select signal events in an $e^+e^- \to \Upsilon(4S) \to B\overline{B}$ event. These are m_{ES} and ΔE: $m_{ES} = \sqrt{(s/2 + \mathbf{p}_i \cdot \mathbf{p}_B)^2/E_i^2 - \mathbf{p}_B^2}$ is the beam-energy substituted mass and $\Delta E = E_B^* - \sqrt{s}/2$ is the difference betgween the B candidate energy and the beam energy in the e^+e^- CM frame. Here the $B_{\rm rec}$ momentum $\mathbf{p}_B$ and four-momentum of the initial state $(E_i, \mathbf{p}_i)$ are defined in the laboratory frame, and E_B^* is the $B_{\rm rec}$ energy in the e^+e^- CM frame. The distribution of m_{ES} (ΔE) peaks at the B mass (near zero) for signal events and does not peak for background.

We can simultaneously fit the m_{ES} and ΔE to isolate our signal. In order to do this we first need to decide how many fit components there are, secondly we need to determine what the functional forms of the PDFs we will use to describe these components. The example `examples/rareBdecay.cc` assumes that there is a signal component as well as a background from B decays and a background from $e^+e^- \rightarrow q\overline{q}$ events, where $q = u, d, s, c$.

The fit configuration of this example can be found in `examples/rareBdecay.txt`. The first line after `[FitConfiguration]` specifies the variable names `bMes` (m_{ES}) and `bDeltaE` (ΔE),

```
[FitConfiguration]
variables = bMes,bDeltaE
```

Each fit component (these are `single`, `continuum`, and `Bbg0`) listed in the configuration file option `components` needs to have its form defined (see `signal` etc listed below. The names are defined by the comma separated values in the `components` list.

```
components = signal,continuum,Bbg0
```

The fit range and number of bins used when plotting the discriminating variables is also defined.

```
bMes = 5.2700 +/- 0 L(5.25 - 5.29) B(30)
bDeltaE = 0.0000 +/- 0 L(-0.3 - 0.3) B(30)
```

The type of each fit component is specified, see Section 3 for a list of possible fit component options.

```
signal = default
continuum = default
Bbg0 = default
```

The final part of the `FitConfiguration` block specifies the fit yields, and allowed ranges for each of the fit components.

```
signalYield = 500.00 +/- 10.000 L(-100 - 10000)
continuumYield = 2000.00 +/- 10.000 L(-100 - 10000)
Bbg0Yield = 50.000 +/- 10.000 L(-100 - 10000)
```

Each fit component specified has to have the functional form of the PDFs defined. The available options are listed in Table 1. For example the `signal` PDF for this example has a Gaussian for the m_{ES} distribution and a Landau function for the ΔE.

```
[signal]
signal_bMes_type = gaussian
signal_bDeltaE_type = landau
```

where each PDF type is chosen by specifying a value to the label given by < *component name* > _ < *variable name* > _*type*. The remainder of the configuration file specifies the functional form of the other fit components, and the PDF parameters for each PDF of each variable for the fit components.

The AFitMaster class us used in order to construct this fit model at the start of `examples/rareBdecay.cc`:

```
AFitMaster master(``AFit/example/rareBdecay.txt'');
RooAbsPdf * pdf = master.getPdf();
```

The remainder of the example macro uses this PDF to generate a sample of simulated data using the `AFitToy` class (See section 5.5) and to plot the pdf and simulated data using the `AFitProjectionPlot` class (See section 5.2).

6.3 Fitting a Δt resolution function

The `Resolution` PDF described in Section 2 is used to model the resolution on the proper time difference Δt between the decays of two neutral B mesons in an event at the B-Factories. The resolution function parameters are scaled (multiplied) by the error on $\sigma(\Delta t)$ on an event-by-event basis. In order to use this PDF you need to define the discriminating variable, and if appropriate the corresponding conditional variable $\sigma(\Delta t)$. Having done this you can instantiate the `AFitResolution` class:

```
RooRealVar deltat("reso_dt", "dt", -10.0, 10.0);
RooRealVar deltatErr("deltaterr", "sdt", 1.2, 0.0, 2.50);
AFitResolution resoBld(&deltat, &deltatErr, "SigReso");
```

As this example requires that the mean and core parameters are scaled by $\sigma(\Delta t)$, we must set the following

```
resoBld.setParameter(``scaleCoreMean'',"yes");
resoBld.setParameter(``scaleTailMean'',"yes");
resoBld.setParameter(``scaleCoreWidth'',"yes");
resoBld.setParameter(``scaleTailWidth'',"yes");
```

Having configured the PDF, it is possible to now make it using

```
RooAbsPdf * pdf = resoBld.getPdf();
```

All of these steps can be replaced by a configuration file with the appropriate settings, and the use of the `AFitMaster` member function `makeConditionalPdf`. The `makeConditionalPdf` function takes a discriminating variable, a conditional variable, a PDF type and a PDF name as arguments. Once you have built your resolution model, it is possible to use it - as the PDF uses a conditional variable, you need to construct a prototype. For example, if you have a TTree called tree, you can construct a RooDataSet, and $\sigma(\Delta t)$ prototype with the following

```
RooDataSet data("rds", "the data", tree , RooArgSet(deltat, deltatErr));
RooDataSet * proto = data->reduce(RooArgSet(deltatErr));
```

and subsequently fit the data, and make a plot of the data

```
pdf->fitTo(data, "trh");

AFitProjectionPlot plotter;
RooPlot * frame = plotter.makePlot(deltat, &data, pdf, proto);
```

where you note that the prototype is required for plotting. An example macro demonstrating the use of the resolution function PDF with a test data sample can be found in the examples directory as `resolutionFunction.cc` and `resolutionFunctionData.root`.

6.4 RooSimultaneous: splitting a PDF by categories

The `AFitMaster::makeSimPdf` member function allows a user to construct a `RooAbsPdf` that is subsequently split by one or more `RooCategories` according to a specified rule. The example described below can be found in the files `simpdf.cc` and `simpdf.txt`.

Before defining how to split a PDF up according to categories, the fit model needs to be defined in the normal way. This example uses two two discriminating variables (m_{ES} and ΔE) and two fit components: signal, and background. The signal component has a Gaussian PDF for each of the discriminating variables, whereas the background m_{ES} distribution is described by an Argus PDF, and the background ΔE distribution is described by a polynomial (similar to the example in Section 6.2).

In order to specify how the PDF is modified according to different categories, these categories and splitting rules have to be specified in the `[FitConfiguration]` data card block. For this example, the pdf will be split according to signal decay type (the `decay` category) and by the flavour tag (the `tagcat` category). The following snipped of the example data card defines how the PDF is modified:

```
catVars = tagcat,decay
splitRule = tagcat :  signal_bMes_mean
tagcat_categories = Lepton,Kaon1,Kaon2,KaonPion,Pion,Other,NoTag
decay_categories = JpsiK0S,JpsiK0S_pi0pi0,JPsiK*0,Psi2SK0S,Chic1K0S
decay_splitrule = decay :  signal_bDeltaE_width
```

The `catVars` variable is followed by a comma separated list of category variables that the PDF will be split by. The individual categories are themselves defined by the ⟨*category name*⟩*_categories* variables. The individual category labels are numerically numbered $0, 1, 2, \ldots$ for each of the category variables. The splitting rule that defines what PDF parameters are split by what category needs to be defined. This rule can either be defined using the `splitRule` variable of the data card, or by the individual ⟨*category name*⟩*_splitrule* variables for each of the categories. The format of the split rules is the category name, followed by a colon ':', and then a comma separated list of PDF parameters that should be split by this category. Note that it is not possible to split a parameter by more than one category. In this example, the tagcat splitting rule has been specified using `splitRule`, and the decay splitting rule was specified using `decay_splitrule`.

Once the data card has been properly defined, the simultaneous pdf is built using

```
AFitMaster master(``macros/simpdf.txt'');
RooAbsPdf * simpdf = master.getSimPdf();
```

The second example of using a RooSimultaneous to split a PDF by category, is based on the previous discussion: see `simpdf2.cc` and `simpdf2.txt`. This is a signal plus background model as above, where the PDF is split by decay mode. There are four different signal decays under consideration: $J/\psi K^0_S$, $J/\psi K^0_S(K^0_S \to \pi^0\pi^0)$, $J/\psi K^{*0}$, $\psi(2S)K^0_S$. The parameters that are split by the decay category are signal yield, background yield and the ΔE width. Figure 31 shows the m_{ES} and ΔE distributions of four of the five decay channels based using a simulated data sample.

In order to generate the simulated data, a prototype data set containing the decay category is constructed. The total number of entries in the data set of each decay type corresponds to the sum of signal and background. Having prepared the prototype data set, the AFitToy utility (see Section 5.5) is used to generate the data sample to fit and plot. The projections are made using the AFitProjection utility (see Section 5.2).

6.5 Time-dependent CP asymmetry fit

In order to fit for time-dependent CP asymmetries in $\Upsilon(4S) \to B^0\overline{B}^0$ decays one can use the `AFitBCPGenDecay` class. Before instantiating this class, a resolution function needs to be set up (see Sec. 6.3). The example `cpfit.cc` illustrates how to set up a time-dependent fit for a signal decay like $B^0 \to J/\Psi K^0_S$. Figure 32 shows the resulting distributions of Δt for B^0, and $\overline{B}^0$ tagged events, as well as the time-dependent CP asymmetry.

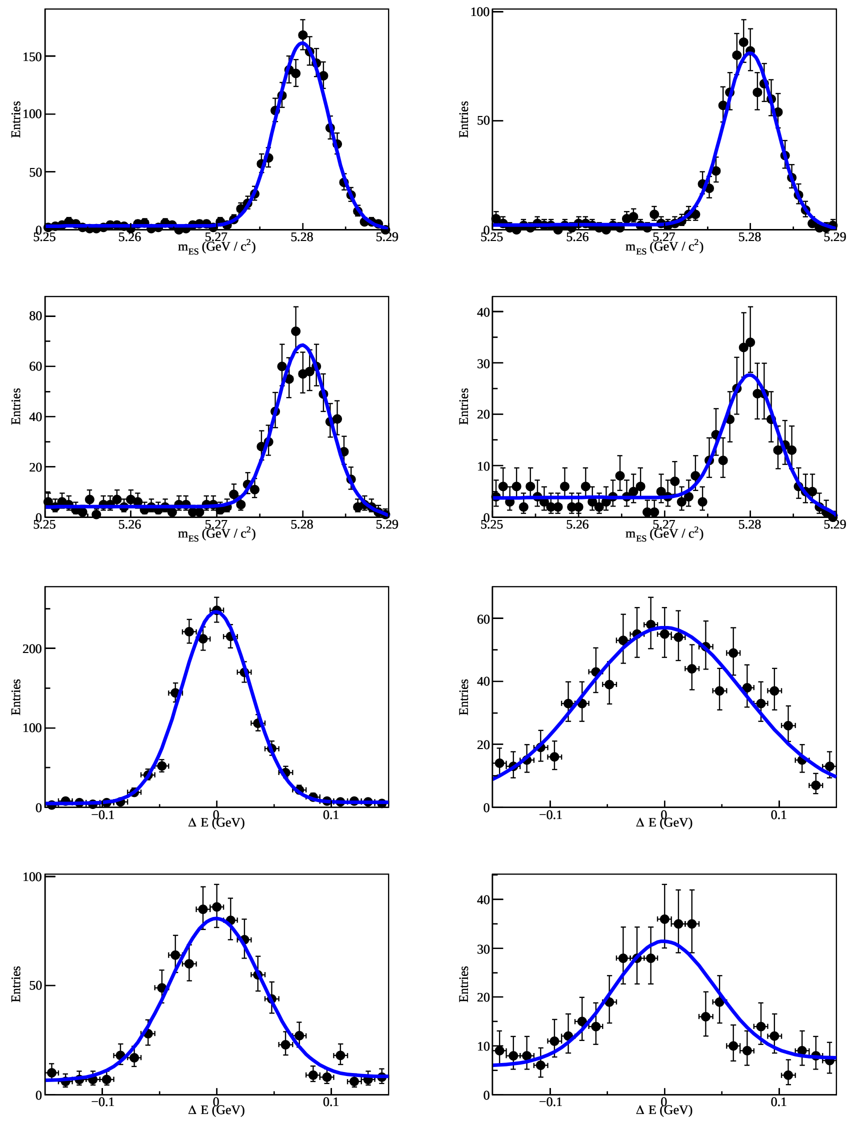

Figure 31: The (top four) m_{ES} and (bottom four) ΔE distributions for the four decay channels used in the example. The decay channels are (left to right top to bottom) $J/\psi K^0_S$, $J/\psi K^0_S(K^0_S \to \pi^0\pi^0)$, $J/\psi K^{*0}$, $\psi(2S)K^0_S$.

One final finesse that is required for a realistic time-dependent CP asymmetry fit is to split the final PDF by flavour tagging category, and to ensure that all appropriate parameters are split accordingly: ω, $\Delta\omega$, μ, and any resolution function parameters that are different for different flavour tagging categories (see Sec. 6.4).

The example `cpfit_tagging.cc` and associated configuration file illustrate how to extend the simple macro described above in order to split by tagging category. The parameters that are

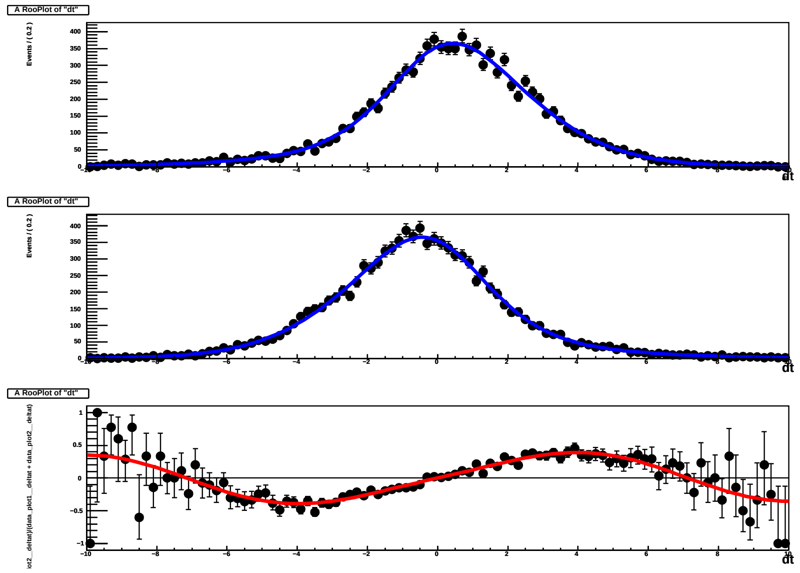

Figure 32: The Δt distribution for (top) B^0, and (middle) $\overline{B}^0$ tagged events, as well as (bottom) the time-dependent CP asymmetry.

split by the tagging category are a tagging efficiency associated with the signal yield which is called `signal_tageff`, as well as the mistag parameters ω and $\Delta\omega$.

7 Acknowledgements

The following people have contributed to the development of this package: Fergus Wilson (Rutherford Appleton Laboratory).

References

[1] The RooFit web page is `http://roofit.sourceforge.net/`.

[2] The MINUIT user guide can be obtained from `http://wwwasdoc.web.cern.ch/wwwasdoc/minuit/minmain.h`

[3] The ROOT web page is `root.cern.ch`.

[4] "Statistics: A Guide to the Use of Statistical Methods in the Physical Sciences", R. Barlow, John Wiley & Sons Ltd (1989).

[5] "Statistical Data Analysis", G. Cowan, OUP (1998).

[6] H. Albrecht *et al.*(ARGUS Collaboration) Phys Lett B241 (1990) 278.

[7] M. J. Oreglia, Ph.D Thesis, SLAC-236, Appendix D, (1980); J. E. Gaiser, Ph.D Thesis, SLAC-255, Appendix F, (1982); T. Skwarnicki, Ph.D Thesis, DESY F31-86-02, Appendix E, (1986).

[8] G. J. Gounaris and J. J. Sakurai, Phys. Rev. Lett **21** 244 (1968).

[9] K. S. Cranmer, Comp. Phys. Comm. **136**, 198 (2001).

[10] L. Landau, J. Phys. USSR **8** (1944) 201; see also W. Allison and J. Cobb, Ann. Rev. Nucl. Part. Sci. **30** (1980) 253.

[11] The TMVA web page is `http://tmva.sourceforge.net/`

[12] The apparatus referred to here is described at `http://www.matphys.com/` and references therein.