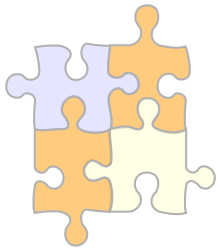


RooFit

授课教师：董燎原
中国科学院高能物理研究所

Tutorial by Wouter Verkerke

1 Introduction & Overview

- *Introduction*
- *Some basics statistics*
- *RooFit design philosophy*

RooFit: Your toolkit for data modeling

What is it?

- A powerful toolkit for modeling the expected distribution(s) of events in a physics analysis
- Primarily targeted to high-energy physicists using ROOT
- Originally developed for the BaBar collaboration by Wouter Verkerke and David Kirkby.
- Included with ROOT v5.xx

Documentation:

- http://root.cern.ch/root/Reference.html – for latest class descriptions. RooFit classes start with "Roo".
- http://roofit.sourceforge.net – for documentation and tutorials

Tutorials:

- Dig $ROOTSYS/tutorials/rootfit

RooFit purpose - Data Modeling for Physics Analysis

Distribution of observables $\vec{x}$

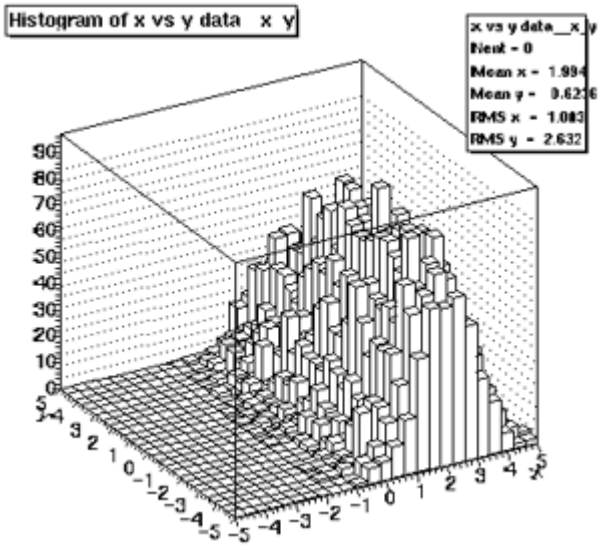
Define data model

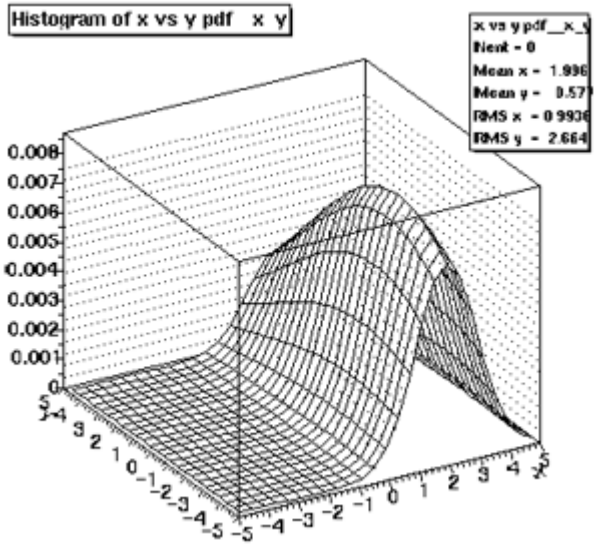
Probability Density Function $F(\vec{x}; \vec{p}, \vec{q})$

- Physical parameters of interest $\vec{p}$
- Other parameters $\vec{q}$ to describe detector effect (resolution,efficiency,…)
- Normalized over allowed range of the observables $\vec{x}$ w.r.t the parameters $\vec{p}$ and $\vec{q}$

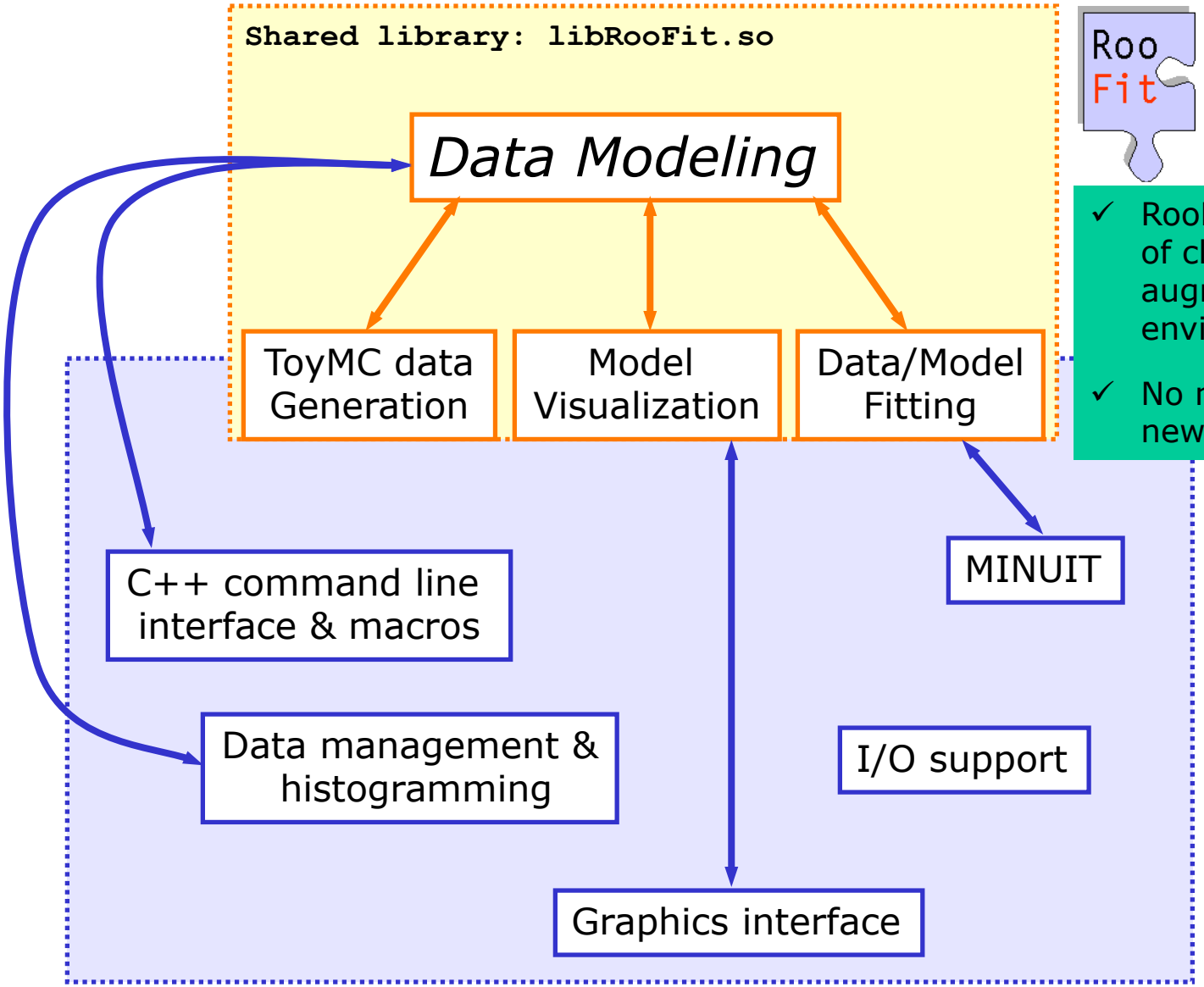
Fit model to data

Determination of $\vec{p},\vec{q}$



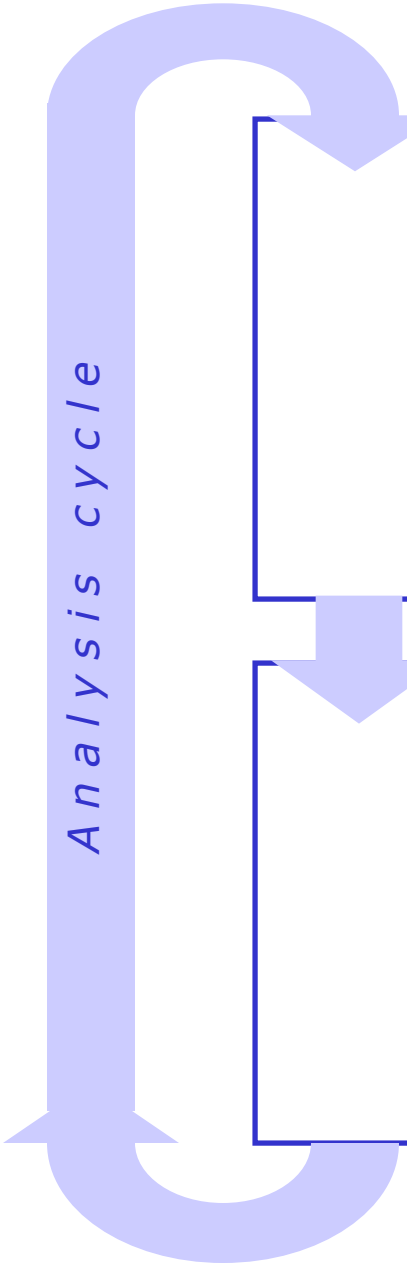


Implementation – Add-on package to ROOT



- ✓ RooFit is collection of classes that augment the ROOT environment.
- ✓ No need to learn new language.

Data modeling - Desired functionality



Building/Adjusting Models

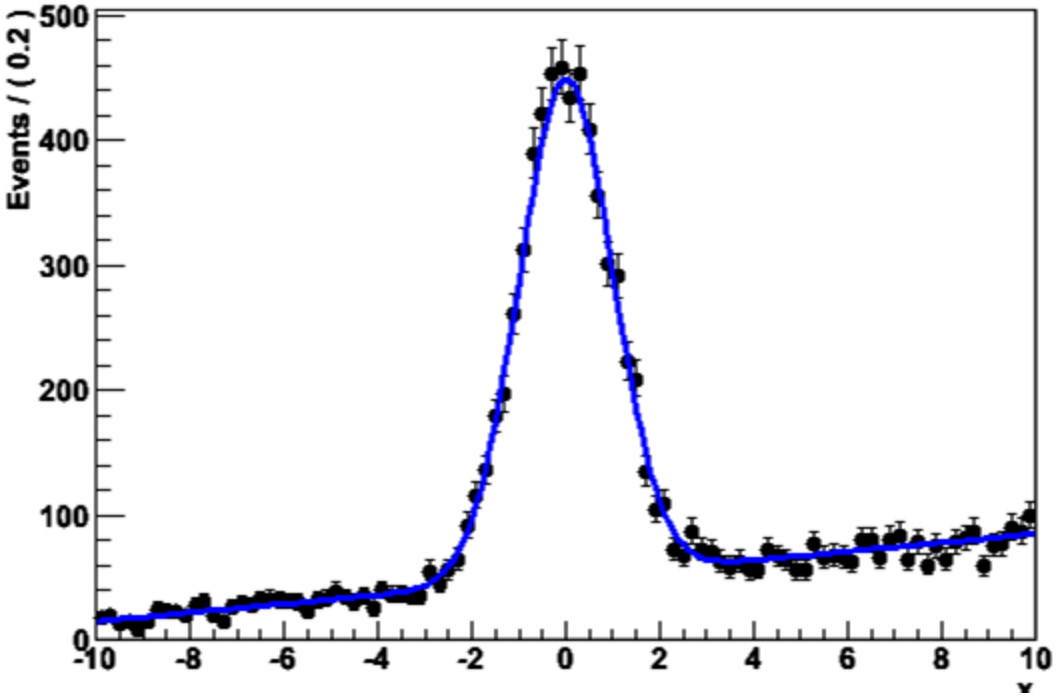
- ✓ *Easy to write* basic PDFs (→ normalization)
- ✓ Easy to *compose complex models* (modular design)
- ✓ *Reuse* of existing functions
- ✓ *Flexibility* – No arbitrary implementation-related restrictions

Using Models

- ✓ Fitting : Binned/Unbinned (extended) MLL fits, Chi^2 fits
- ✓ Toy MC generation: Generate MC datasets from *any* model
- ✓ Visualization: Slice/project model & data in *any possible way*
- ✓ Speed – Should be *as fast or faster* than hand-coded model

Introduction -- Focus: coding a probability density function

- Focus on one practical aspect of many data analysis in HEP: How do you formulate your p.d.f. in ROOT
  - For 'simple' problems (gauss, polynomial), ROOT built-in models well sufficient

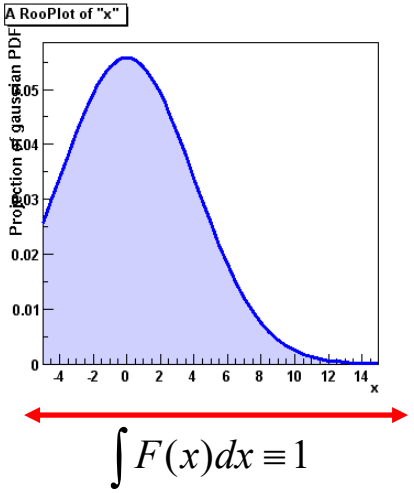


  - But if you want to do unbinned ML fits, use non-trivial functions, or work with multidimensional functions you are quickly running into trouble

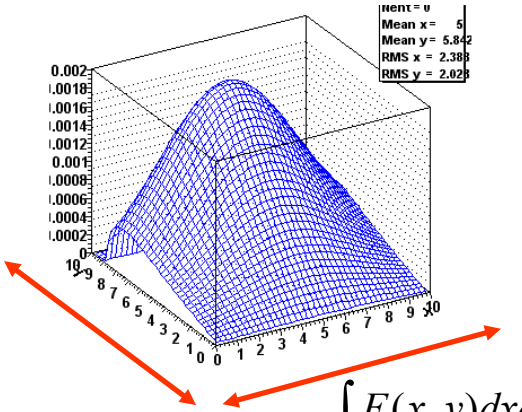
Mathematic – Probability density functions

- Probability Density Functions describe probabilities, thus
  - All values most be >0
  - The total probability must be 1 *for each* *p*, i.e.
  - Can have any number of dimensions

$$\int_{\vec{x}_{\min}}^{\vec{x}_{\max}} g(\vec{x},\vec{p})d\vec{x} \equiv 1$$



$$\int F(x)dx \equiv 1$$

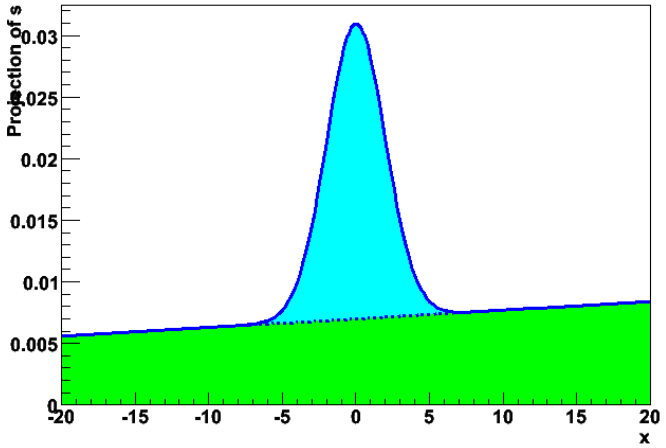


$$\int F(x,y)dxdy \equiv 1$$

- Note distinction in role between *parameters* (p) and *observables* (x)
  - Observables are measured quantities
  - Parameters are degrees of freedom in your model

Math – Functions vs probability density functions

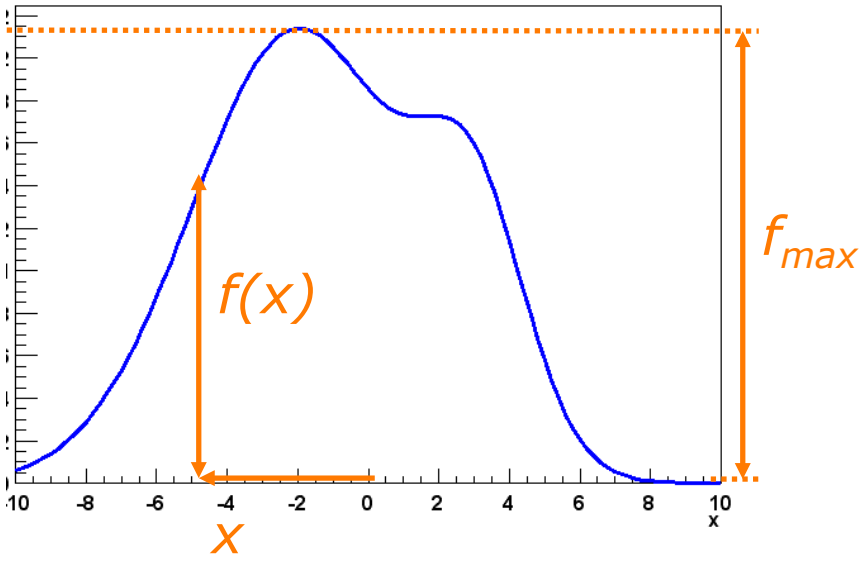
- Why use *probability density* functions rather than 'plain' functions to describe your data?
  - *Easier to interpret your models*. If Blue and Green pdf are each guaranteed to be normalized to 1, then fractions of Blue,Green can be cleanly interpreted as #events
  - Many statistical techniques only function properly with PDFs (e.g maximum likelihood)
  - Can sample 'toy Monte Carlo' events from p.d.f because value is always guaranteed to be >=0



- So why is not everybody always using them
  - The normalization can be hard to calculate (e.g. it can be different for each set of parameter values p)
  - In >1 dimension (numeric) integration can be particularly hard
  - RooFit aims to simplify these tasks

Math – Event generation

- For every p.d.f, can generate 'toy' event sample as follows
  - Determine maximum PDF value by repeated random sample
  - Throw a uniform random value (x) for the observable to be generated
  - Throw another uniform random number between 0 and fmax
    If ran*f_{max} < f(x) accept x as generated event
  - *More efficient techniques exist*



Math – What is an estimator?

- An ***estimator*** is a ***procedure*** giving a value for a parameter or a property of a distribution as a function of the actual data values, i.e.

$$\hat{\mu}(x) = \frac{1}{N}\sum_i x_i$$ ← **Estimator of the mean**

$$\hat{V}(x) = \frac{1}{N}\sum_i (x_i - \vec{\mu})^2$$ ← **Estimator of the variance**

- A perfect estimator is （一致性，无偏性，有效性）
  - Consistent: $\lim_{n\to\infty}(\hat{a}) = a$
  - Unbiased – *With finite statistics you get the right answer on average*
  - Efficient $V(\hat{a}) = \left\langle (\hat{a} - \left\langle \hat{a} \right\rangle)^2 \right\rangle$ ← This is called the 最小方差界 **Minimum Variance Bound**
  - ***There are no perfect estimators for real-life problems***

Math – The Likelihood estimator

- Definition of Likelihood
  - given **D($\vec{x}$)** and **F($\vec{x}$;$\vec{p}$)**

Functions used in likelihoods must be Probability Density Functions:

$$\int F(\vec{x};\vec{p})d\vec{x} \equiv 1, \quad F(\vec{x};\vec{p}) > 0$$

$$L(\vec{p}) = \prod_i F(\vec{x}_i;\vec{p}), \quad \text{i.e.} \quad L(\vec{p}) = F(x_0;\vec{p}) \cdot F(x_1;\vec{p}) \cdot F(x_2;\vec{p}) \ldots$$

  - For convenience the *negative log* of the Likelihood is often used

$$-\ln L(\vec{p}) = -\sum_i \ln F(\vec{x}_i;\vec{p})$$

- Parameters are estimated by maximizing the Likelihood, or equivalently minimizing –log(L)

$$\left.\frac{d \ln L(\vec{p})}{d\vec{p}}\right|_{p_i=\hat{p}_i} = 0$$

Math – Variance on ML parameter estimates

- **Estimator** for the **parameter variance** is

$$\hat{\sigma}(p)^2 = \hat{V}(p) = \left(\frac{d^2 \ln L}{d^2 p} \right)^{-1}$$

  - I.e. variance is estimated from 2nd derivative of –log(L) at minimum
  - Valid if estimator is **efficient** and **unbiased**!

From Rao-Cramer-Frechet inequality

$$V(\hat{p}) \geq \left. 1 + \frac{db}{dp} \middle/ \left(\frac{d^2 \ln L}{d^2 p} \right) \right.$$

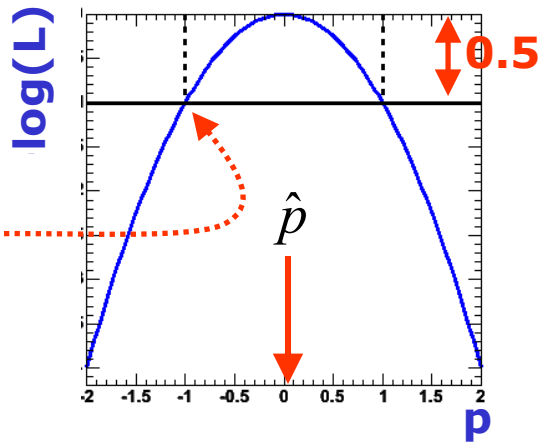
b = bias as function of p, inequality becomes equality in limit of efficient estimator

- **Visual interpretation** of variance estimate
  - Taylor expand –log(L) around minimum

$$\ln L(p) = \ln L(\hat{p}) + \left. \frac{d \ln L}{dp} \right|_{p=\hat{p}} (p - \hat{p}) + \tfrac{1}{2} \left. \frac{d^2 \ln L}{d^2 p} \right|_{p=\hat{p}} (p - \hat{p})^2$$

$$= \ln L_{\max} + \left. \frac{d^2 \ln L}{d^2 p} \right|_{p=\hat{p}} \frac{(p - \hat{p})^2}{2}$$

$$= \ln L_{\max} + \frac{(p - \hat{p})^2}{2\hat{\sigma}_p^2} \quad \Rightarrow \quad \ln L(p \pm \sigma) = \ln L_{\max} - \frac{1}{2}$$



Math – Properties of Maximum Likelihood estimators

- In general, Maximum Likelihood estimators are
  - **Consistent** (gives right answer for N→∞)
  - **Mostly unbiased** (bias ∝1/N, may need to worry at small N)
  - **Efficient for large N** (you get the smallest possible error)
  - **Invariant**: (a transformation of parameters will Not change your answer, e.g $(\hat{p})^2 = \widehat{(p^2)}$

Use of 2nd derivative of –log(L) for variance estimate is usually OK

- MLE efficiency theorem: the MLE will be unbiased and efficient if an unbiased efficient estimator exists

Math – Extended Maximum Likelihood

- Maximum likelihood information only parameterizes *shape* of distribution
  - I.e. one can determine *fraction* of signal events from ML fit, but not *number* of signal events

$$L(\vec{p}) = \prod_i F(\vec{x}_i; \vec{p}), \quad \text{i.e.} \quad L(\vec{p}) = F(x_0; \vec{p}) \cdot F(x_1; \vec{p}) \cdot F(x_2; \vec{p}) \ldots$$

- Extended Maximum likelihood add extra term

> The information can be incorporated by combining the standard maximum likelihood with the knowledge that a particular $Q(x;a)$ predicts ν events in the observed range, and accordingly multiplies the likelihood of a given data sample of N events by the Poisson probability of obtaining N events from a mean of ν:

→ $e^{-\nu}\frac{\nu^N}{N!}$

$$-\log\left(L(\vec{p})\right) = -\sum_D \log(g(\vec{x}_i, \vec{p})) + \boxed{N_{\exp} - N_{obs}\log(N_{\exp})}$$

Log of Poisson(Nexp,Nobs) (modulo a constant)

  - Clever choice of parameters will allows us to extract N_{sig} and N_{bkg} in one pass ($N_{exp}=N_{sig}+N_{bkg}$, $f_{sig}=N_{sig}/(N_{sig}+N_{bkg})$)

RooFit core design philosophy

- Mathematical objects are represented as C++ objects

Mathematical concept		RooFit class
variable	x	`RooRealVar`
function	$f(x)$	`RooAbsReal`
PDF	$f(x)$	`RooAbsPdf`
space point	$\vec{x}$	`RooArgSet`
integral	$\int_{x_{\min}}^{x_{\max}} f(x)dx$	`RooRealIntegral`
list of space points		`RooAbsData`

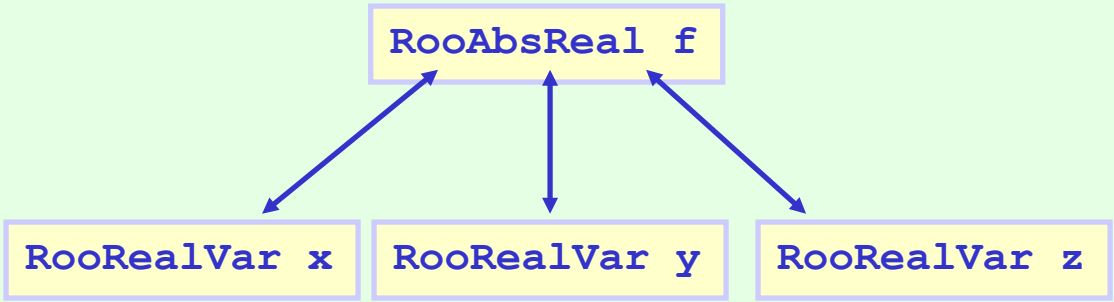
RooFit core design philosophy

- Represent relations between variables and functions as client/server links between objects

Math

$$f(x,y,z)$$

RooFit diagram



RooFit code

```
RooRealVar x("x","x",5) ;
RooRealVar y("y","y",5) ;
RooRealVar z("z","z",5) ;
RooBogusFunction f("f","f",x,y,z) ;
```

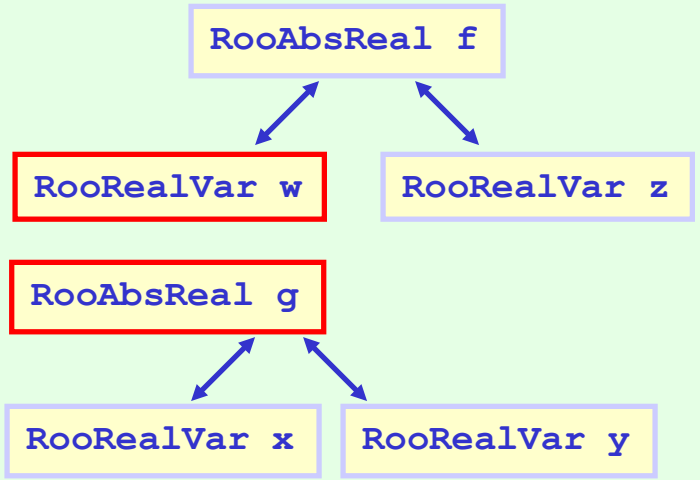
RooFit core design philosophy

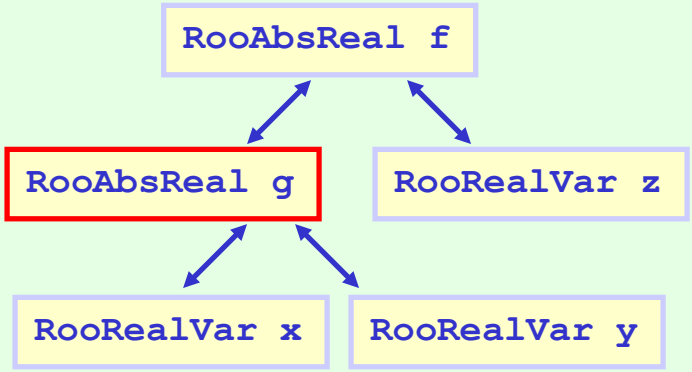
- Composite functions → Composite objects

Math

$f(w,z) \quad g(x,y) \rightarrow f(g(x,y),z) = f(x,y,z)$

RooFit diagram





RooFit code

```
RooRealVar x("x","x",2) ;
RooRealVar y("y","y",3) ;
RooGooFunc g("g","g",x,y) ;

RooRealVar w("w","w",0) ;
RooRealVar z("z","z",5) ;
RooFooFunc f("f","f",w,z) ;
```

```
RooRealVar x("x","x",2) ;
RooRealVar y("y","y",3) ;
RooGooFunc g("g","g",x,y) ;

RooRealVar z("z","z",5) ;
RooFooFunc f("f","f",g,z) ;
```

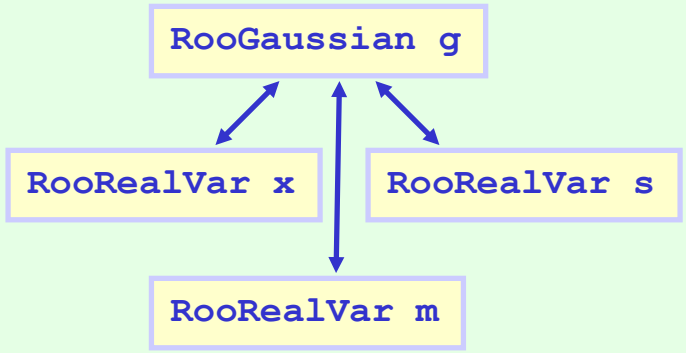
RooFit core design philosophy

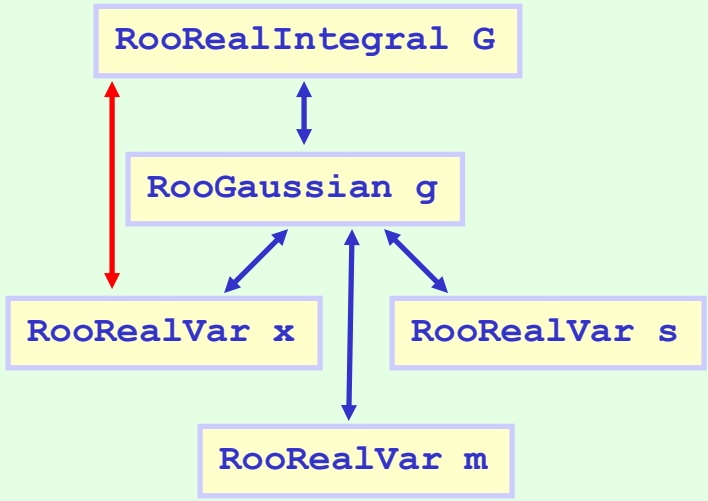
- Represent integral as an object,
  instead of representing integration as an action

Math

$$g(x,m,s) \rightarrow \int_{x_{\min}}^{x_{\max}} g(x,m,s)dx = G(m,s,x_{\min},x_{\max})$$

RooFit diagram





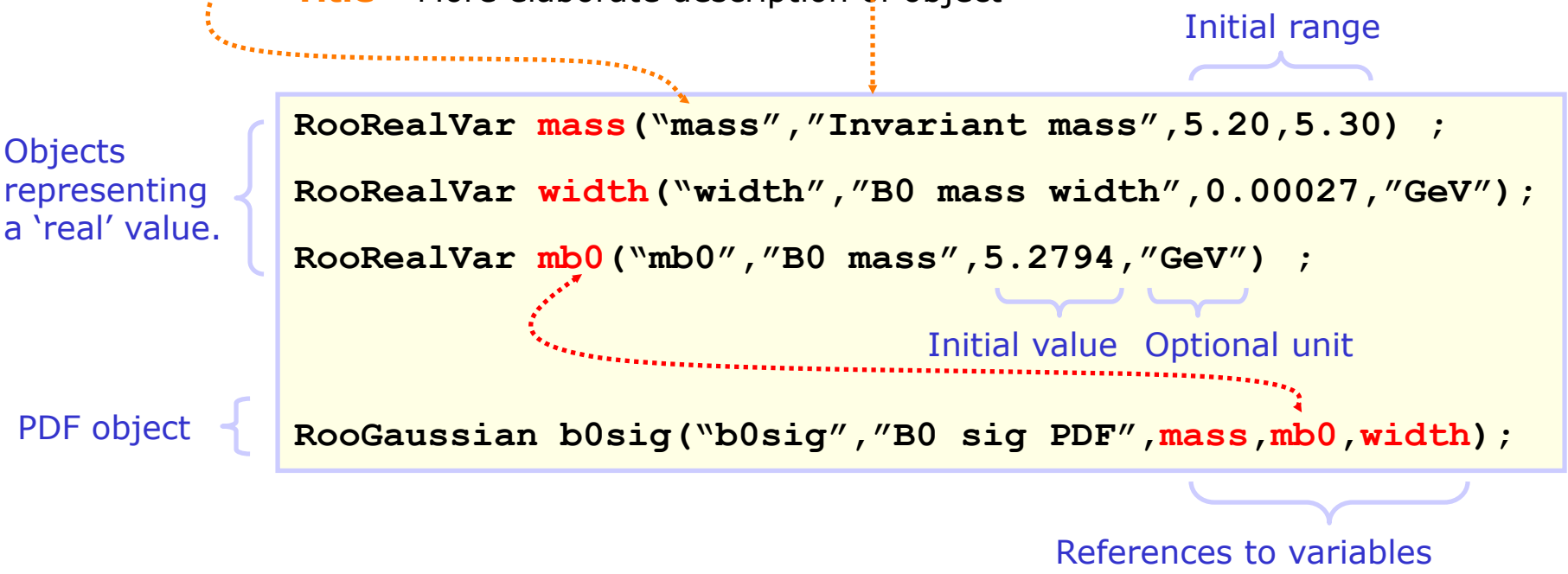
RooFit code

```
RooRealVar x("x","x",2,-10,10)
RooRealVar s("s","s",3) ;
RooRealVar m("m","m",0) ;
RooGaussian g("g","g",x,m,s)
```

```
RooAbsReal *G =
      g.createIntegral(x) ;
```

Object-oriented data modeling

- In RooFit every variable, data point, function, PDF represented in a C++ object
  - Objects classified by data/function type they represent, not by their role in a particular setup
  - All objects are **self documenting**
    - **Name** - Unique identifier of object
    - **Title** – More elaborate description of object



Object-oriented data modeling

- Elementary operations on value holder objects

Print value and attributes

```
mass.Print()
RooRealVar::mass:  5.2500 L(5.2 - 5.3)
```

Assign new value

```
mass = 5.27 ;
mass.setVal(5.27) ;
mass = 9.0  ;
RooAbsRealLValue::inFitRange(mass):
    value 9 rounded down to max limit 5.3
```

Error: new value out of allowed range

Retrieve contents

```
Double_t massVal = mass.getVal();
```

Print works for all RooFit objects

```
b0sig.Print()
RooGaussian::b0sig(mass,mb0,width) = 0
```

getVal() works for all real-valued objects (variables and functions)

```
Double_t val = b0sig.getVal()
```

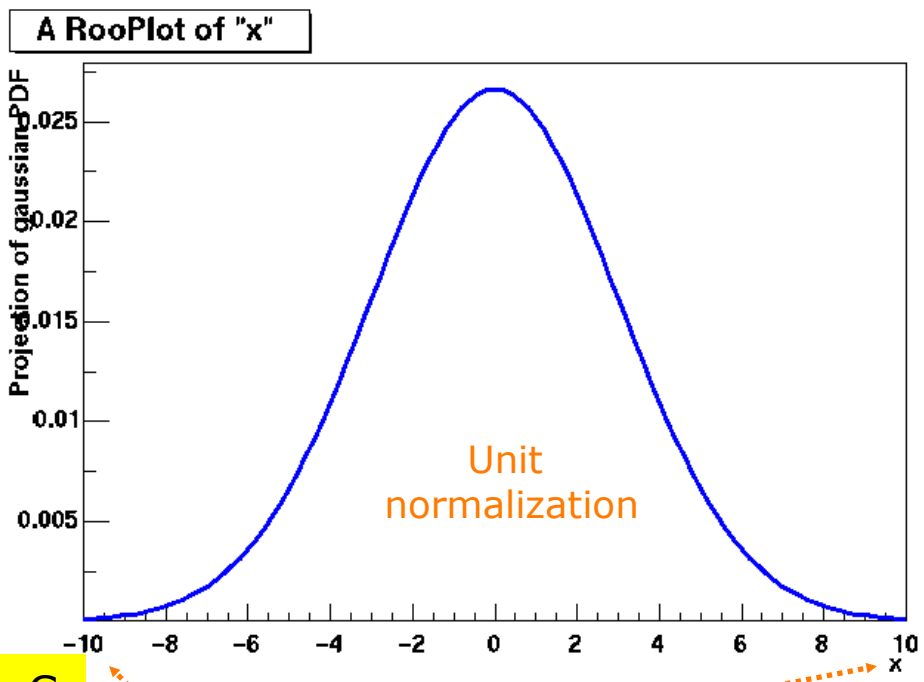
2 Basic Functionality

- *Creating a p.d.f*
- *Basic fitting, plotting, event generation*
- *Some details on normalization, event generation*
- *Library of basic shapes (including non-parametric shapes)*

Basics – Creating and plotting a Gaussian p.d.f

Setup gaussian PDF and plot

```
// Build Gaussian PDF
RooRealVar x("x","x",-10,10) ;
RooRealVar mean("mean","mean of gaussian",0,-10,10) ;
RooRealVar sigma("sigma","width of gaussian",3) ;

RooGaussian gauss("gauss","gaussian PDF",x,mean,sigma) ;

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



Axis label from `gauss` title

A `RooPlot` is an empty frame capable of holding anything plotted versus it variable

$ROOTSYS/tutorials/roofit/rf101_basics.C

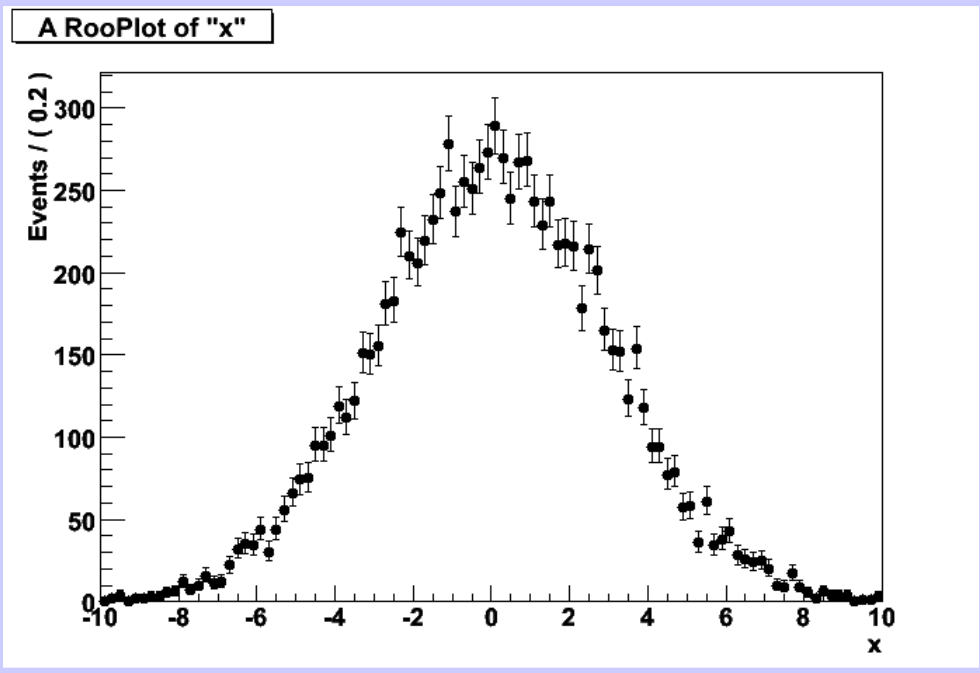
Plot range taken from limits of `x`

Basics – Generating toy MC events

demo1.cc

Generate 10000 events from Gaussian p.d.f and show distribution

```
// Generate a toy MC set
RooDataSet* data = gauss.generate(x,10000) ;

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



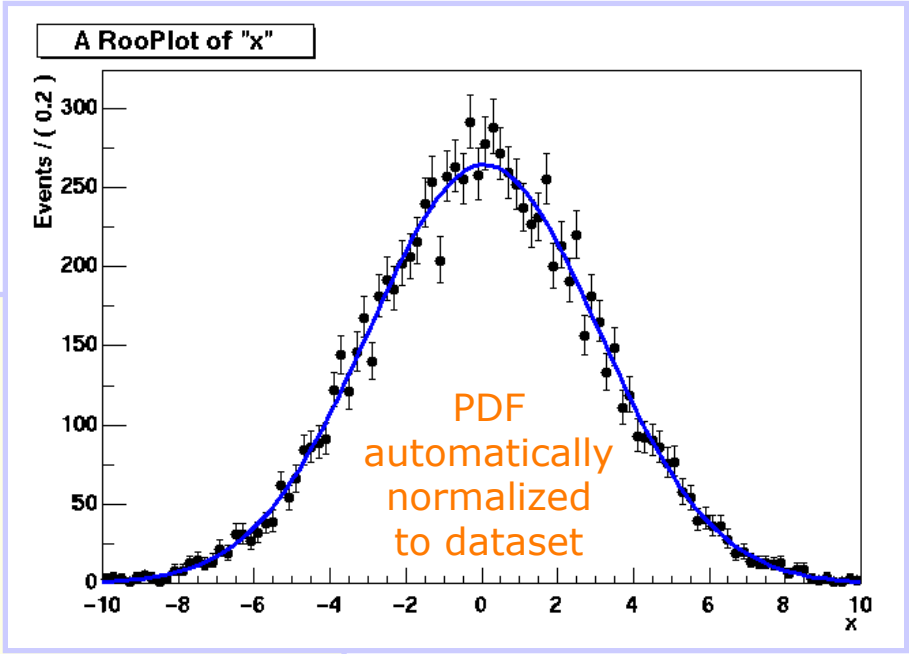
Returned dataset is ***unbinned*** dataset

Binning into histogram is performed in `data->plotOn()` call

Once the model is built, Generating ToyMC, fitting, plotting are mostly one-line operations!

Basics – ML fit of p.d.f to *unbinned* data

demo1.cc



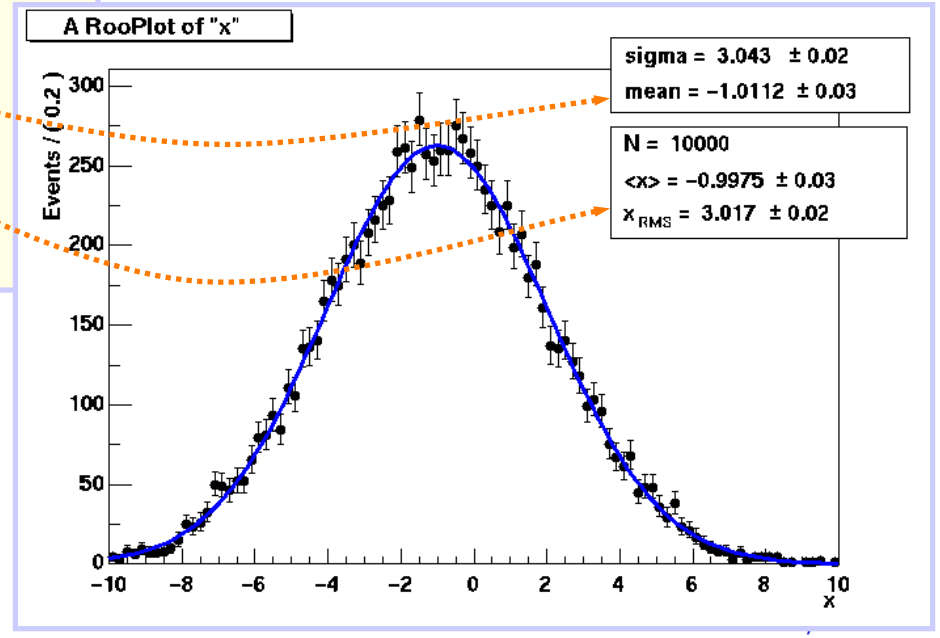

```
// ML fit of gauss to data
gauss.fitTo(*data) ;
(MINUIT printout omitted)

// Parameters if gauss now
// reflect fitted values
mean.Print()
RooRealVar::mean = 0.0172335 +/- 0.0299542
sigma.Print()
RooRealVar::sigma = 2.98094  +/- 0.0217306

// Plot fitted PDF and toy data overlaid
RooPlot* xframe2 = x.frame() ;
data->plotOn(xframe2) ;
gauss.plotOn(xframe2) ;
xframe2->Draw() ;
```

Basics – RooPlot Decoration

- A RooPlot is an empty frame that can contain
  - RooDataSet projections
  - PDF and generic real-valued function projections
  - Any ROOT drawable object (arrows, text boxes etc)

- Adding a dataset statistics box / PDF parameter box

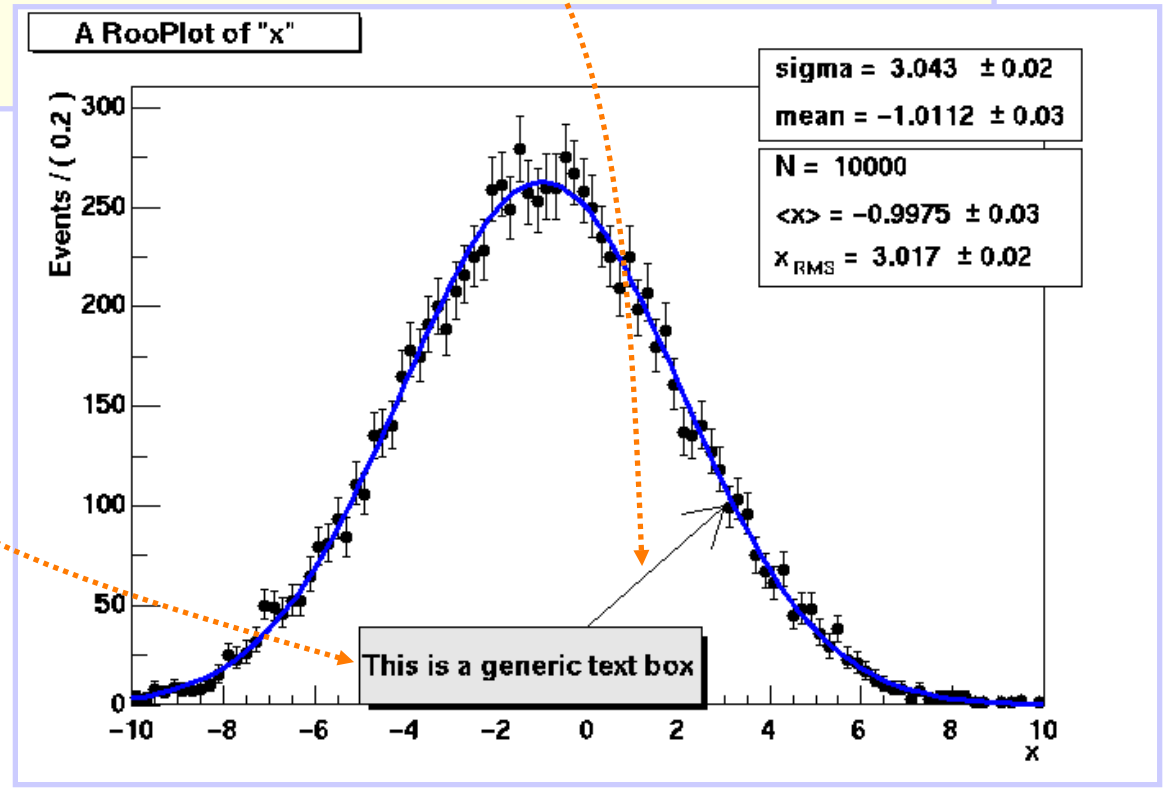
```
RooPlot* frame = x.frame() ;
data.plotOn(xframe) ;
pdf.plotOn(xframe) ;
pdf.paramOn(xframe,data) ;
data.statOn(xframe) ;
xframe->Draw() ;
```



Basics – RooPlot decoration

- Adding generic ROOT text boxes, arrows etc.

```
TPaveText* tbox = new TPaveText(0.3,0.1,0.6,0.2,"BRNDC");
tbox->AddText("This is a generic text box") ;
TArrow* arr = new TArrow(0,40,3,100) ;

xframe2->addObject(arr) ;
xframe2->addObject(tbox) ;
```



You can save a RooPlot with all its decorations in a ROOT file

Basics – Observables and parameters of Gauss

- Class **`RooGaussian`** has *no intrinsic notion* of distinction between observables and parameters

- Distinction always implicit in use context with dataset
  - **x** = observable (as it is a variable in the dataset)
  - **mean,sigma** = parameters

- Choice of observables (for unit normalization) always passed to **`gauss.getVal()`**

```
gauss.getVal();      // Not normalized (i.e. this is _not_ a pdf)
gauss.getVal(x);     // Guarantees Int[xmin,xmax] Gauss(x,m,s)dx==1
gauss.getVal(sigma);// Guarantees Int[smin,smax] Gauss(x,m,s)ds==1
```

How does it work – Normalization

- Flexible choice of normalization facilitated by explicit normalization step in RooFit p.d.f.s

`gauss.getVal(x)`

$$g(\mathbf{x};m,s)=\frac{g(x,m,s)}{\int_{x_{\min}}^{x_{\max}} g(x,m,s)dx}$$

`gauss.getVal(s)`

$$g(\mathbf{s};m,x)=\frac{g(x,m,s)}{\int_{s_{\min}}^{s_{\max}} g(x,m,s)ds}$$

- Supporting class **`RooRealIntegral`** responsible for calculation of any

$$\int_{\vec{x}_{\min}}^{\vec{x}_{\max}} g(\vec{x};\vec{p})d\vec{x}$$

  - Negotiation with p.d.f on which (partial) integrals it can internally perform analytically
  - Missing parted are supplemented with numerical integration
  - Class **`RooRealIntegral`** can in principle integrate *everything.*

How does it work – Normalization

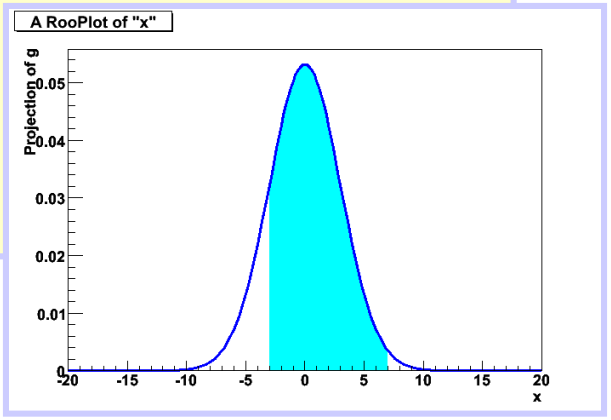
- A peak in the code of class **RooGaussian**

```
// Raw (unnormalized value) of Gaussian
Double_t RooGaussian::evaluate() const {
  Double_t arg= x - mean;
  return exp(-0.5*arg*arg/(sigma*sigma)) ;
}

// Advertise that x can be integrated internally
Int_t RooGaussian::getAnalyticalIntegral(RooArgSet& allVars,
      RooArgSet& analVars, const char* /*rangeName*/) const {
  if (matchArgs(allVars,analVars,x)) return 1 ;
  return 0 ;
}

// Implementation of analytical integral over x
Double_t RooGaussian::analyticalIntegral(Int_t code,
                                         const char* rname) const {
  static const Double_t root2 = sqrt(2.) ;
  static const Double_t rootPiBy2 = sqrt(atan2(0.0,-1.0)/2.0);
  Double_t xscale = root2*sigma;
  return rootPiBy2*sigma*(RooMath::erf((x.max(rname)-mean)/xscale)
                         -RooMath::erf((x.min(rname)-mean)/xscale));
}
```

Basics – Integrals over p.d.f.s

- It is easy to create an object representing integral over a normalized p.d.f in a sub-range

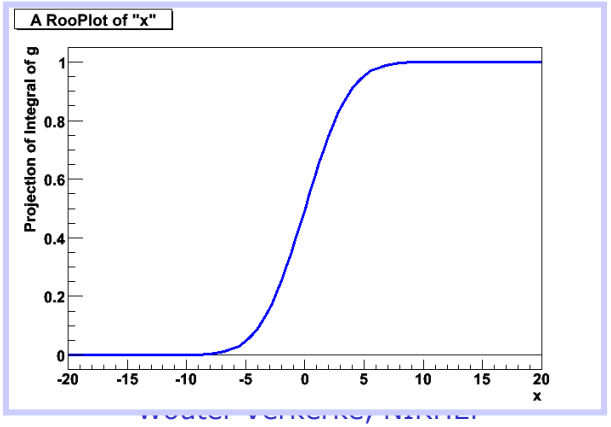
```
x.setRange("sig",-3,7) ;
RooAbsReal* ig = g.createIntegral(x,NormSet(x),Range("sig")) ;
cout << ig.getVal() ;
0.832519
mean=-1
cout << ig.getVal() ;
0.743677
```



- Similarly, one can also request the *cumulative distribution function*

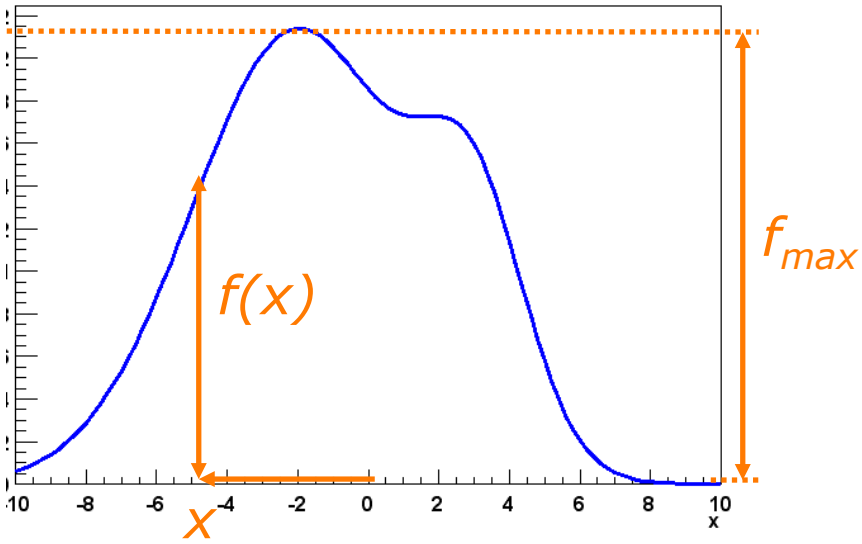
$$C(x) = \int_{x_{\min}}^{x} F(x')dx'$$

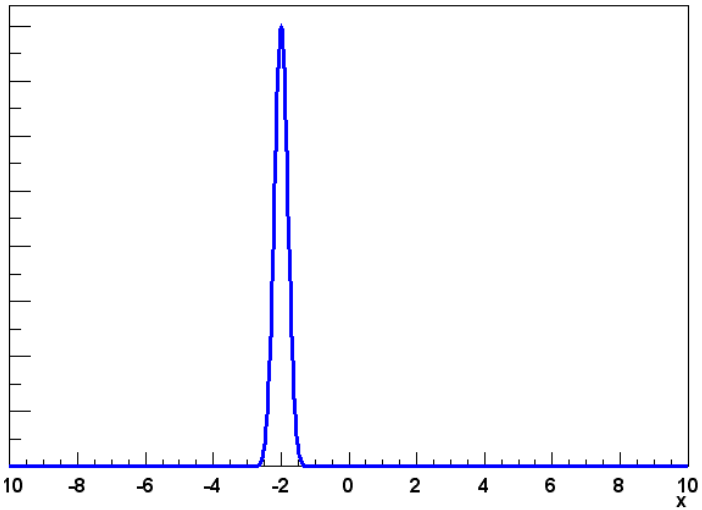
```
RooAbsReal* cdf = gauss.createCdf(x) ;
RooPlot* frame = x.frame() ;
cdf->plotOn(frame)->Draw() ;
```



How does it work – toy event generation

- Accept/reject method can be very inefficient
  - Generating efficiency is $$\frac{\int_{x_{\min}}^{x_{\max}} f(x)dx}{(x_{\max} - x_{\min}) \cdot f_{\max}}$$
  - Efficiency is very low for narrowly peaked functions
  - Initial sampling for f_{max} requires very large trials sets in multiple dimension (~10000000 in 3D)





Toy MC generation – Inversion method

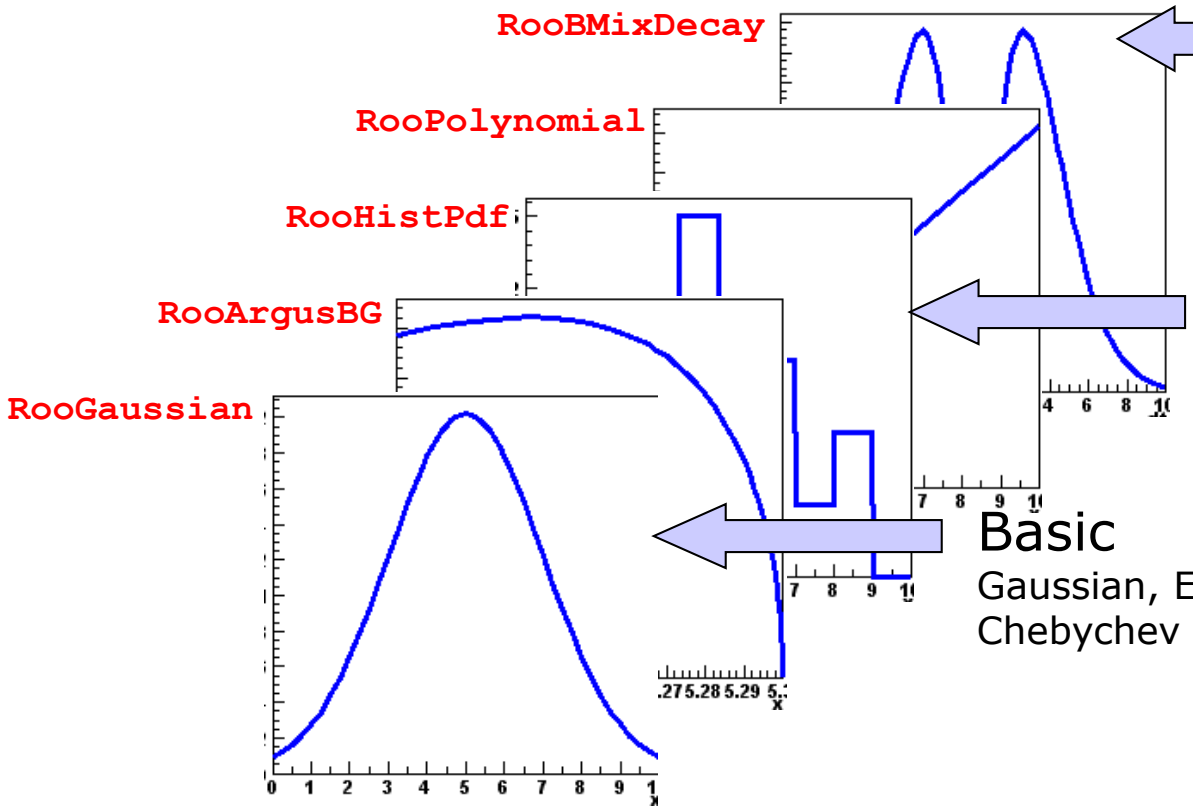
- Analogous to integration, p.d.f can advertise internal generator in case it can be done with a more efficient technique
- E.g. function inversion

1) Given f(x) find inverted function F(x) so that f(F(x)) = x
2) Throw uniform random number x
3) Return F(x)

- Maximally efficient, but only works for class of p.d.f.s that is invertible

Model building – (Re)using standard components

- RooFit provides a collection of compiled standard PDF classes



Physics inspired
ARGUS,Crystal Ball,
Breit-Wigner, Voigtian,
B/D-Decay,….

Non-parametric
Histogram, **KEYS**

Basic
Gaussian, Exponential, Polynomial,…
Chebychev polynomial

Easy to extend the library: each p.d.f. is a separate C++ class

The building blocks

- RooFitModels provides a collection of 'building block' PDFs

RooArgusBG	– Argus background shape
RooBCPEffDecay	– B0 decay with CP violation
RooBMixDecay	-B0 decay with mixing
RooBifurGauss	-Bifurcated Gaussian
RooBreitWigner	-Breit-Wigner shape
RooCBShape	-Crystal Ball function
RooChebychev	-Chebychev polynomial
RooDecay	-Simple decay function
RooDircPdf	-DIRC resolution description
RooDstD0BG	-D* background description
RooExponential	– Exponential function
RooGaussian	-Gaussian function
RooKeysPdf	-Non-parametric data description
Roo2DKeysPdf	-Non-parametric data description
RooPolynomial	-Generic polynomial PDF
RooVoigtian	-Breit-Wigner (X) Gaussian

- More will PDFs will follow
  - Easy to for users to write/contribute new PDFs

以上源程序都在 roofit/src 中

Model building – Generic expression-based PDFs

- If your favorite PDF isn't there
  and you don't want to code a PDF class right away
  → use **RooGenericPdf**
- Just write down the PDFs expression as a C++ formula

```
// PDF variables
RooRealVar x("x","x",-10,10) ;
RooRealVar y("y","y",0,5) ;
RooRealVar a("a","a",3.0) ;
RooRealVar b("b","b",-2.0) ;

// Generic PDF
RooGenericPdf gp("gp","Generic PDF","exp(x*y+a)-b*x",
                 RooArgSet(x,y,a,b)) ;
```

- Numeric normalization automatically provided

Model Building – Writing your own class

- Factory class exists (**`RooClassFactory`**) that can write, compile, link C++ code for RooFit p.d.f. and function classes
- **Example 1**:
  - Write class **`MyPdf`** with variable x,a,b in files **`MyPdf.h, MyPdf.cxx`**

```
RooClassFactory::makePdf("MyPdf","x,a,b");
```

  - Only need to fill **`evaluate()`** method in **`MyPdf.cxx`** in terms of a,b,x
  - Can add optional code to support for analytical integration, internal event generation

.x tutorials/roofit/rf104_classfactory.C
生成 MyPdfV?.h 和 MyPdfV?.cxx (共6个文件)

MyPdf.cxx and MyPdf.h

MyPdfV1.cxx

```
#include "Riostream.h"

#include "MyPdfV1.h"
#include "RooAbsReal.h"
#include "RooAbsCategory.h"
#include <math.h>
#include "TMath.h"

ClassImp(MyPdfV1)

 MyPdfV1::MyPdfV1(const char *name, const char *title,
                  RooAbsReal& _x,
                  RooAbsReal& _A,
                  RooAbsReal& _B) :
   RooAbsPdf(name,title),
   x("x","x",this,_x),
   A("A","A",this,_A),
   B("B","B",this,_B)
 {
 }


 MyPdfV1::MyPdfV1(const MyPdfV1& other, const char* name) :
   RooAbsPdf(other,name),
   x("x",this,other.x),
   A("A",this,other.A),
   B("B",this,other.B)
 {
 }

 Double_t MyPdfV1::evaluate() const
 {
   // ENTER EXPRESSION IN TERMS OF VARIABLE ARGUMENTS HERE
   return 1.0 ;
 }
```

MyPdfV1.h

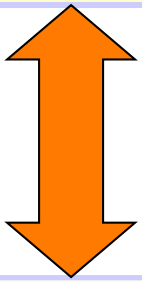
```
#ifndef MYPDFV1
#define MYPDFV1

#include "RooAbsPdf.h"
#include "RooRealProxy.h"
#include "RooCategoryProxy.h"
#include "RooAbsReal.h"
#include "RooAbsCategory.h"

class MyPdfV1 : public RooAbsPdf {
public:
  MyPdfV1() {} ;
  MyPdfV1(const char *name, const char *title,
          RooAbsReal& _x,
          RooAbsReal& _A,
          RooAbsReal& _B);
  MyPdfV1(const MyPdfV1& other, const char* name=0) ;
  virtual TObject* clone(const char* newname) const {
                  return new MyPdfV1(*this,newname); }
  inline virtual ~MyPdfV1() { }

protected:

  RooRealProxy x ;
  RooRealProxy A ;
  RooRealProxy B ;

  Double_t evaluate() const ;

private:

  ClassDef(MyPdfV1,1) // Your description goes here...
};

#endif
```

Model Building – Writing your own class

- **Example 2**:
  - Functional equivalent to `RooGenericPdf`: Write class `MyPdf` with prefilled one-line function expression, compile and link p.d.f, create and return instance of class

Compiled code

```
RooAbsPdf* gp = RooClassFactory::makePdfInstance("gp",
                          "exp(x*y+a)-b*x",RooArgSet(x,y,a,b));
```



Interpreted code

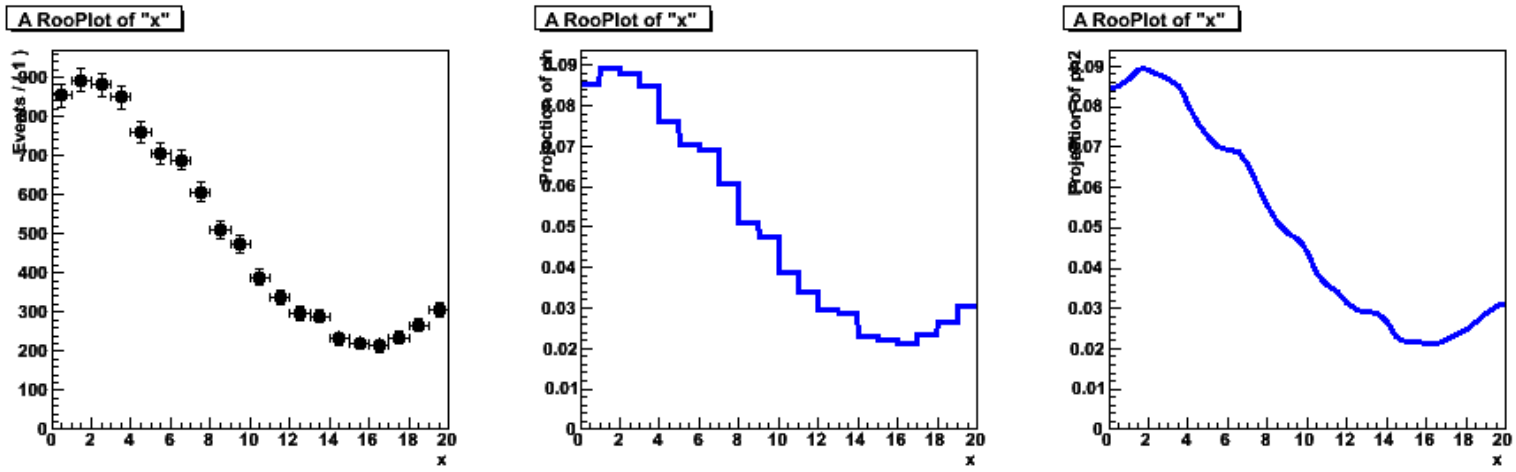
```
RooGenericPdf gp("gp","Generic PDF","exp(x*y+a)-b*x",
                    RooArgSet(x,y,a,b)) ;
```

参考 tutorials/roofit/rf104_classfactory.C

Highlight of non-parametric shapes - histograms

- Will highlight two types of non-parametric p.d.f.s
- Class **`RooHistPdf`** – a p.d.f. described by a histogram

`dataHist` **`RooHistPdf(N=0)`** **`RooHistPdf(N=4)`**



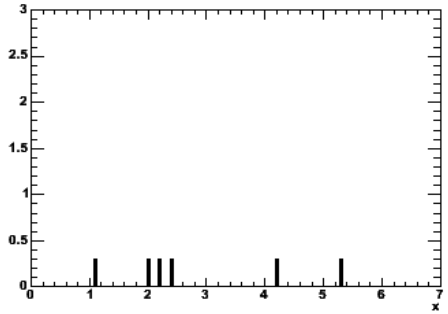

```
// Histogram based p.d.f with N-th order interpolation（插值）
RooHistPdf ph("ph","ph",x,*dataHist,N) ;
```

- Not so great at low statistics (especially problematic in >1 dim)

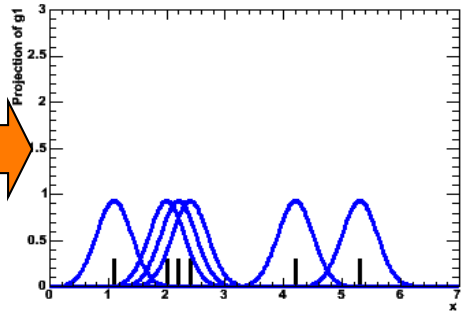
Highlight of non-parametric shapes – kernel estimation

- Class **RooKeysPdf** – A kernel estimation p.d.f.
  - Uses *unbinned* data
  - Idea represent each event of your MC sample as a Gaussian probability distribution
  - Add probability distributions from all events in sample

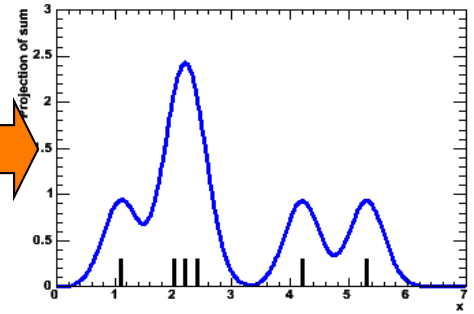
Sample of events



Gaussian probability distributions for each event



Summed probability distribution for all events in sample

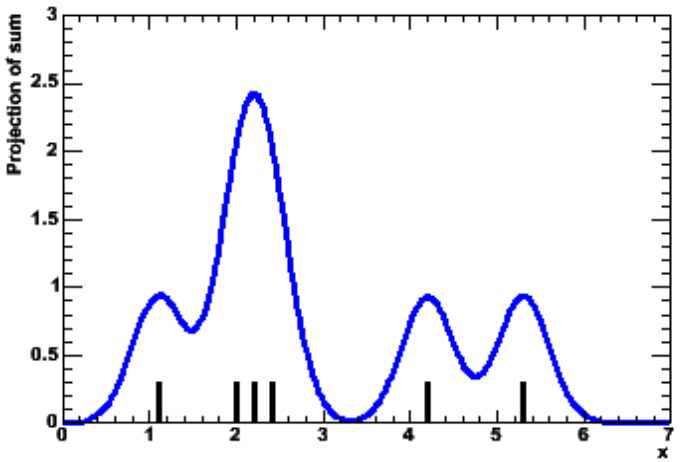


Kernel Estimation in High-Energy Physics: http://arxiv.org/abs/hep-ex/0011057

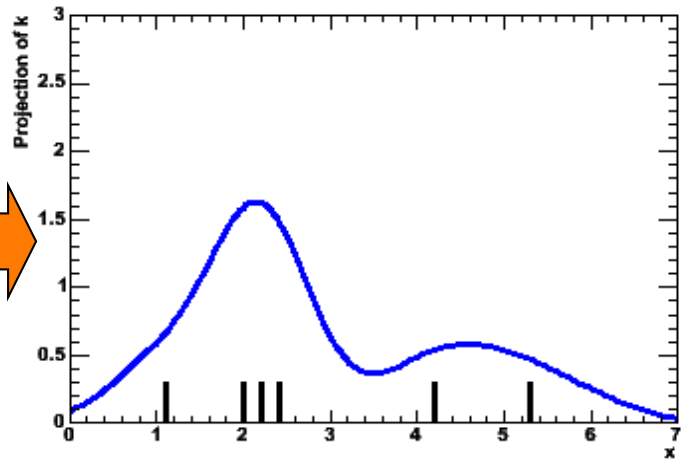
Highlight of non-parametric shapes – kernel estimation

- Width of Gaussian kernels need not be the same for all events
  - As long as each event contributes 1/N to the integral
- Idea: 'Adaptive kernel' technique
  - Choose wide Gaussian if local density of events is low
  - Choose narrow Gaussian if local density of events is high
  - Preserves small features in high statistics areas, minimize jitter in low statistics areas
  - Automatically calculated

Static Kernel (with of all Gaussian identical)



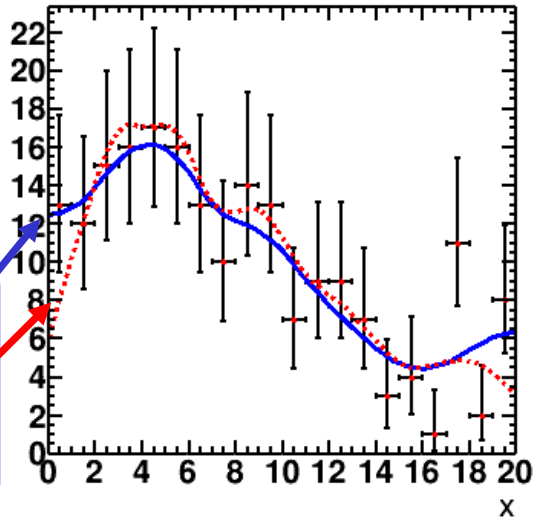
Adaptive Kernel (width of all Gaussian depends on local density of events)



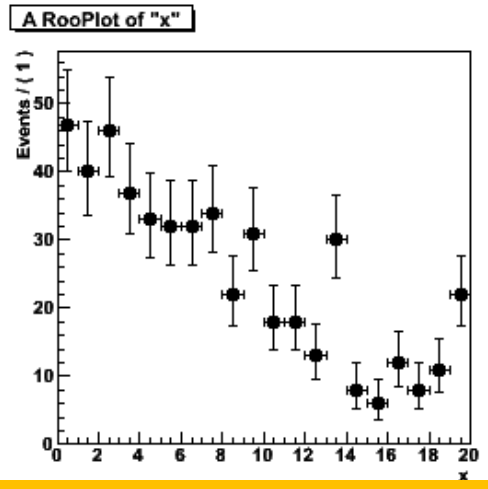
Highlight of non-parametric shapes – kernel estimation

- Example with comparison to histogram based p.d.f
  - Superior performance at low statistics
  - Can mirror input data over boundaries to reduce 'edge leakage'
  - Works also in >1 dimensions (class `RooNDKeysPdf`)

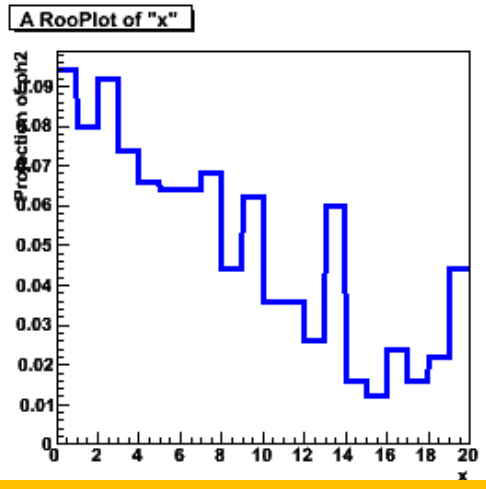
```
// Adaptive kernel estimation p.d.f
 RooKeysPdf k("k","k",x,*d,RooKeysPdf::MirrorBoth);
//                         RooKeysPdf::noMirror
```



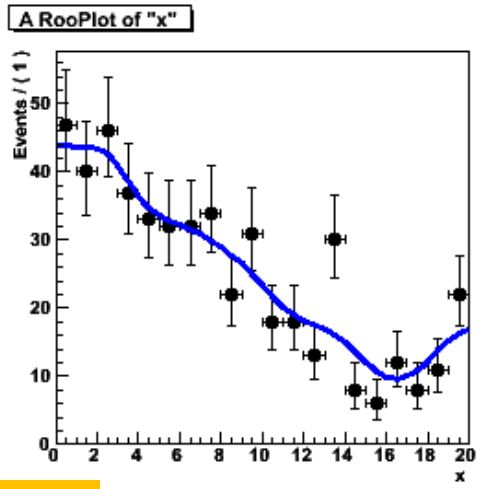
Data (N=500)



RooHistPdf(data)



RooKeysPdf(data)



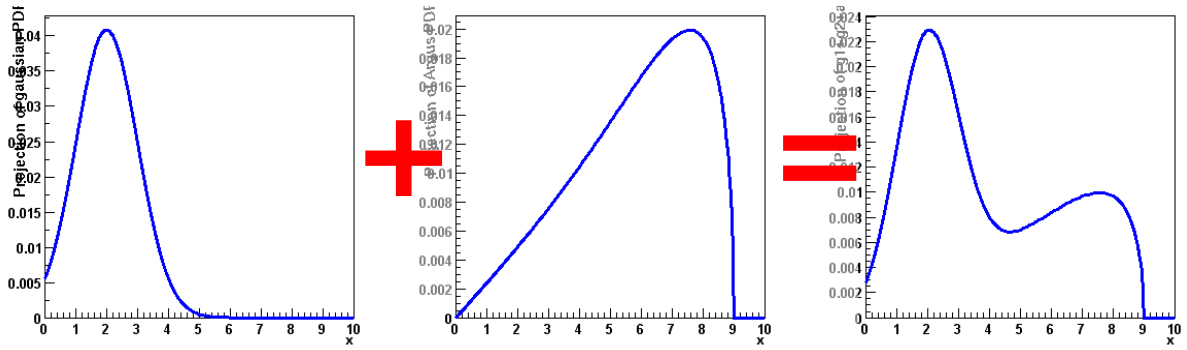
参考 tutorials/roofit/rf707_kernelestimation.C

3 P.d.f. addition & convolution

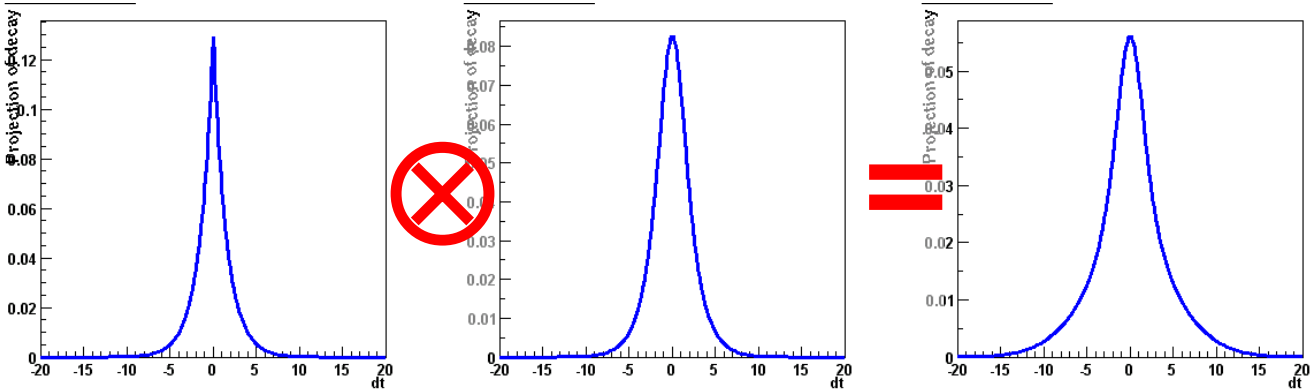
- *Using the addition operator p.d.f*
- *Using the convolution operator p.d.f.*

Building realistic models

- Complex PDFs be can be trivially composed using operator classes
  - Addition

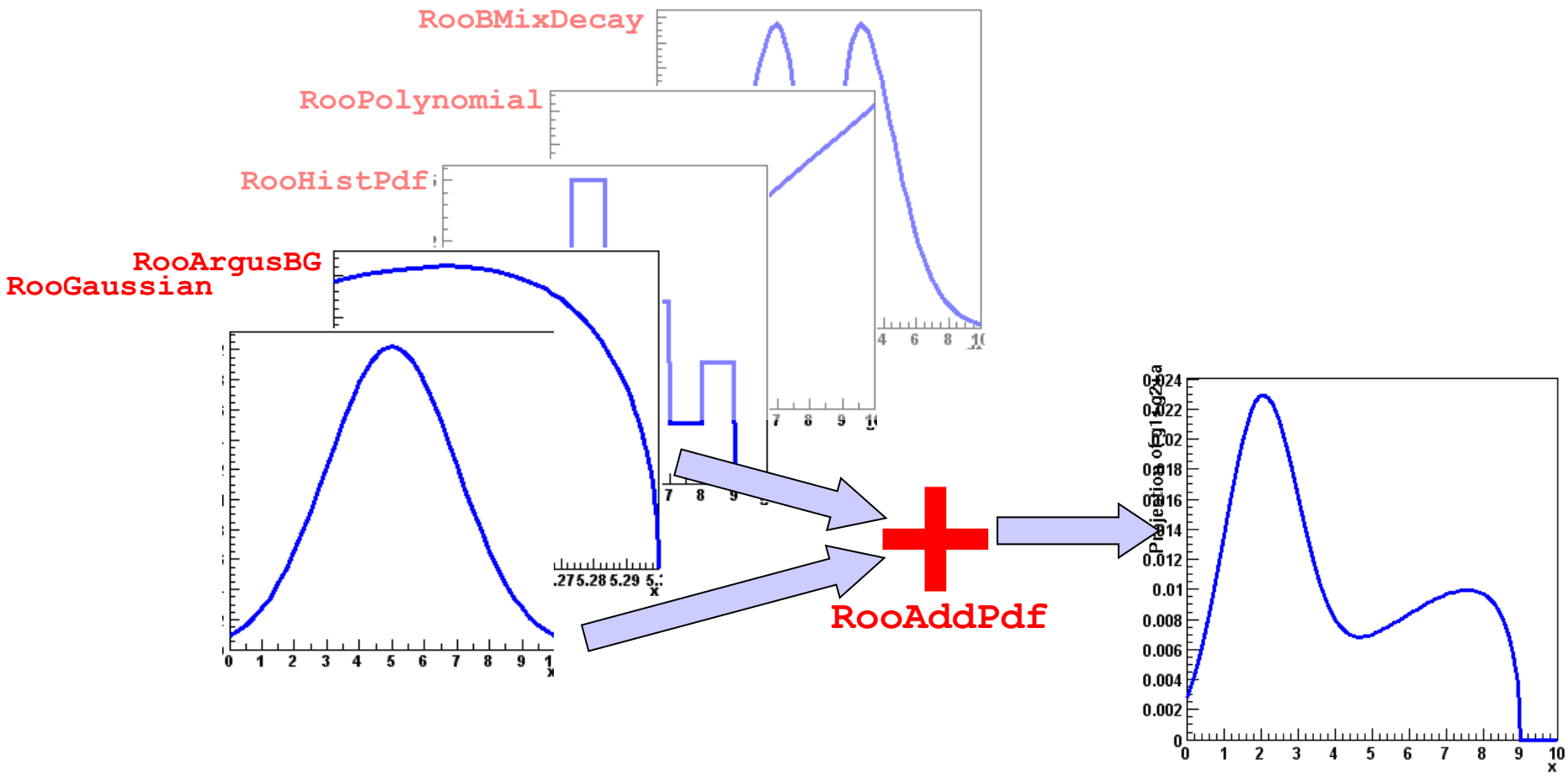


  - Convolution



Model building – (Re)using standard components

- Most realistic models are constructed as the sum of one or more p.d.f.s (e.g. signal and background)
- Facilitated through operator p.d.f **RooAddPdf**



Adding p.d.f.s – Mathematical side

- From math point of view adding p.d.f is simple
  - Two components *F, G*

$$S(x) = fF(x) + (1-f)G(x)$$

  - Generically for *N* components P_0-P_N

$$S(x) = c_0 P_0(x) + c_1 P_1(x) + \ldots + c_{n-1} P_{n-1}(x) + \left(1 - \sum_{i=0,n-1} c_i\right) P_n(x)$$

- For *N* p.d.f.s, there are *N-1* fraction coefficients that should sum to less 1
  - The remainder is by construction 1 minus the sum of all other coefficients

Constructing a sum of p.d.f.s

RooAddPdf constructs the sum of N PDFs with N-1 coefficients:

$$S = c_0 P_0 + c_1 P_1 + c_2 P_2 + \ldots + c_{n-1} P_{n-1} + \left(1 - \sum_{i=0,n-1} c_i\right) P_n$$

Build 2 Gaussian PDFs

Build ArgusBG PDF

```
// Build two Gaussian PDFs
RooRealVar x("x","x",0,10) ;
RooRealVar mean1("mean1","mean of gaussian 1",2) ;
RooRealVar mean2("mean2","mean of gaussian 2",3) ;
RooRealVar sigma("sigma","width of gaussians",1) ;
RooGaussian gauss1("gauss1","gaussian PDF",x,mean1,sigma) ;
RooGaussian gauss2("gauss2","gaussian PDF",x,mean2,sigma) ;

// Build Argus background PDF
RooRealVar argpar("argpar","argus shape parameter",-1.0) ;
RooRealVar cutoff("cutoff","argus cutoff",9.0) ;
RooArgusBG argus("argus","Argus PDF",x,cutoff,argpar) ;

// Add the components
RooRealVar g1frac("g1frac","fraction of gauss1",0.5) ;
RooRealVar g2frac("g2frac","fraction of gauss2",0.1) ;
RooAddPdf  sum("sum","g1+g2+a",RooArgList(gauss1,gauss2,argus),
                               RooArgList(g1frac,g2frac)) ;
```

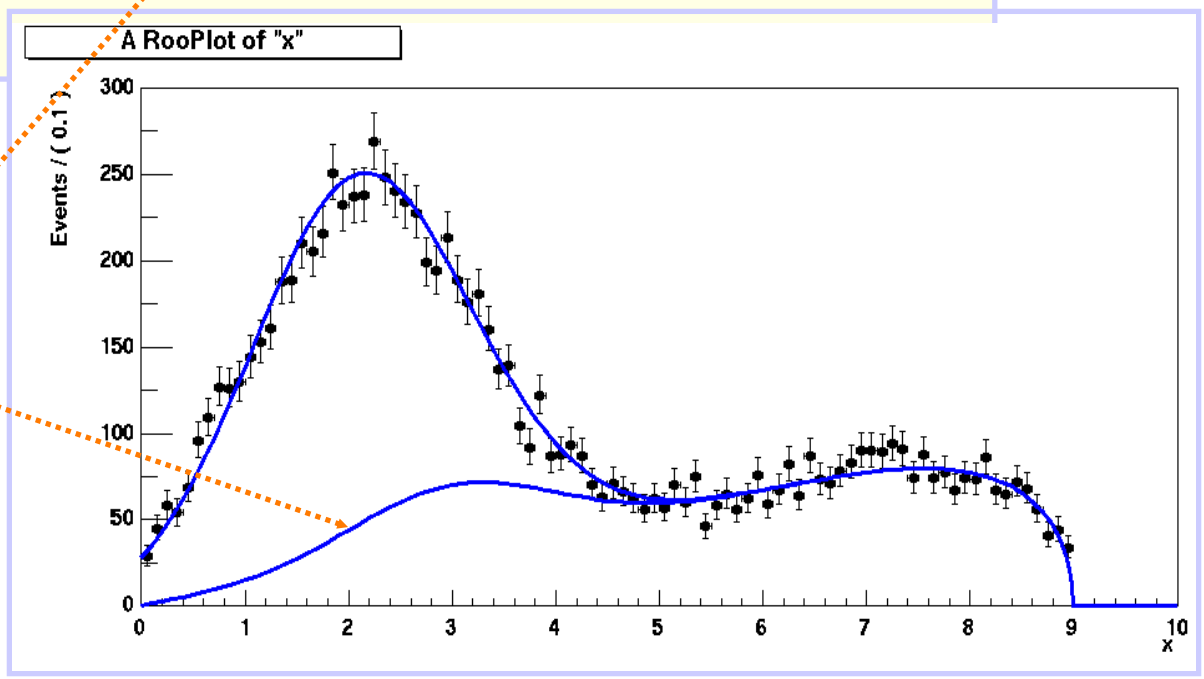
List of PDFs

List of coefficients

Plotting a sum of p.d.f.s, and its components

```
// Generate a toyMC sample
RooDataSet *data =
     sum.generate(x,10000) ;

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

// Plot only argus and gauss2
sum->plotOn(xframe,Components(RooArgSet(argus,gauss2))) ;
xframe->Draw() ;
```

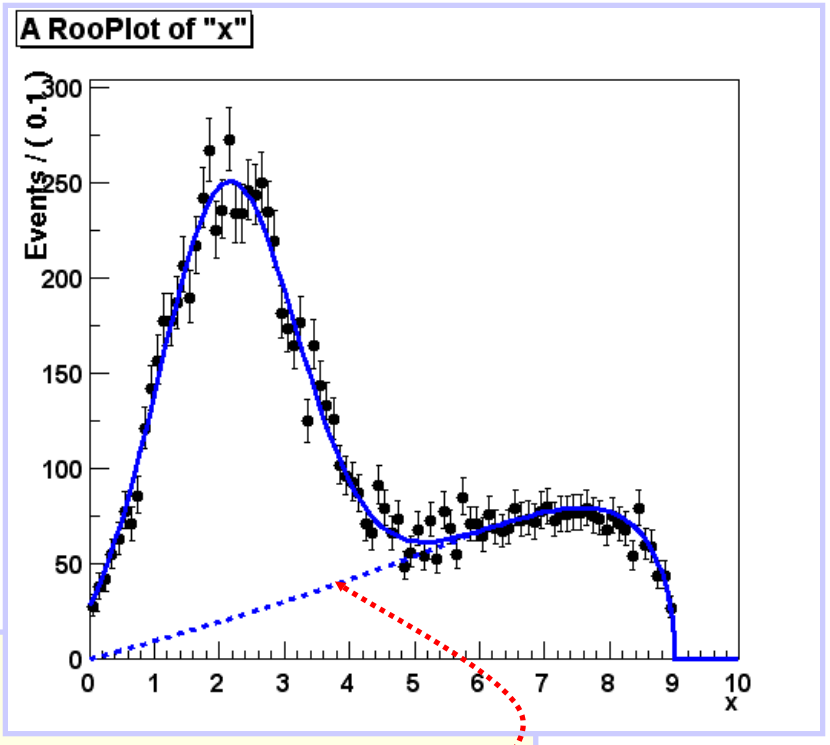
Plot selected components of a **RooAddPdf**



Component plotting - Introduction

- Also special tools for plotting of components in RooPlots
  - Use Method `Components()`

- Example:
  Argus + Gaussian PDF




```
// Plot data and full PDF first

// Now plot only argus component
sum->plotOn(xframe,
            Components(argus), LineStyle(kDashed)) ;
```

Component plotting – Selecting components

There are various ways to select single or multiple components to plot

Can refer to components either by name or reference

```
// Single component selection
pdf->plotOn(frame,Components(argus)) ;
pdf->plotOn(frame,Components("gauss")) ;


// Multiple component selection
pdf->plotOn(frame,Components(RooArgSet(pdfA,pdfB)) ;
pdf->plotOn(frame,Components("pdfA,pdfB")) ;



// Wild card expression allowed
pdf->plotOn(frame,Components("bkgA*,bkgB*")) ;
```

Recursive fraction form of RooAddPdf

- Fitting a sum of >2 p.d.f.s can pose some problems as the sum of the coefficients $f_1 \ldots f_{N-1}$ may become >1
  - This results in a negative remainder component ($\equiv 1-\Sigma_i f_i$)
  - Composite p.d.f may still be positive definite, but interpretation less clear
  - Could set limits on fractions f_i to avoid $\Sigma f_i > 1$ scenario, but where to put limits?
- Viable alternative to write as sum of *recursive* fractions

$$S_2(x) = f_1 P_1(x) + (1-f_1) P_2(x)$$
$$S_3(x) = f_1 P_1(x) + (1-f_1)\big(f_2 P_2(x) + (1-f_2) P_3(x)\big)$$
$$S_4(x) = f_1 P_1(x) + (1-f_1)\big(f_2 P_2(x) + (1-f_2)\big(f_3 P_3(x) + (1-f_3) P_4(x)\big)\big)$$

```
// Add the components with recursive fractions
RooAddPdf  sum("sum","fA*a+(fG*g1+g2)",RooArgList(a,g1,g2),
                                       RooArgList(afrac,gfrac),kTRUE) ;
```

参考 tutorials/roofit/rf201_composite.C

Extended p.d.f form of RooAddPdf

- If extended ML term is introduced, we can fit expected number of events (N_{exp}) in addition to shape parameters
- In case of sum of p.d.f.s it is convenient to *re-parameterize* sum of p.d.f.s.

$$\begin{pmatrix} f_{sig} \\ N_{\exp} \end{pmatrix} \Rightarrow \begin{pmatrix} N_{sig} \equiv f_{sig} N_{\exp} \\ N_{bkg} \equiv (1 - f_{sig}) N_{\exp} \end{pmatrix}$$

- This transformation is applied automatically in **RooAddPdf** if equal number of p.d.f.s and coefs are given

```
RooRealVar nsig("nsig","number of signal events",100,0,10000) ;
RooRealVar nbkg("nbkg","number of backgnd events",100,0,10000) ;
RooAddPdf sume("sume","extended sum pdf",RooArgList(gauss,argus),
                                         RooArgList(nsig,nbkg)) ;
```

General features of extended p.d.f.s

- Extended term –log(Poisson(N_{obs},N_{exp})) is not added by default to likelihood
  - Use the `Extended()` argument to fit to have it added

```
// Regular maximum likelihood fit
pdf.fitTo(*data) ;

// Extended maximum likelihood fit
pdf.fitTo(*data,Extended(kTRUE)) ;
```

- If p.d.f. is extended, N_{exp} is default number of events to generate

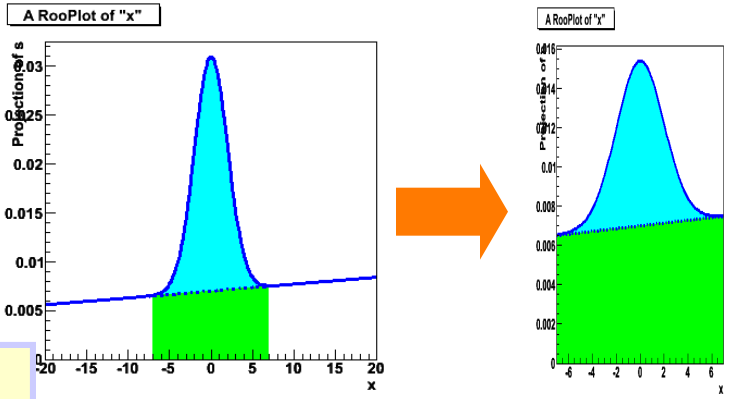
```
// Generate pdf.expectedEvents() events
RooDataSet* data = pdf.generate(x) ;

// Generate 1000 events
RooDataSet* data = pdf.generate(x,1000) ;
```

How it works – Normalization of **RooAddPdfs**

- Since all component p.d.f.s are normalized, resulting sum of p.d.f.s is automatically normalized
  - As long as sum of coefficients is 1, which is automatically enforced

$$S(x) = c_0 P_0(x) + c_1 P_1(x) + \ldots + c_{n-1} P_{n-1}(x) + \left(1 - \sum_{i=0,n-1} c_i\right) P_n(x)$$

  - But note that fraction parameter multiplies *normalized* p.d.f.s
- Interpretation of fraction depends on range of observables (and number of observables for >1D)
  - If range of observable is changed and fraction parameter is same, the shape effectively different
  - Can mitigate this by specifying a fixed reference range for fraction interpretation



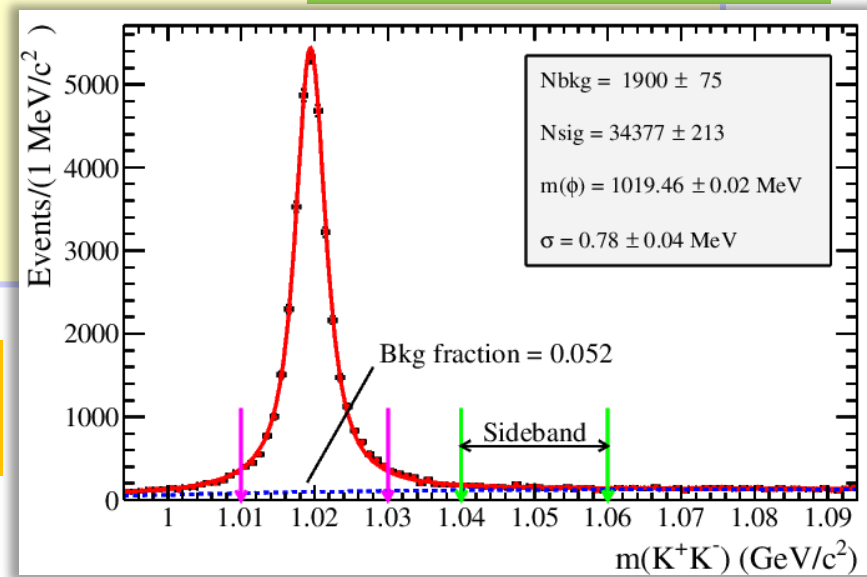

```
x.setRange("ref",-20,20) ;
pdf->setAddCoefRange("ref") ;
```

参考 tutorials/roofit/rf203_ranges.C，rf204_extrangefit.C，rf312_multirangefit.C

Extended ML fit with range definition

```
RooRealVar x("x", "m(K^{+}K^{-})", 0.994,1.094);
RooRealVar mass("Xmass", "Tmass", 1.02, 1.01 , 1.03);
RooRealVar width("Xwidth", "Twidth", 0.00426, 0.00 , 0.00);
RooRealVar sigma("Xsigma", "Tsigma", 0.00, 0.00 , 0.10);
RooVoigtian sig("Voigtian", "VTp.d.f", x, mass, width, sigma);
RooChebychev bkg("bkg","bkg",m34,RooArgList(c0,c1,c2));
double nmax = mkk->numEntries()+100;
RooRealVar nsig("nsig","#signal events",    nmax*0.4,0,nmax);
RooRealVar nbkg("nbkg","#background events",nmax*0.6,0,nmax);
m34.setRange("cut",1.01,1.03);
RooExtendPdf sige1 ("sige1","sige1",sig, nsig,"cut");
RooExtendPdf bkge1 ("bkge1","bkge1",bkg, nbkg,"cut");
RooAddPdf sum("sum","g+b",RooArgList(sige1,bkge1));
RooFitResult* r =sum.fitTo(*mkk,
RooFit::Extended(kTRUE),RooFit::Save(kTRUE));
r->Print("v");
RooPlot* phiplot = x.frame(100);
phiplot->Draw();
```

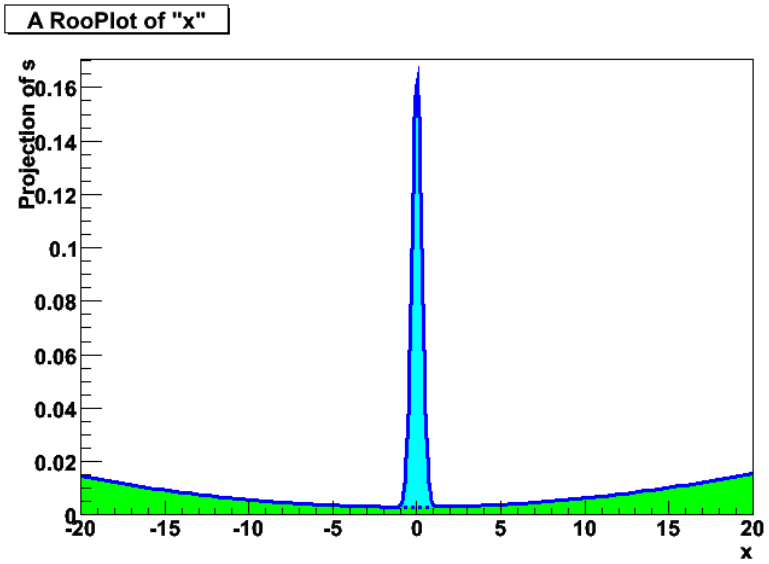
拟合得到的Nsig和Nbkg为信号区间1.01-1.03的事例数



类似的拟合脚本，参考
$ROOTSYS/tutorials/roofit/rf204_extrangefit.C

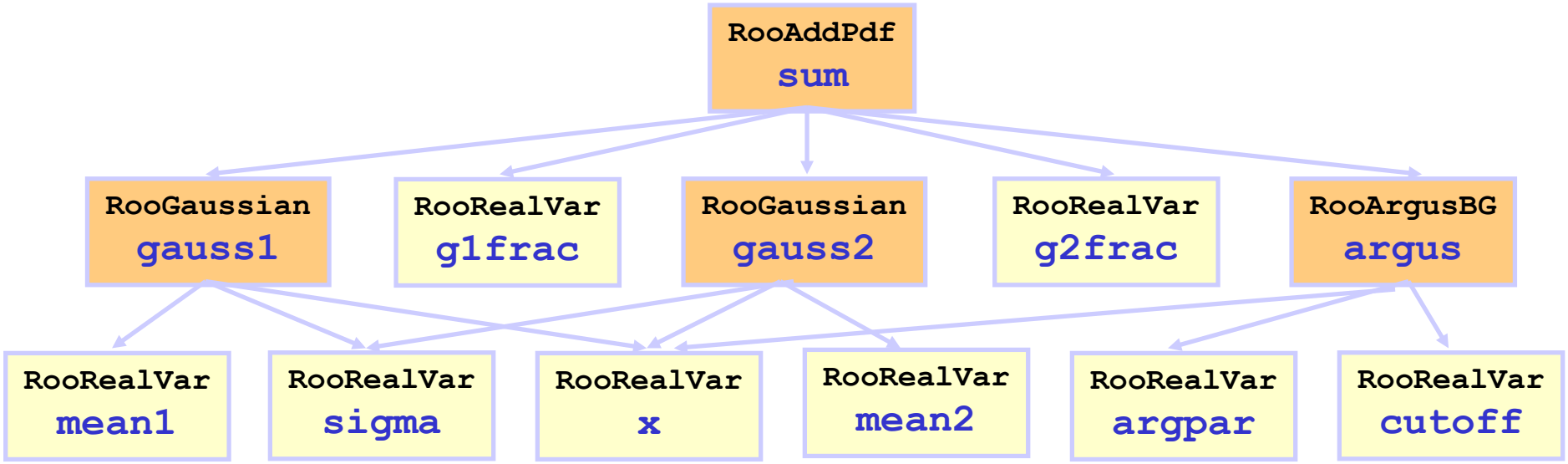
How it works – event generation of RooAddPdf

- Composite event generation algorithm of **`RooAddPdf`**
  - Choose randomly a component to generate (probability proportional to coefficient fractions)
  - Delegate generation of observable to algorithm of component p.d.f.
- Allows to efficiently handle sum of p.d.f with very different shapes in most cases
  - Example:
    Blue Gaussian
    (internal generator)
    plus Green Polynomial
    (accept/reject)



Dealing with composite p.d.f.s

- A RooAddPdf is an example of a composite p.d.f
  - The value of the sum is represented by a *tree* of components



  - The compositeness of a p.d.f. is completely transparent to most high-level operations
  - Can e.g. do `sum->fitTo(*data)` or `sum->generate(x,1000)` without being aware of composite nature of p.d.f.

Dealing with composite p.d.f.s

- The observables reported by a composite p.d.f and the 'leaf' of the expression tree
  - For example, request for list of parameters of composite sum, will return parameters of components of sum

```
RooArgSet *paramList = sum.getParameters(data) ;
paramList->Print("v") ;
RooArgSet::parameters:
  1) RooRealVar::argpar : -1.00000 C
  2) RooRealVar::cutoff :  9.0000 C
  3) RooRealVar::g1frac :  0.50000 C
  4) RooRealVar::g2frac :  0.10000 C
  5) RooRealVar::mean1  :  2.0000 C
  6) RooRealVar::mean2  :  3.0000 C
  7) RooRealVar::sigma  :  1.0000 C
```

- In general, composite p.d.f.s work *exactly the same* as basic p.d.f.s.

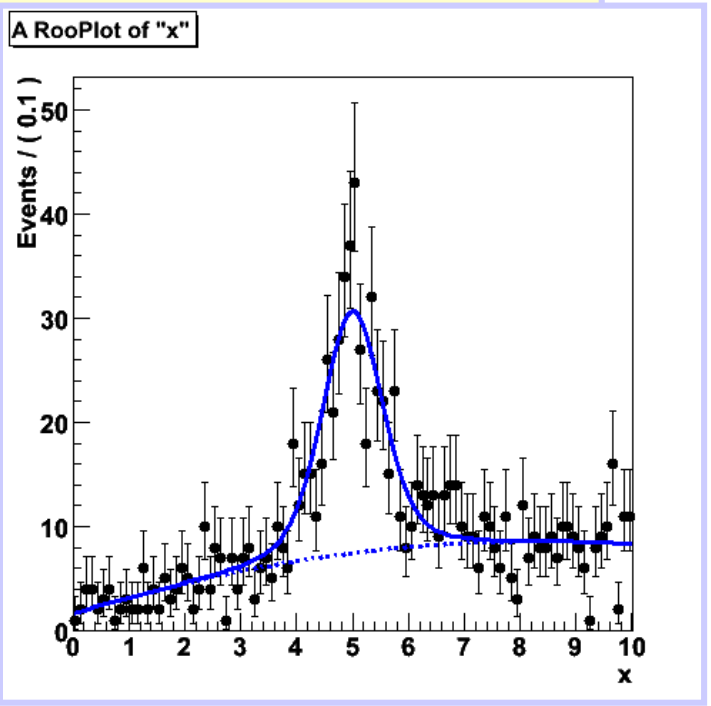
Visualization tools for composite objects

- Special tools exist to visualize the tree structure of composite objects
  - On the command line

```
Root> sum.Print("t") ;
0x927b8d0 RooAddPdf::sum (g1+g2+a)  [Auto]
  0x9254008 RooGaussian::gauss1 (gaussian PDF)  [Auto] V
    0x9249360 RooRealVar::x (x) V
    0x924a080 RooRealVar::mean1 (mean of gaussian 1) V
    0x924d2d0 RooRealVar::sigma (width of gaussians) V
  0x9267b70 RooRealVar::g1frac (fraction of gauss1) V
  0x9259dc0 RooGaussian::gauss2 (gaussian PDF)  [Auto] V
    0x9249360 RooRealVar::x (x) V
    0x924cde0 RooRealVar::mean2 (mean of gaussian 2) V
    0x924d2d0 RooRealVar::sigma (width of gaussians) V
  0x92680e8 RooRealVar::g2frac (fraction of gauss2) V
  0x9261760 RooArgusBG::argus (Argus PDF)  [Auto] V
    0x9249360 RooRealVar::x (x) V
    0x925fe80 RooRealVar::cutoff (argus cutoff) V
    0x925f900 RooRealVar::argpar (argus shape parameter) V
    0x9267288 RooConstVar::0.500000 (0.500000) V
```

Putting it all together – Extended unbinned ML Fit to signal and background

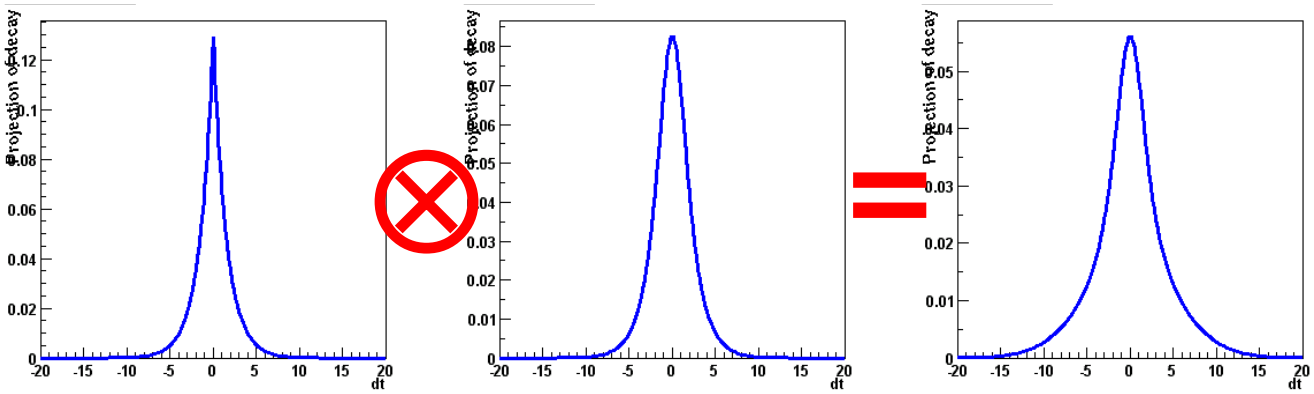
```
// Declare observable x
RooRealVar x("x","x",0,10) ;

// Creation of 'sig', 'bkg' component p.d.f.s omitted for clarity

// Model = Nsig*sig + Nbkg*bkg (extended form)
RooRealVar nsig("nsig","#signal events",300,0.,2000.) ;
RooRealVar nbkg("nbkg","#background events",700,0,2000.) ;
RooAddPdf  model("model","sig+bkg",RooArgList(sig,bkg),RooArgList(nsig,nbkg)) ;

// Generate a data sample of Nexpected events
RooDataSet *data = model.generate(x) ;

// Fit model to data
model.fitTo(*data, Extended(kTRUE)) ;

// Plot data and PDF overlaid
RooPlot* xframe = x.frame() ;
data->plotOn(xframe) ;
model.plotOn(xframe) ;
model.plotOn(xframe,Components(bkg),
             LineStyle(kDashed)) ;
xframe->Draw() ;
```



参考 tutorials/roofit/rf202_extendedmlfit.C

Building models – Convolutions

- Many experimental observable quantities are well described by convolutions
  - Typically physics distribution smeared with experimental resolution (e.g. for B0 → J/ψ K_S exponential decay distribution smeared with Gaussian)



  - By explicitly describing observed distribution with a convolution p.d.f can disentangle detector and physics
    - To the extent that enough information is in the data to make this possible

Mathematical introduction & Numeric issues

- Mathematical form of convolution
  - Convolution of two functions

$$f(x) \otimes g(x) = \int_{-\infty}^{+\infty} f(x) g(x - x') dx'$$

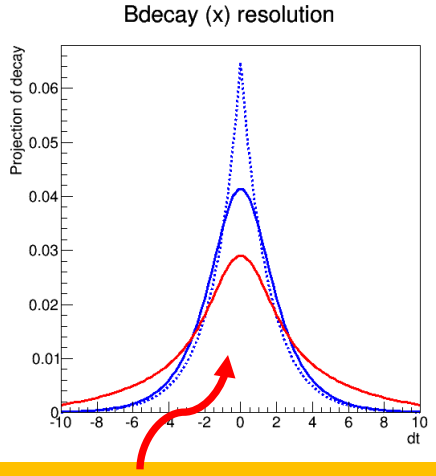
  - Convolution of two normalized p.d.f.s itself is *not* automatically normalized, so expression for convolution p.d.f is

$$F(x) \otimes G(x) = \frac{\int_{-\infty}^{+\infty} F(x) G(x - x') dx'}{\int_{x_{\min}}^{x_{\max}} \int_{-\infty}^{+\infty} F(x) G(x - x') dx' dx}$$

  - Because of (multiple) integrations required convolution are difficult to calculate
  - Convolution integrals are best done analytically, but often not possible

Convolution operation in RooFit

- RooFit has several options to construct convolution p.d.f.s
  - Class **RooNumConvPdf** – 'Brute force' numeric calculation of convolution (and normalization integrals)
  - Class **RooFFTConvPdf** – Calculate convolution integral using discrete FFT technology in fourier-transformed space.
  - Bases classes **RooAbsAnaConvPdf**, **RooResolutionModel**. Framework to construct analytical convolutions (with implementations mostly for B physics)
  - Class **RooVoigtian** – Analytical convolution of non-relativistic Breit-Wigner shape with a Gaussian
- All convolution in one dimension so far
  - N-dim extension of **RooFFTConvPdf** foreseen in future



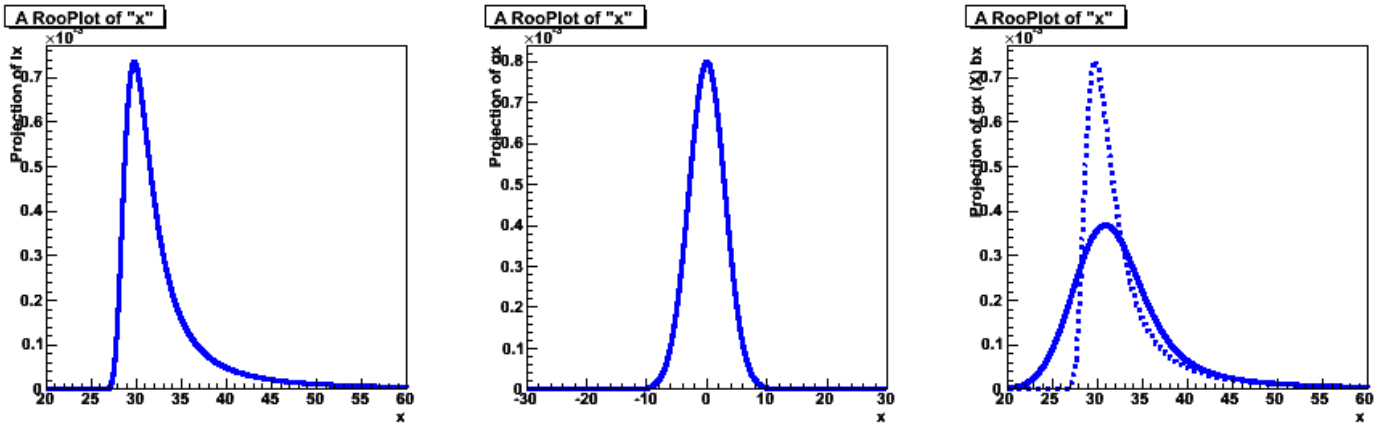
参考 tutorials/roofit/rf209_anaconv.C
(分别卷积delta function、 Gaussian和double Gaussian)

Numeric convolutions – Class RooNumConvPdf

- Properties of **RooNumConvPdf**
  - Can convolve *any* two input p.d.f.s
  - Uses special numeric integrator that can compute integrals in [-∞,+∞] domain
  - Slow (very!) especially if requiring sufficient numeric precision to allow use in MINUIT (requires ~10^{-7} estimated precision). *Converge problems in MINUIT if precision is insufficient*

```
// Construct landau (x) gauss
RooNumConvPdf lxg("lxg","landau (X) gauss",t,landau,gauss) ;
```

Landau *Gauss* *Landau ⊗ Gauss*



Numeric convolutions – Class RooFFTConvPdf

- Properties of **RooFFTConvPdf**
  - Uses convolution theorem to compute *discrete* convolution in Fourier-Transformed space.
  - Transforms both input p.d.f.s with forward FFT

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn} \qquad k = 0, \ldots, N-1$$

(x_i are sampled values of p.d.f)

  - Makes use of Circular Convolution Theorem in Fourier Space

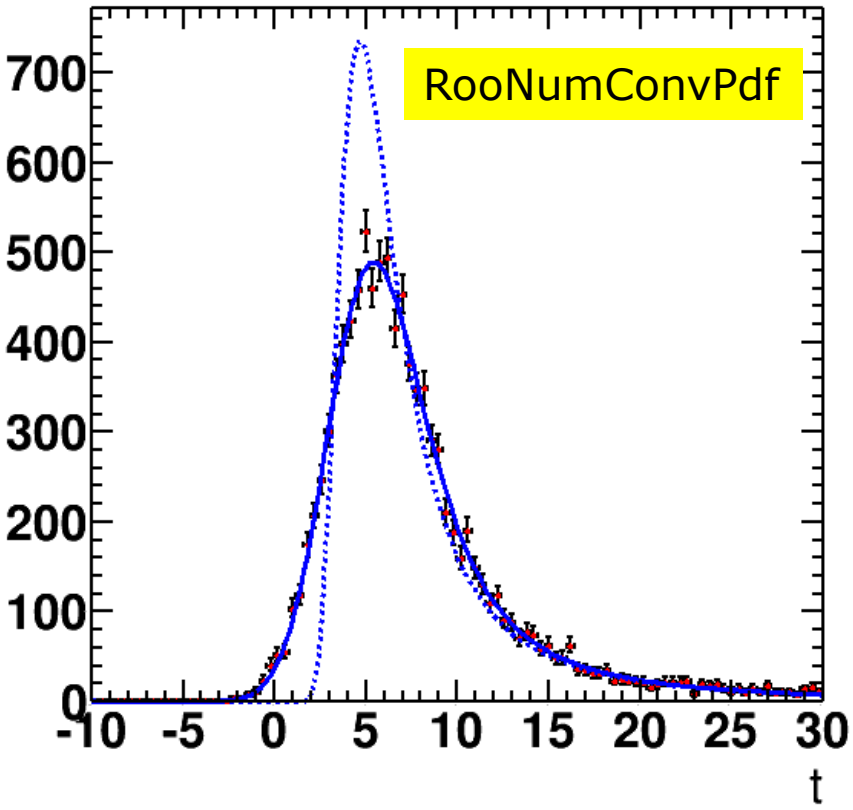
$$\begin{aligned}\mathcal{F}^{-1}\{\mathbf{X}\cdot\mathbf{Y}\}_n &= \sum_{l=0}^{N-1} x_l \sum_{m=-\infty}^{\infty} y_m \left(\sum_{p=-\infty}^{\infty} \delta_{m(n-l-pN)} \right) \\ &= \sum_{l=0}^{N-1} x_l \sum_{p=-\infty}^{\infty} \left(\sum_{m=-\infty}^{\infty} y_m \cdot \delta_{m(n-l-pN)} \right) \\ &= \sum_{l=0}^{N-1} x_l \left(\sum_{p=-\infty}^{\infty} y_{n-l-pN} \right) \overset{\text{def}}{=} (\mathbf{x} * \mathbf{y_N})_n \ ,\end{aligned}$$

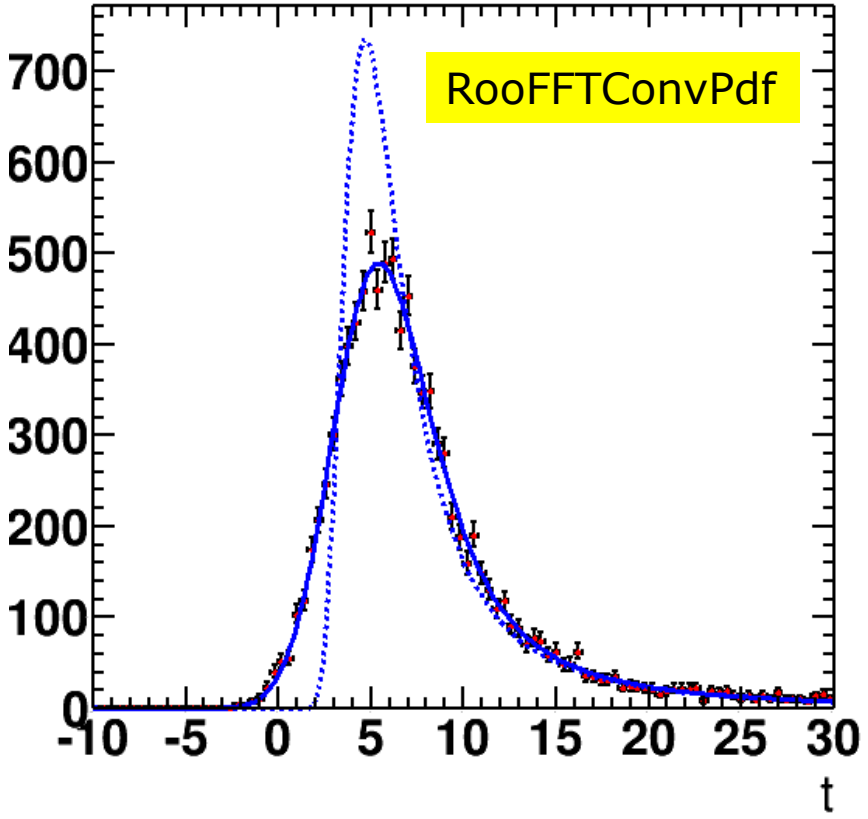
  - *Convolution can be computed in terms of products of Fourier components (easy)*
  - Apply inverse Fourier transform to obtained convoluted p.d.f in space domain

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn} \qquad n = 0, \ldots, N-1.$$

参考 tutorials/roofit/rf208_convolution.C (用**RooFFTConvPdf**卷积)

RooNumConvPdf and RooFFTConvPdf





参考 tutorials/roofit/rf208_convolution.C
(分别用**RooNumConvPdf**和**RooFFTConvPdf**卷积)

Numeric convolutions – Class RooFFTConvPdf

- Fourier transforms calculated by FFTW3 package
  - Interfaced in ROOT through **`TVirtualFFT`** class
- About 100x faster than **`RooNumConvPdf`**
  - Also much better numeric stability (c.f. MINUIT converge)
  - Choose sufficiently large number of samplings to obtain smooth output p.d.f
  - CPU time is **not** proportional to number of samples, e.g. 10000 bins works fine in practice
- Note: p.d.f.s are not sampled from $[-\infty,+\infty]$, but from $[x_{min},x_{max}]$
- Note: p.d.f is explicitly treated as *cyclical* beyond range
  - Excellent for cyclical observables such as angles
  - If p.d.f converges to zero towards both ends of range if non-cyclical observable, all works out fine
  - If p.d.f does not converge to zero towards domain end, cyclical leakage will occur

Framework for analytical calculations of convolutions demo6.cc

- Convoluted PDFs that can be written if the following form can be used in a very modular way in RooFit

$$P(dt,...) = \sum_k c_k(...)\left(f_k(dt,...) \otimes R(dt,...)\right)$$

coefficient

'basis function'

resolution function

Example: B^0 decay with mixing

$$c_0 = 1 \pm \Delta w, \qquad f_0 = e^{-|t|/\tau}$$

$$c_1 = \pm(1-2w), \qquad f_1 = e^{-|t|/\tau}\cos(\Delta m \cdot t)$$

Convoluted PDFs

- Physics model and resolution model are implemented separately in RooFit

Implements $f_i(dt,...) \otimes R(dt,...)$
Also a PDF by itself

RooResolutionModel

$$P(dt,...) = \sum_k c_k(...)\left(f_k(dt,...) \otimes R(dt,...)\right)$$

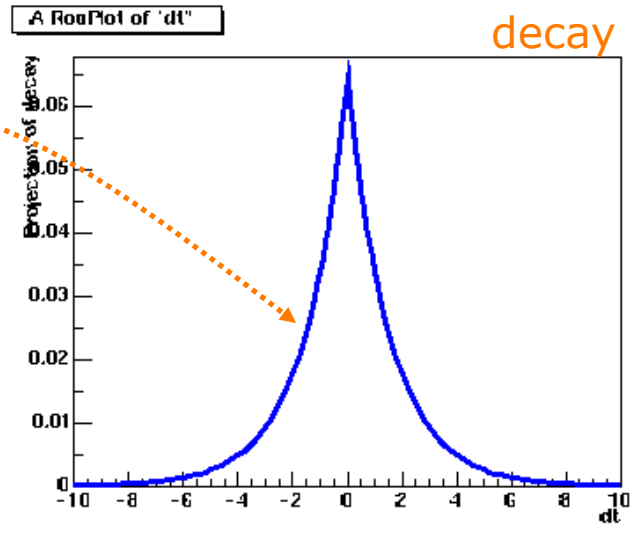
RooConvolutedPdf (physics model)

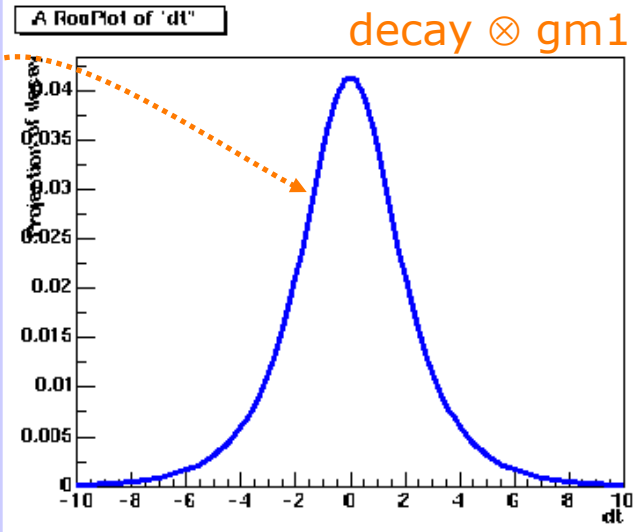
Implements $\boldsymbol{c_k}$
Declares list of $\boldsymbol{f_k}$ needed

User can choose combination of physics model and resolution model at run time
(Provided resolution model implements all f_k declared by physics model)

Convoluted PDFs

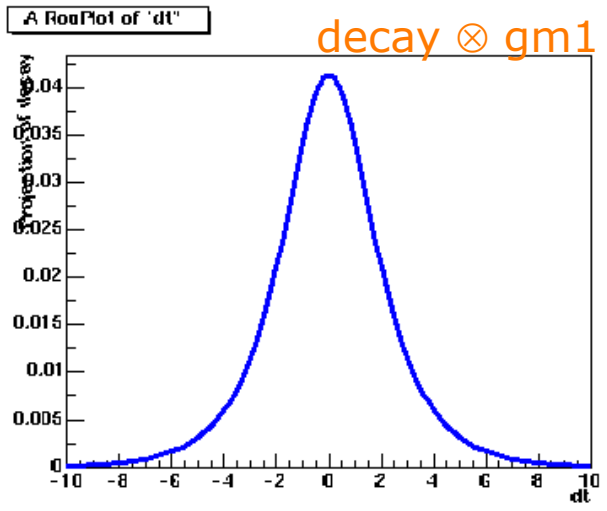
```
RooRealVar dt("dt","dt",-10,10) ;
RooRealVar tau("tau","tau",1.548) ;

// Truth resolution model
RooTruthModel tm("tm","truth model",dt) ;

// Unsmeared decay PDF
RooDecay decay_tm("decay_tm","decay",
   dt,tau,tm,RooDecay::DoubleSided) ;

// Gaussian resolution model
RooRealVar bias1("bias1","bias1",0) ;
RooRealVar sigma1("sigma1","sigma1",1) ;
RooGaussModel gm1("gm1","gauss model",
                  dt,bias1,sigma1) ;

// Construct a decay (x) gauss PDF
RooDecay decay_gm1("decay_gm1","decay",
    dt,tau,gm1,RooDecay::DoubleSided) ;
```

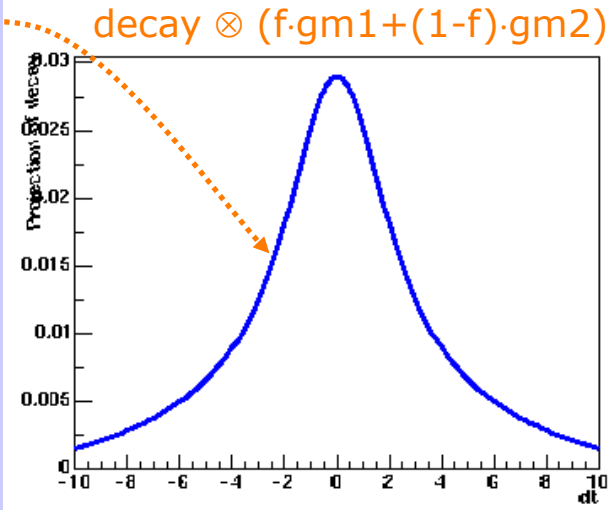




Composite Resolution Models: **RooAddModel**

```
//... (continued from last page)

// Wide gaussian resolution model
RooRealVar bias2("bias2","bias2",0) ;
RooRealVar sigma2("sigma2","sigma2",5) ;
RooGaussModel gm2("gm2","gauss model 2"
                  ,dt,bias2,sigma2) ;

// Build a composite resolution model
RooRealVar f("f","fraction of gm1",0.5)
RooAddModel gmsum("gmsum","gm1+gm2",
        RooArgList(gm1,gm2),f) ;

// decay (x) (gm1 + gm2)
RooDecay decay_gmsum("decay_gmsum",
               "decay",dt,tau,gmsum,
                RooDecay::DoubleSided) ;
```

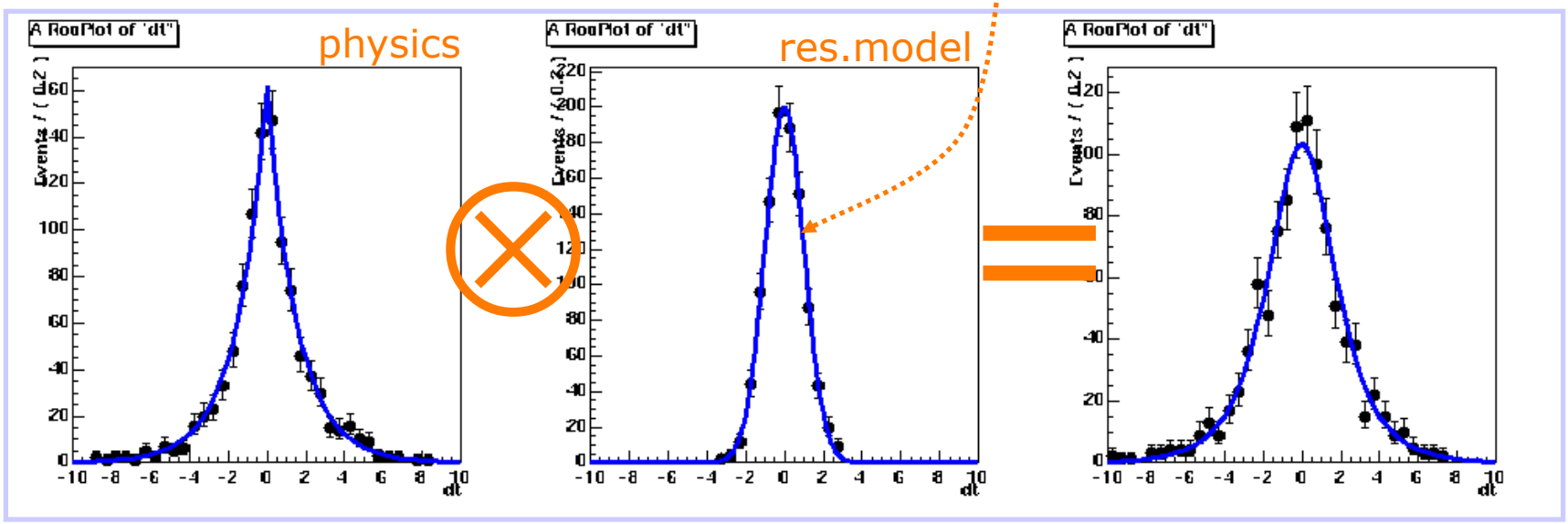




→RooAddModel works like RooAddPdf

Resolution models

- Currently available resolution models
  - **`RooGaussModel`** – Gaussian with bias and sigma
  - **`RooGExpModel`** – Gaussian (X) Exp with sigma and lifetime
  - **`RooTruthModel`** – Delta function
- A **`RooResolutionModel`** is also a PDF
  - You can use the same resolution model
    you use to convolve your physics PDFs to fit to MC residuals



How it works – generating events from convolution p.d.f.s

- A very efficient implementation of event generation is possible

$$x_{P \otimes R} = x_P + x_R$$

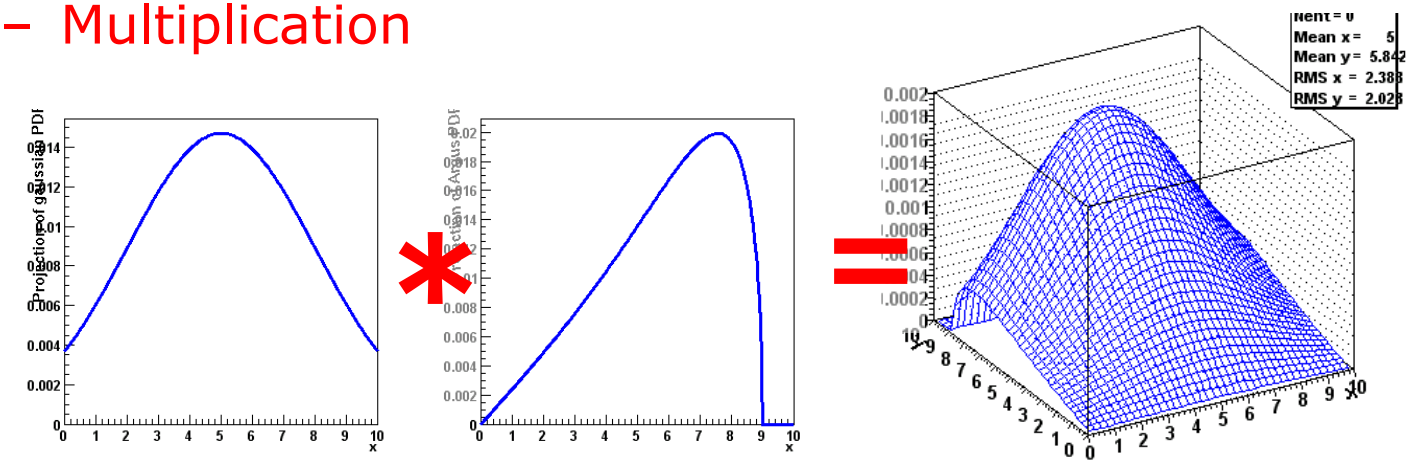
  - Reflect 'smearing' view of convolution
  - Very fast as no computation of convolution integrals is required
  - But only if both input p.d.f.s can generate observables in the range [-∞,+∞] which is not possible with accept/reject so this can only be done if both input p.d.f.s have an internal generator implementation
  - If above conditions are not met, automatic fallback solution is to perform accept/reject sampling on convoluted p.d.f. shape

4 Multidimensional models

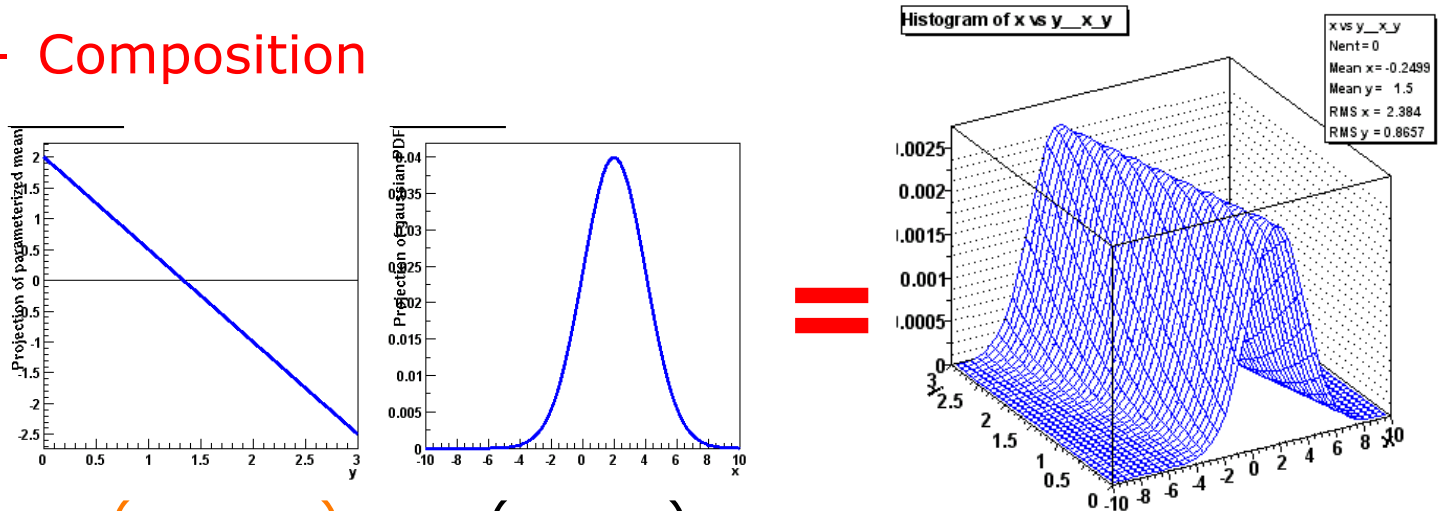
- *Uncorrelated products of p.d.f.s*
- *Using composition to p.d.f.s with correlation*
- *Products of conditional and plain p.d.f.s*

Building realistic models

- Multiplication



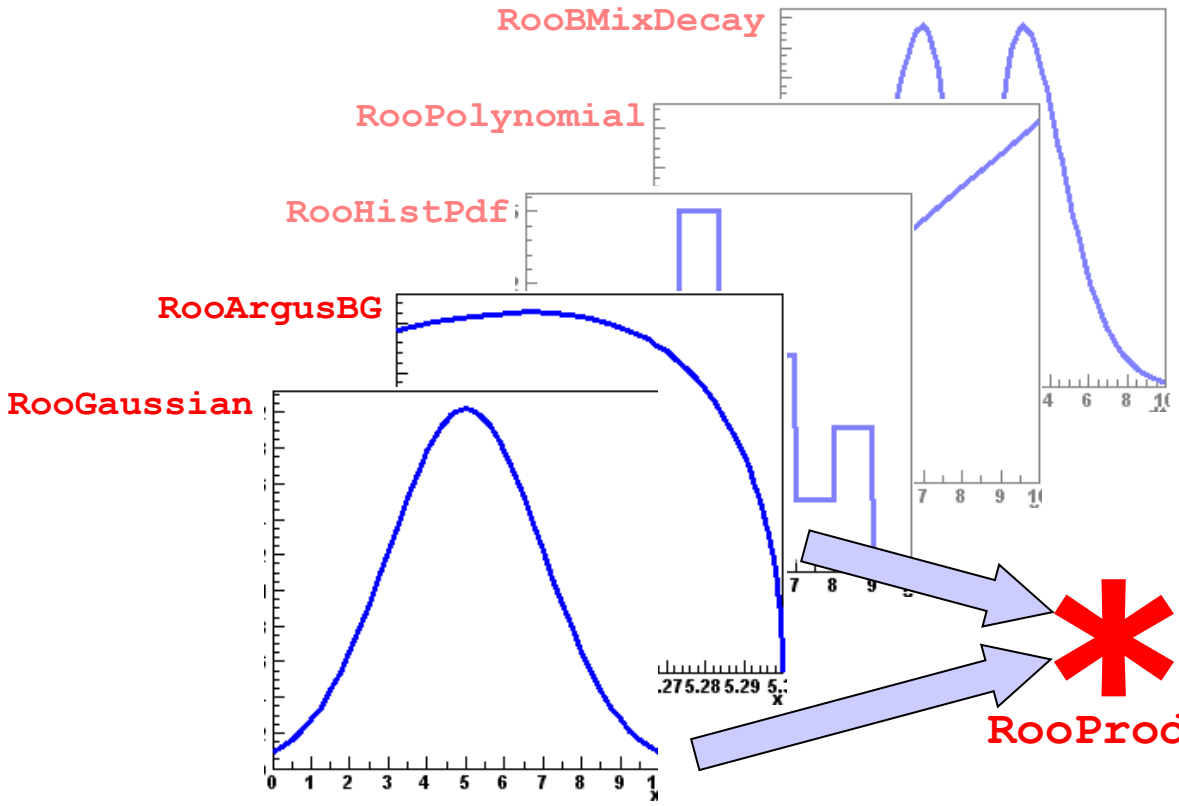
- Composition



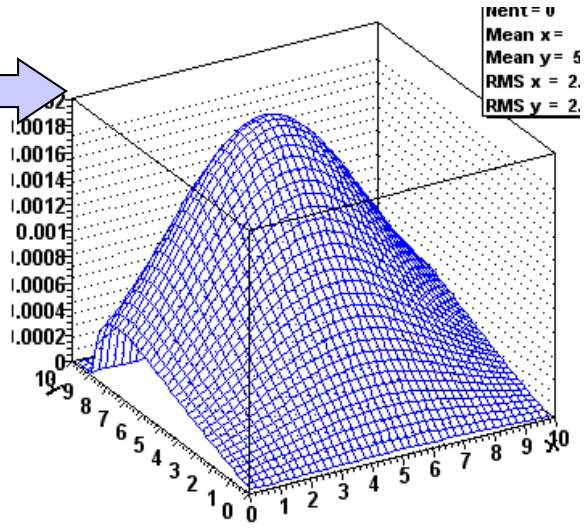
$m(y;a_0,a_1)$ $g(x;m,s)$ $g(x,y;a0,a1,s)$

Possible in *any* PDF
No explicit support in PDF code needed

Model building – Products of uncorrelated p.d.f.s



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



Uncorrelated products – Mathematics and constructors

- Mathematical construction of products of uncorrelated p.d.f.s is straightforward

2D

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

nD

$$H(x^{\{i\}}) = \prod_i F^{\{i\}}(x^{\{i\}})$$

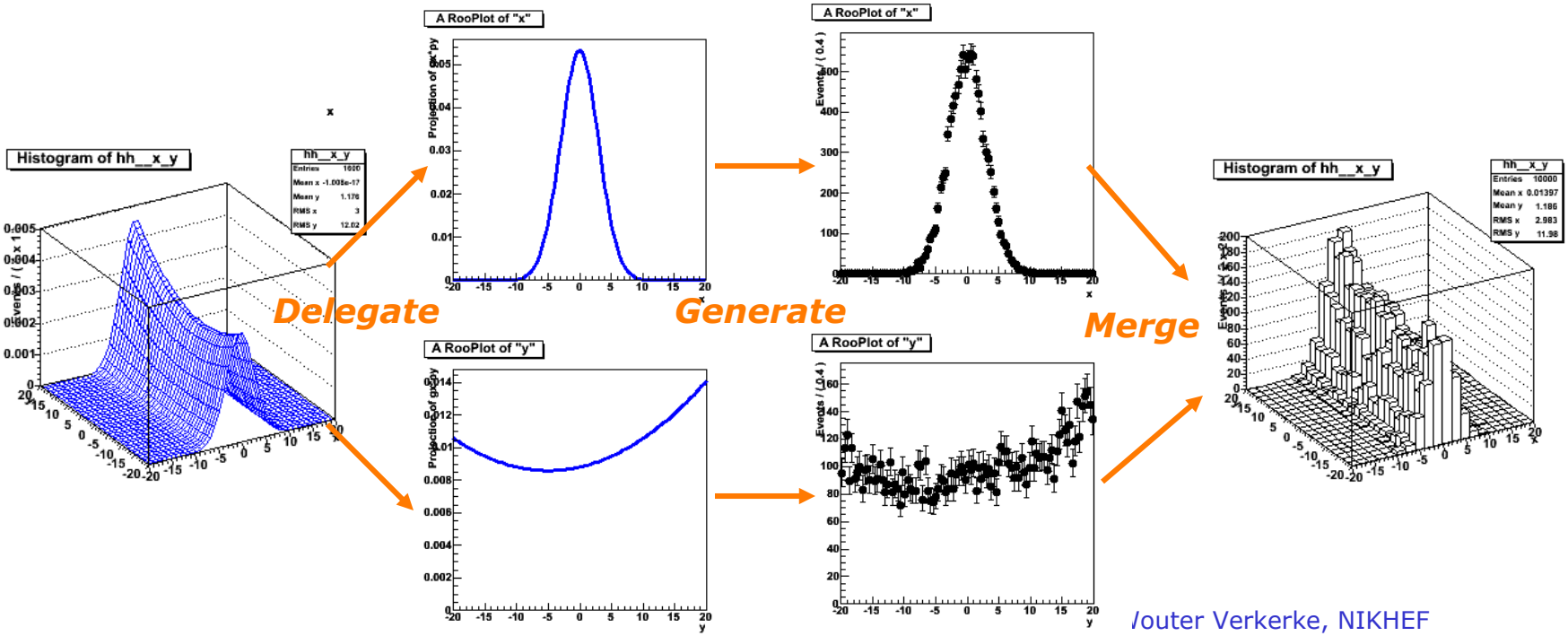
  - No explicit normalization required → If input p.d.f.s are unit normalized, product is also unit normalized
    (this is true *only* because of the absence of correlations)

- Corresponding RooFit operator p.d.f. is **`RooProdPdf`**
  - Returns product of *normalized* input p.d.f values

```
RooGaussian gx("gx","gaussian PDF",x,meanx,sigmax) ;
RooGaussian gy("gy","gaussian PDF",y,meany,sigmay) ;

// Multiply gaussx and gaussy into a two-dimensional p.d.f. gaussxy
RooProdPdf  gaussxy("gxy","gx*gy",RooArgList(gx,gy)) ;
```

How it work – event generation on uncorrelated products

- If p.d.f.s are uncorrelated, each observable can be generated separately
  - Reduced dimensionality of problem (important for e.g. accept/reject sampling)
  - Actual event generation delegated to component p.d.f (can e.g. use internal generator if available)
  - **`RooProdPdf`** just aggregates output in single dataset



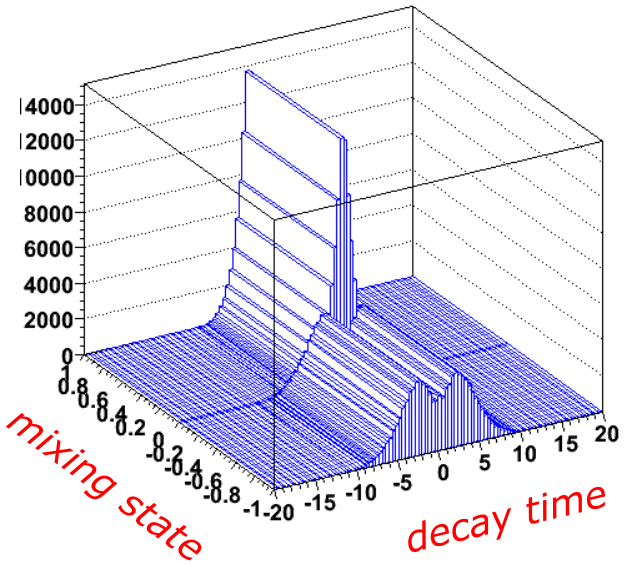
Fundamental multi-dimensional p.d.fs

- It also possible define multi-dimensional p.d.f.s that do not arise through a product construction
  - For example

```
RooGenericPdf gp("gp","sqrt(x+y)*sqrt(x-y)",RooArSet(x,y)) ;
```

  - But *usually n-dim p.d.f.s are constructed more intuitively* through product constructs. Also correlations can be introduced efficiently (more on that in a moment)

- Example of fundamental 2-D B-physics p.d.f. **RooBMixDecay**
  - Two observables:
    decay time (t, continuous)
    mixingState (m, discrete [-1,+1])

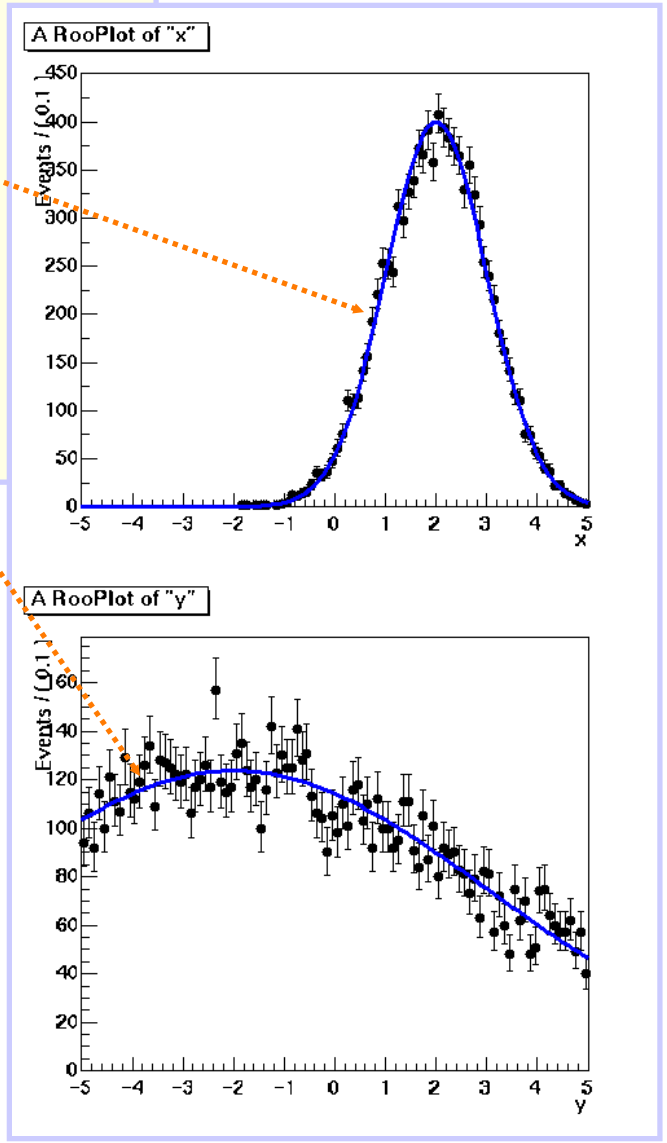


Plotting multi-dimensional PDFs

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

c->cd(2) ;
RooPlot* yframe = y.frame() ;
data->plotOn(yframe) ;
prod->plotOn(yframe) ;
yframe->Draw() ;
```

$$f(x) = \int pdf(x,y)dy$$

$$f(y) = \int pdf(x,y)dx$$



-Plotting a dataset D(x,y) versus x represents a *projection over y*

-To overlay PDF(x,y), you must plot *Int(dy)PDF(x,y)*

-RooFit automatically takes care of this!

- RooPlot remembers dimensions of plotted datasets

Projecting out hidden dimensions

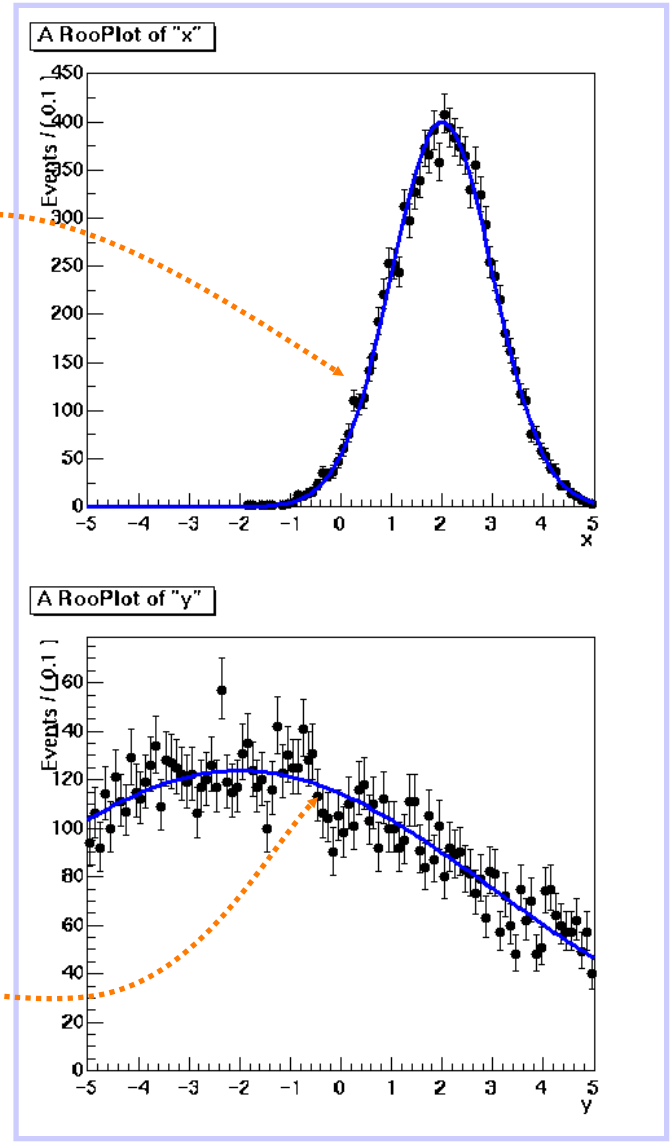
- Example in 2 dimensions
  - 2-dim dataset D(x,y)
  - 2-dim PDF P(x,y)=gauss(x)*gauss(y)

- 1-dim plot versus x

$$P_p(x) = \frac{\int p(x,y)dy}{\int p(x,y)dxdy}$$

- 1-dim plot versus y

$$P_p(y) = \frac{\int p(x,y)dx}{\int p(x,y)dxdy}$$



RooProdPdf automatic optimization for uncorrelated terms

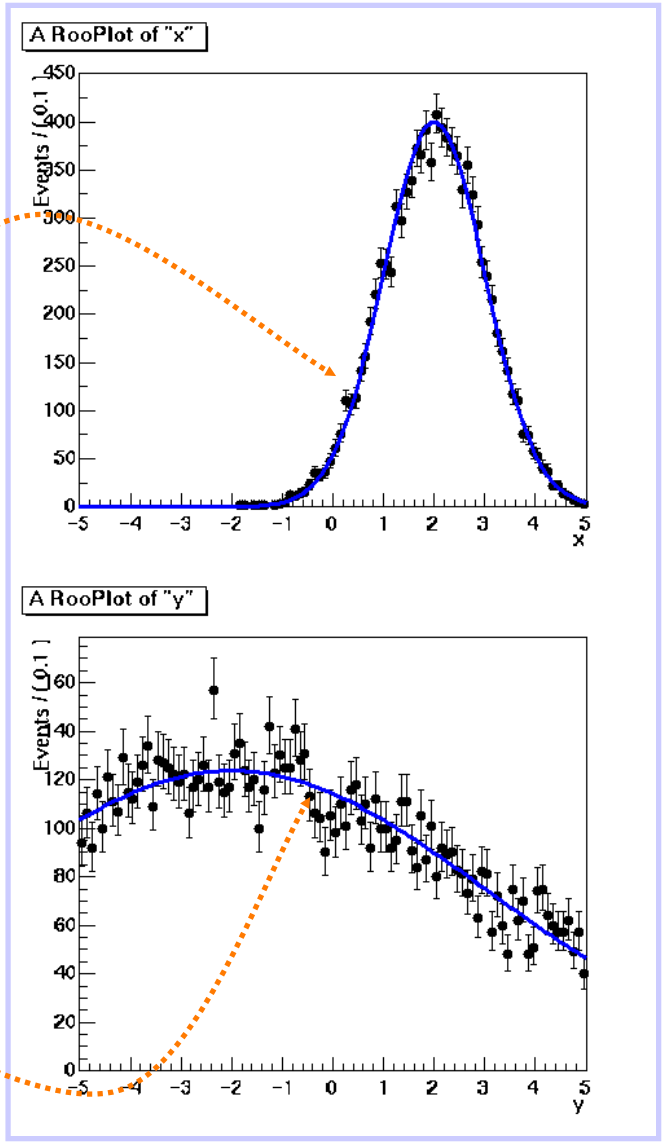
- Example in 2 dimensions
  - 2-dim dataset D(x,y)
  - 2-dim PDF P(x,y)=gaus(x)*gauss(y)

- 1-dim plot versus x

$$P_p(x) = \frac{\int g(x)g(y)dy}{\int g(x)g(y)dxdy} = \frac{g(x)\int g(y)dy}{\int g(x)dx \int g(y)dy} = \frac{g(x)}{\int g(x)dx}$$

- 1-dim plot versus y

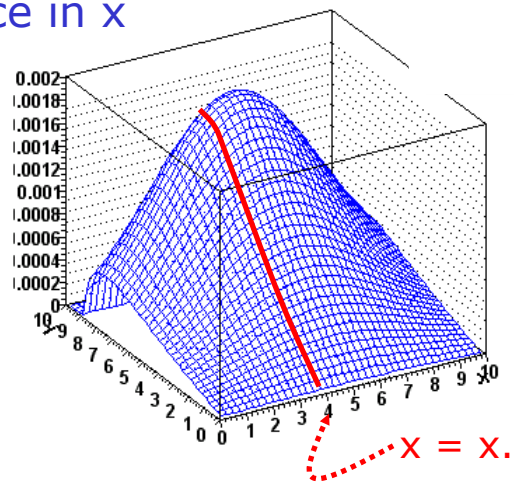
$$P_p(y) = \frac{\int g(x)g(y)dx}{\int g(x)g(y)dxdy} = \frac{\int g(x)dx \cdot g(y)}{\int g(x)dx \int g(y)dy} = \frac{g(y)}{\int g(y)dy}$$



Introduction to slicing

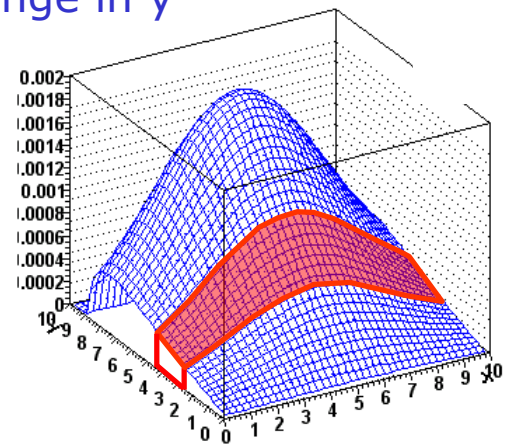
- With multidimensional p.d.f.s it is also often useful to be able to plot a slice of a p.d.f
- In RooFit
  - A *slice* is thin
  - A *range* is thick
- Slices mostly useful in discrete observables
  - A slice in a continuous observable has no width and usually no data with the corresponding cut (e.g. "x=5.234")
- Ranges work for both continuous and discrete observables
  - Range of discrete observable can be list of >=1 state

Slice in x



x = x.getVal()

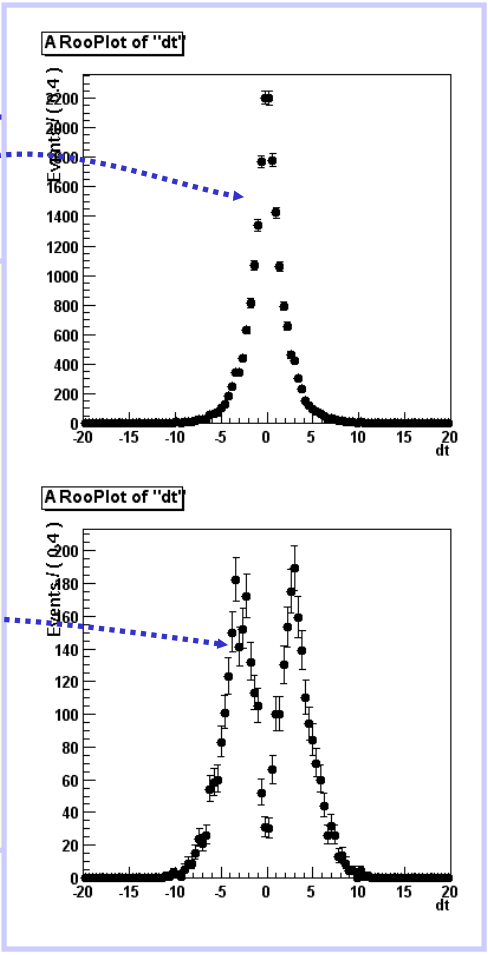
Range in y



Plotting a *slice* of a dataset

- Use the optional cut string expression

```
// Mixing dataset defines dt,mixState
RooDataSet* data ;

// Plot the entire dataset
RooPlot* frame = dt.frame() ;
data->plotOn(frame) ;

// Plot the mixed part of the data
RooPlot* frame_mix = dt.frame() ;
data->plotOn(frame,
             Cut("mixState==mixState::mixed"))
```



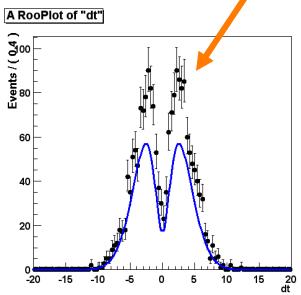
  - Works the *same* for *binned data* sets

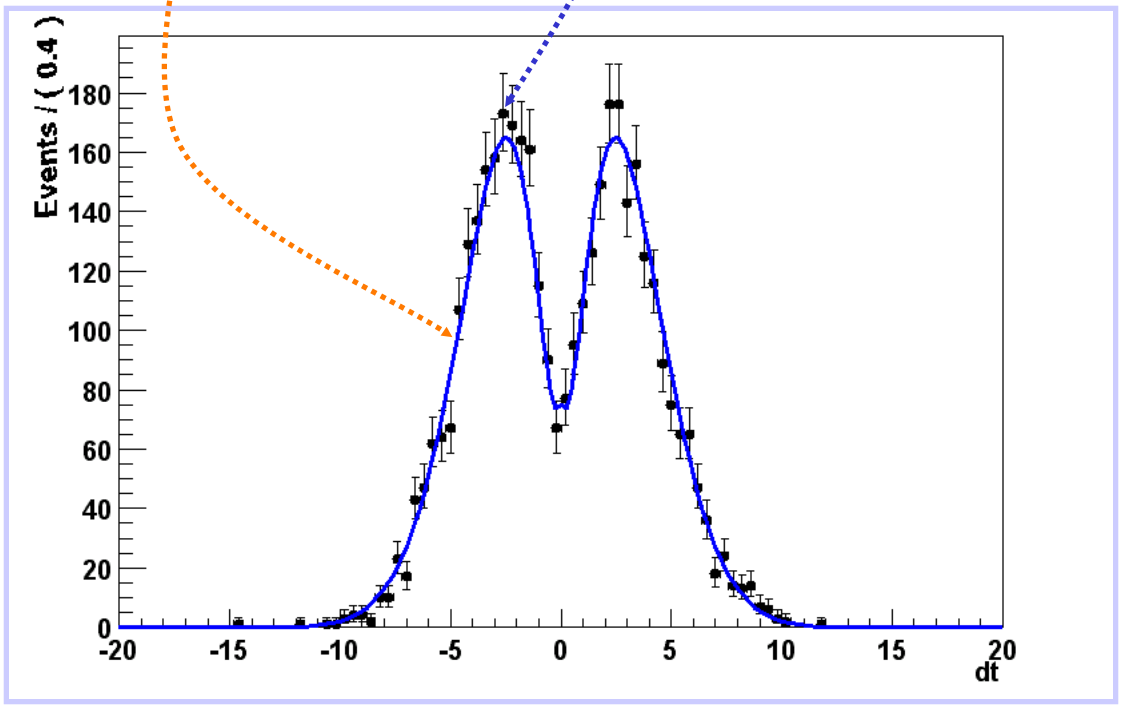
Plotting a *slice* of a p.d.f

```
RooPlot* dtframe = dt.frame() ;
data->plotOn(dtframe,Cut("mixState==mixState::mixed")) ;

mixState = "mixed" ;
bmix.plotOn(dtframe,Slice(mixState)) ;
dtframe->Draw() ;
```

Slice is positioned at 'current' value of sliced observable

For slices both data and p.d.f normalize with respect to full dataset. If fraction 'mixed' in above example disagrees between data and p.d.f prediction, this discrepancy will show in plot



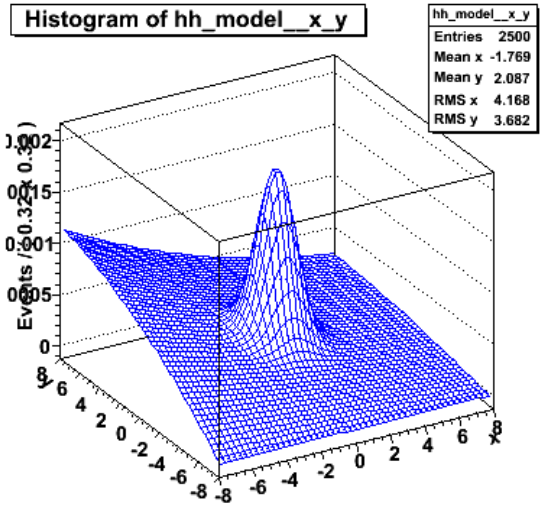


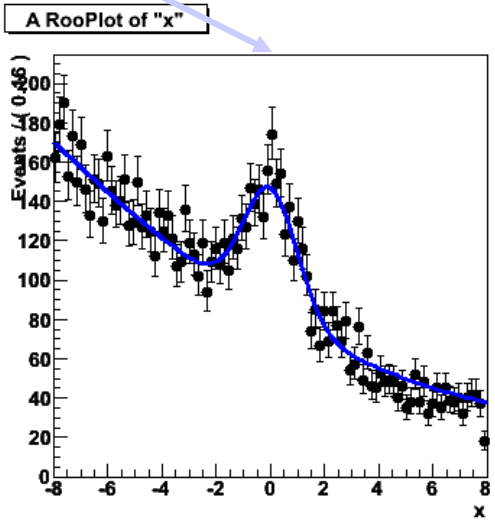
Plotting a *range* of a p.d.f and a dataset

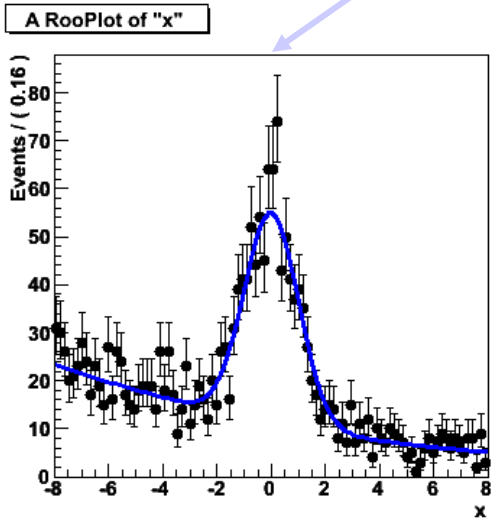
model(x,y) = gauss(x)*gauss(y) + poly(x)*poly(y)

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

```
y.setRange("sig",-1,1) ;
RooPlot* xframe2 = x.frame() ;
data->plotOn(xframe2,CutRange("sig")) ;
model.plotOn(xframe2,ProjectionRange("sig")) ;
```







→ *Works also with >2D projections (just specify projection range on all projected observables)*

→ *Works also with multidimensional p.d.fs that have correlations*

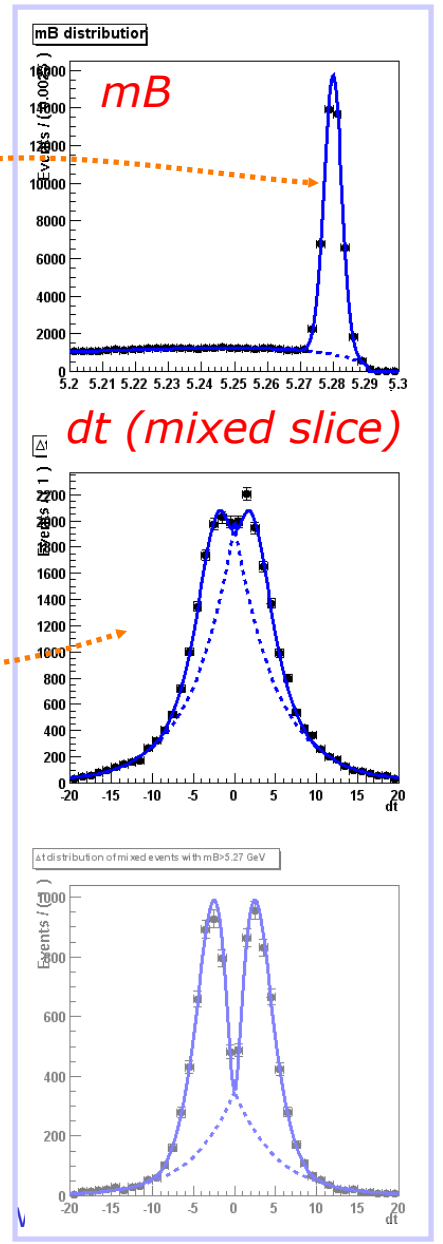
Physics example of combined range and slice plotting

Example setup:

Argus(mB)*Decay(dt) + (background)

Gauss(mB)*BMixDecay(dt) (signal)

```
// Plot projection on mB
RooPlot* mbframe = mb.frame(40) ;
data->plotOn(mbframe) ;
model.plotOn(mbframe) ;

// Plot mixed slice projection on deltat
RooPlot* dtframe = dt.frame(40) ;
data>plotOn(dtframe,
            Cut("mixState==mixState::mixed")) ;
mixState="mixed" ;
model.plotOn(dtframe,Slice(mixState)) ;
```



参考 tutorials/roofit/rf310_sliceplot.C

Plotting slices with finite width - Example

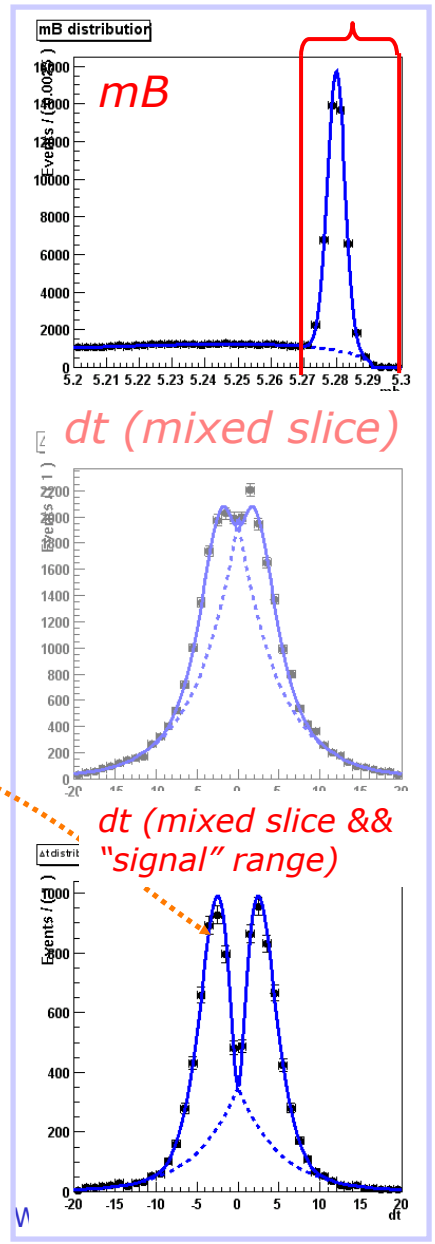
"signal"

Example setup:

Argus(mB)*Decay(dt) + (background)

Gauss(mB)*BMixDecay(dt) (signal)

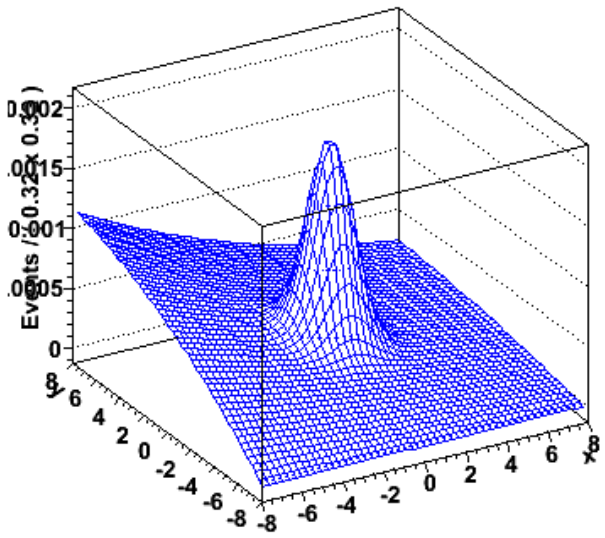
```
mb.setRange("signal",5.27,5.30) ;

mbSliceData->plotOn(dtframe2,
   Cut("mixState==mixState::mixed"),
   CutRange("signal"))

model.plotOn(dtframe2,Slice(mixState),
             ProjectionRange("signal"))
```

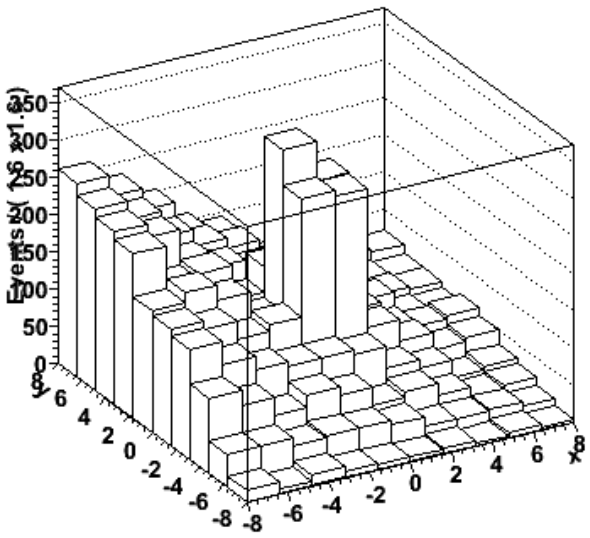


Plotting in more than 2,3 dimensions

- No equivalent of RooPlot for >1 dimensions
  - Usually >1D plots are not overlaid anyway
- Easy to use **`createHistogram()`** methods provided in both **`RooAbsData`** and **`RooAbsPdf`** to fill ROOT 2D,3D histograms

```
TH2D* ph2 = pdf.createHistogram("ph2",x,YVar(y)) ;

TH2* dh2 = data.createHistogram("dg2",x,Binning(10),
                                YVar(y,Binning(10)));
ph2->Draw("SURF") ;
dh2->Draw("LEGO") ;
```





Building models – Introducing correlations

- Easiest way to do this is
  - start with 1-dim p.d.f. and change on of its parameters into a function that depends on another observable

$$f(x;p) \Rightarrow f(x,p(y,q)) = f(x,y;q)$$

  - Natural way to think about it
- Example problem
  - Observable is reconstructed mass M of some object.
  - Fitting Gaussian g(M,mean,sigma) some background to dataset D(M)
  - But reconstructed mass has *bias* depending on some other observable X
  - Rewrite fit functions as g(M,meanCorr(mtrue,X,alpha),sigma) where meanCorr is an (emperical) function that corrects for the bias depending on X

Coding the example problem

How do you code the preceding example problem

$PDF(x,y) = gauss(x,m(y),s)$

$m(y) = m_0 + m_1 \cdot sqrt(y)$

How do you do that? Just like that:

```
RooRealVar x("x","x",-10,10) ;
RooRealVar y("y","y",0,3) ;

// Build a parameterized mean variable for gauss
RooRealVar mean0("mean0","mean offset",0.5) ;
RooRealVar mean1("mean1","mean slope",3.0) ;
RooFormulaVar mean("mean","mean0+mean1*y",
                   RooArgList(mean0,mean1,y)) ;

RooRealVar sigma("sigma","width of gaussian",3) ;
RooGaussian gauss("gauss","gaussian",x,mean,sigma);
```

Build a function object $m(y)=m_0+m_1*sqrt(y)$

Simply plug in function mean(y) where mean value is expected!

PDF expects a real-valued object as input, not necessarily a variable

Plug-and-play parameters!

Generic real-valued functions

- **RooFormulaVar** makes use of the ROOT **TFormula** technology to build interpreted functions
  - Understands generic C++ expressions, operators etc
  - Two ways to reference RooFit objects
    By name:

```
RooFormulaVar f("f","exp(foo)*sqrt(bar)", RooArgList(foo,bar)) ;
```

By position:

```
RooFormulaVar f("f","exp(@0)*sqrt(@1)",RooArgList(foo,bar)) ;
```

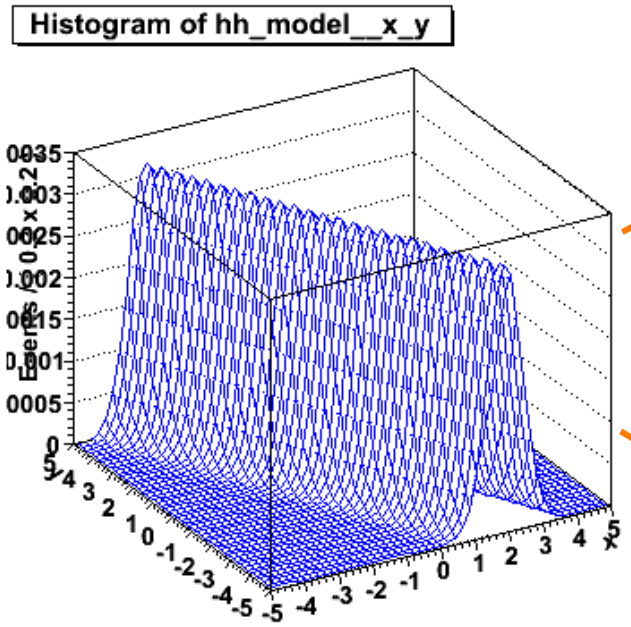
  - You can use **RooFormulaVar** where ever a 'real' variable is requested

- **RooPolyVar** is a compiled polynomial function

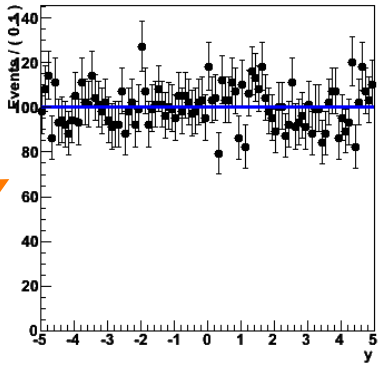
```
RooRealVar x("x","x",0.,1.) ;
RooRealVar p0("p0","p0",5.0) ;
RooRealVar p1("p1","p1",-2.0) ;
RooRealVar p2("p2","p2",3.0) ;
RooFormulaVar f("f","polynomial",x,RooArgList(p0,p1,p2)) ;
```

What does the example p.d.f look like?

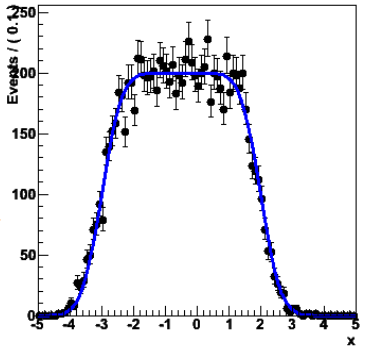
- Make 2D plot of p.d.f in (x,y)



Projection on Y



Projection on X



- Is the correct p.d.f for this problem?
  - Constructed a p.d.f with correct shape in x, given a value of y → OK
  - But p.d.f predicts flat distribution in y → Probably not OK
  - What we want is a pdf for X *given* Y, but without prediction on Y → Definition of a *conditional* p.d.f F(x|y)

参考 tutorials/roofit/rf301_composition.C

Conditional p.d.f.s – Formulation and construction

- Mathematical formulation of a conditional p.d.f
  - A conditional p.d.f is not normalized w.r.t its conditional observables

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

  - Note that denominator in above expression *depends on y* and is thus in general different for each event

- Constructing a conditional p.d.f in RooFit
  - Any RooFit p.d.f can be used as a conditional p.d.f as objects have no internal notion of distinction between parameters, observables and conditional observables
  - Observables that should be used as conditional observables have to be specified in use context (generation, plotting, fitting etc…)

Using a conditional p.d.f – fitting and plotting

- For fitting, indicate in fitTo() call what the conditional observables are

```
pdf.fitTo(data,ConditionalObservables(y))
```

$$F(x \mid y) = \frac{f(x,y)}{\int f(x,y)d\vec{x}}$$

  - You may notice a performance penalty if the normalization integral of the p.d.f needs to be calculated numerically. For a conditional p.d.f it must evaluated again for each event

- Plotting: You cannot project a conditional $F(x|y)$ on x without external information on the distribution of y
  - Substitute integration with averaging over y values in data

Integrate over y

$$P_p(x) = \frac{\int p(x,y)dy}{\int p(x,y)dxdy}$$

Sum over all y_i in dataset D

$$P_p(x) = \frac{1}{N}\sum_{D}^{i=1,N} \frac{p(x,y_i)}{\int p(x,y_i)dx}$$

Physics example with conditional p.d.f.s

- Want to fit decay time distribution of B0 mesons (exponential) convoluted with Gaussian resolution

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

- However, resolution on decay time varies from event by event (e.g. more or less tracks available).
  - We have in the data an error estimate dt for each measurement from the decay vertex fitter ("*per-event error*")
  - Incorporate this information into this physics model

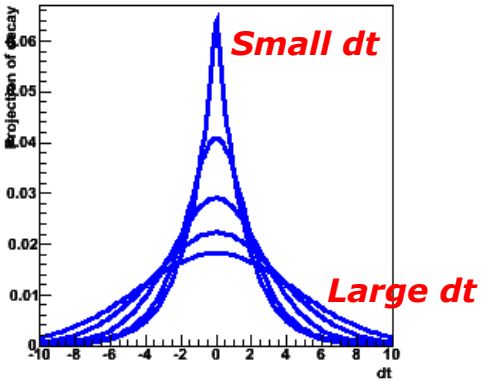
$$F(t \mid \delta t) = D(t;\tau) \otimes R(t,m,\sigma \cdot \delta t)$$

  - Resolution in physics model is adjusted for each event to expected error.
  - Overall scale factor σ can account for incorrect vertex error estimates (i.e. if fitted σ>1 then dt was underestimate of true error)
  - Physics p.d.f must used conditional conditional p.d.f because it give no sensible prediction on the distribution of the per-event errors

Physics example with conditional p.d.f.s

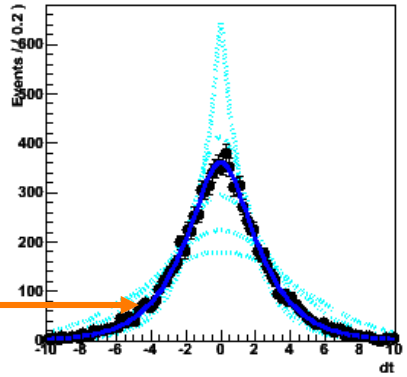
- Some illustrations of decay model with per-event errors
  - Shape of $F(t|\delta t)$ for several values of δt

$$F(t \mid \delta t) = D(t;\tau) \otimes R(t, m, \sigma \cdot \delta t)$$



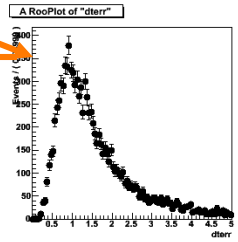
- Plot of *D(t)* and *F(t|dt)* projected over *dt*

```
// Plotting of decay(t|dterr)
 RooPlot* frame = dt.frame() ;
 data->plotOn(frame2) ;
 decay_gm1.plotOn(frame2,ProjWData(*data)) ;
```



Note that projecting over large datasets can be slow. You can speed this up by projecting with a binned copy of the projection data

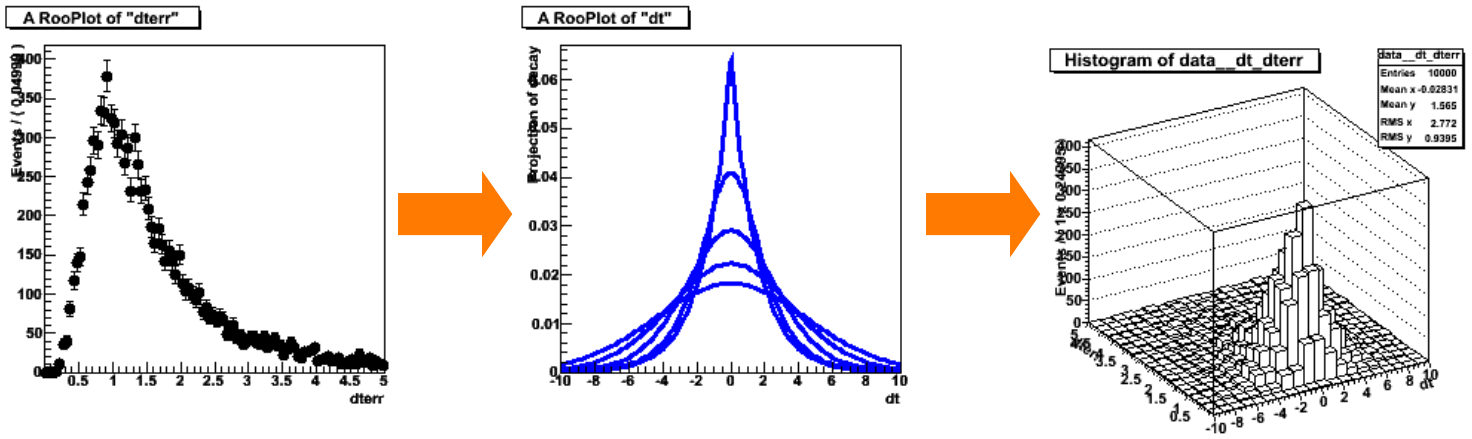
$$P_p(x) = \frac{1}{N} \sum_{D}^{i=1,N} \frac{p(x, y_i)}{\int p(x, y_i) dx}$$



参考 tutorials/roofit/rf303_conditional.C

How it works – event generation with conditional p.d.f.s

- Just like plotting, event generation of conditional p.d.f.s requires external input on the conditional observables
  - Given an external input dataset $\boldsymbol{P}(dt)$
  - For each event in $\boldsymbol{P}$,
    set the value of dt in $F(d|dt)$ to dt_i
    generate one event for observable t from $F(t|dt_i)$
  - Store both t_i and dt_i in the output dataset

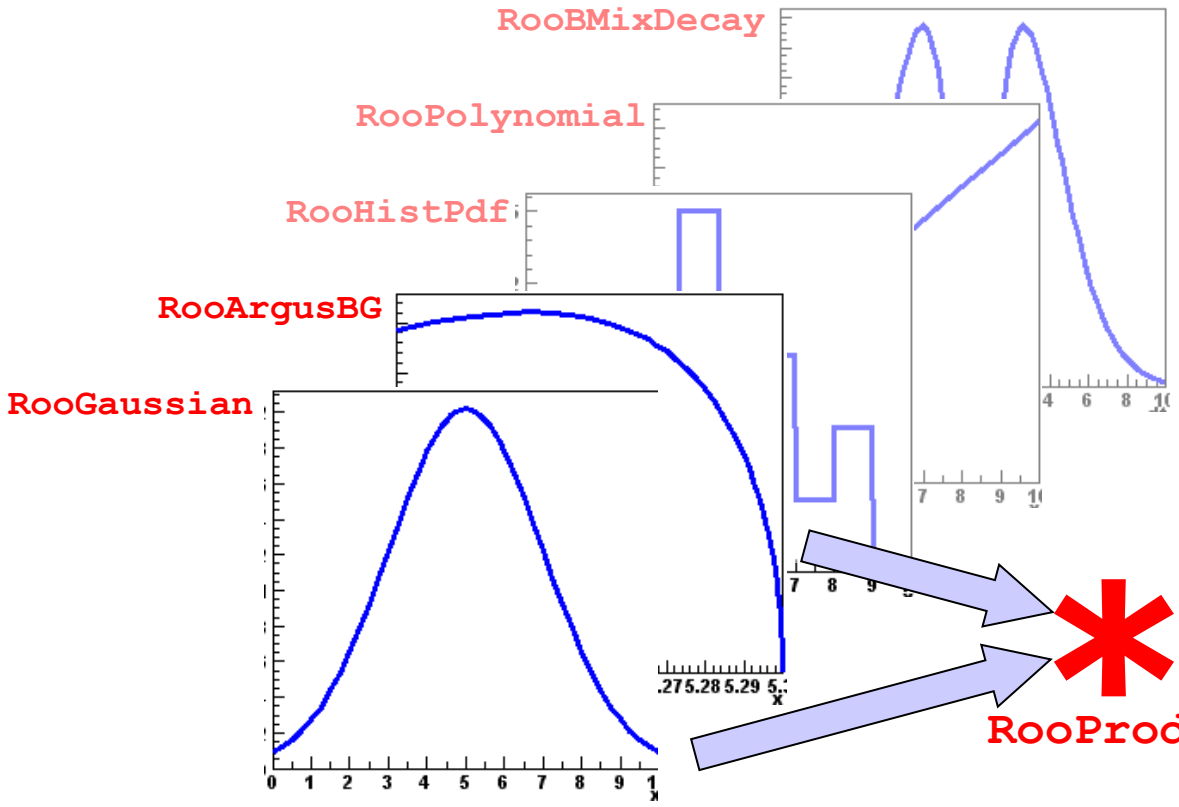


Complete example of decay with per-event errors

```
RooRealVar dt("dt","dt",-10,10) ;
RooRealVar dterr("dterr","dterr",0.001,5) ;
RooRealVar tau("tau","tau",1.548) ;

// Build Gauss(dt,0,sigma*dterr)
RooRealVar sigma("sigma","sigma1",1) ;
RooGaussModel gm1("gm1","gauss model 1",dt,RooConst(0),sigma,dterr) ;

// Construct decay(t,tau) (x) gauss1(t,0,sigma*dterr)
RooDecay decay_gm1("decay_gm1","decay",dt,tau,gm1,RooDecay::DoubleSided) ;

// Toy MC generation of decay(t|dterr)
RooDataSet* toydata = decay_gm1.generate(dt,ProtoData(dterrData)) ;

// Fitting of decay(t|dterr)
decay_gm1.fitTo(*data,ConditionalObservables(dterr))

// Plotting of decay(t|dterr)
RooPlot* frame = dt.frame() ;
data->plotOn(frame2) ;
decay_gm1.plotOn(frame2,ProjWData(*data)) ;
```

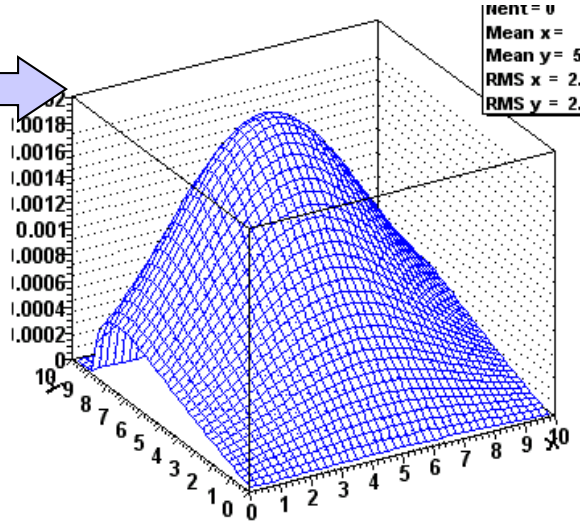
参考 tutorials/roofit/rf306_condpereventerrors.C

Model building – Products with conditional p.d.f.s




```
RooProdPdf k("k","k",g,
             Conditional(f,x))
```

$$K(x,y) = F(x \mid y) \cdot G(y)$$



Products with conditional p.d.f.s – Mathematical form

- Use of conditional p.d.f.s has some drawbacks
  - Practical: Somewhat unwieldy in use because external input needed e.g. in plotting and event generation steps
  - Fundamental: In composite conditional p.d.f.s

$$F(x \mid y) = f \cdot S(x \mid y) + (1 - f) \cdot B(x \mid y)$$

signal and background by construction always using the same distributions for conditional observables. This assumption may not be valid leading, to possible fit biases (Punzi physics/0401045)

- Can mitigate both problems by multiplying conditional p.d.f.s with a p.d.f. for the conditional observables so that product is not conditional
  - Can multiply with different p.d.f for signal and background

$$K(x,y) = F(x \mid y) \cdot G(y) = \frac{f(x,y)}{\int f(x,y)dx} \frac{g(y)}{\int g(y)dy}$$

Normalization and event generation in conditional products

- Products of conditional and plain pdf's are self normalized
  - Proof is trivial

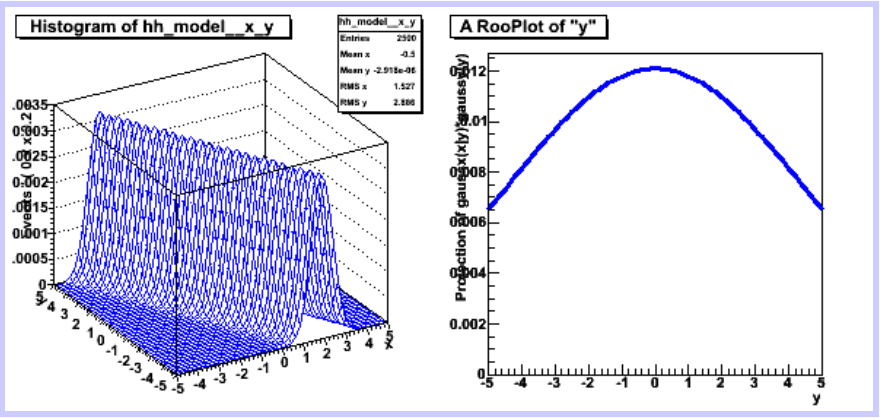
$$\iint K(x,y) = \iint \frac{f(x,y)}{\int f(x,y)dx}\frac{g(y)}{\int g(y)dy}dxdy = \left(\int \frac{f(x,y)}{\int f(x,y)dx}dx\right)\left(\int \frac{g(y)}{\int g(y)dy}dy\right) = 1\cdot 1$$

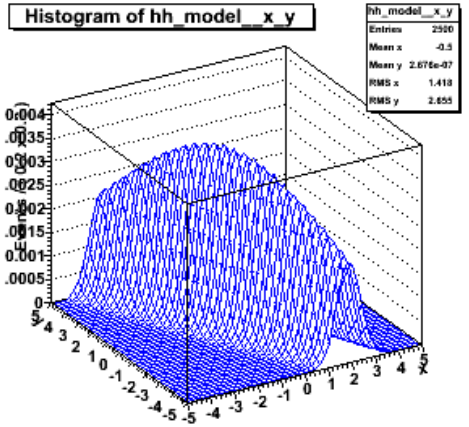
- Generation of events from products of conditional and plain p.d.fs can be handling by handling generation of observables in order

$F(x \mid y)\cdot G(y)$ *First generate y, then x*

$F(x \mid y)\cdot G(y \mid z)\cdot H(z)$ *First generate z, then y, then x*

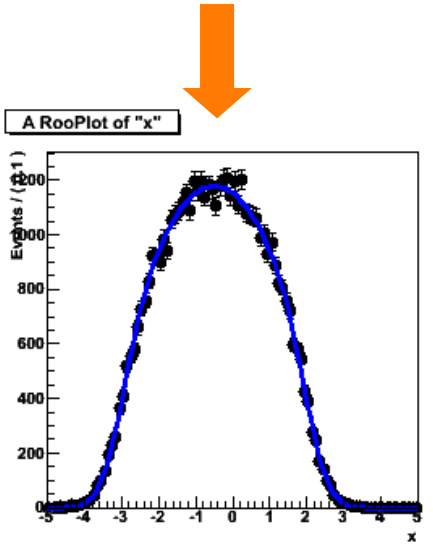
Example with product of conditional and plain p.d.f.





gx(x|y) * gy(y) = model(x,y)

```
// Create function f(y) = a0 + a1*y
RooPolyVar fy("fy","fy",y,RooArgSet(a0,a1)) ;

// Create gaussx(x,f(y),0.5)
RooGaussian gaussx("gaussx","gaussx",x,fy,sx) ;

// Create gaussy(y,0,3)
RooGaussian gaussy("gaussy","Gaussian in y",y,my,sy) ;

// Create gaussx(x,sx|y) * gaussy(y)
RooProdPdf model("model","gaussx(x|y)*gaussy(y)",
                 gaussy,Conditional(gaussx,x)) ;
```



$$\int gx(x \mid y)g(y)dy$$

5

Managing data, discrete variables simultaneous fits

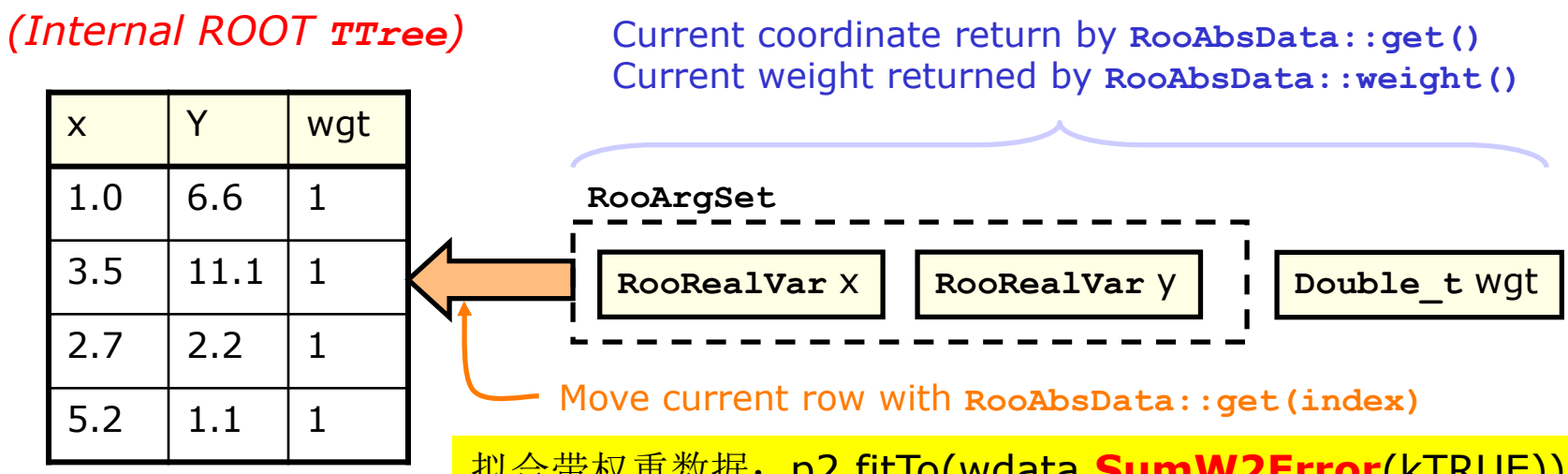
- *Binned, unbinned datasets*
- *Importing data*
- *Using discrete variable to classify data*
- *Simultaneous fits on multiple datasets*

Wouter Verkerke, NIKHEF

A bit more detail on RooFit datasets

- A dataset is a N-dimensional collection of points
  - With optional weights
  - No limit on number of dimensions
  - Observables continuous (`RooRealVar`) or discrete (`RooCategory`)

- Interface of each dataset is 'current' row
  - Set of RooFit value objects that represent coordinate of current event



拟合带权重数据：p2.fitTo(wdata,**SumW2Error**(kTRUE)) ;

参考 tutorials/roofit/rf403_weightedevts.C

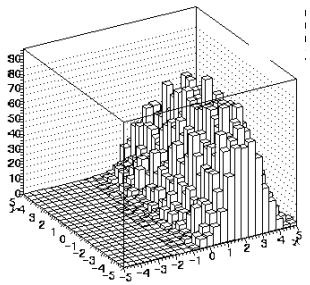
Binned data, or unbinned data (with optional weights)

- Binned or unbinned ML fit?
  - In most RooFit applications it doesn't matter

Unbinned

x	y	z
1	3	5
2	4	6
1	3	5
2	4	6

Binned



Internally binned data is represented the same way as unbinned data,

A ROOT TTree with the bin coordinates

RooDataSet **RooDataHist**

RooAbsData

- For example ML fitting interface takes abstract `RooAbsData` object
  - Binned data → Binned likelihood
  - Unbinned data → Unbinned likelihood
- Weights are supported in unbinned datasets
  - But use with care. Error analysis in ML fits to weighted unbinned data can be complicated!

Importing unbinned data

- From ROOT trees
  - `RooRealVar` variables are imported from /D /F /I tree branches
  - `RooCategory` variables are imported from /I /b tree branches
  - Mapping between `TTree` branches and dataset variables by name: e.g. `RooRealVar x("x","x",-10,10)` imports `TTree` branch "x"

```
RooRealVar x("x","x",-10,10) ;
RooRealVar c("c","c",0,30) ;
RooDataSet data("data","data",inputTree,RooArgSet(x,c));
```

  - Only events with 'valid' entries are imported. In above example any events with |x|>10 or c<0 or c>30 are *not* imported

- From ASCII files
  - One line per event, order of variables as given in RooArgList

```
RooDataSet* data =
   RooDataSet::read("ascii.file",RooArgList(x,c)) ;
```

Importing binned data

- From ROOT **`THx`** histogram objects

```
RooDataHist bdata1("bdata","bdata",RooArgList(x),histo1d);
RooDataHist bdata2("bdata","bdata",RooArgList(x,y),histo2d);
RooDataHist bdata3("bdata","bdata",RooArgList(x,y,z),histo3d);
```

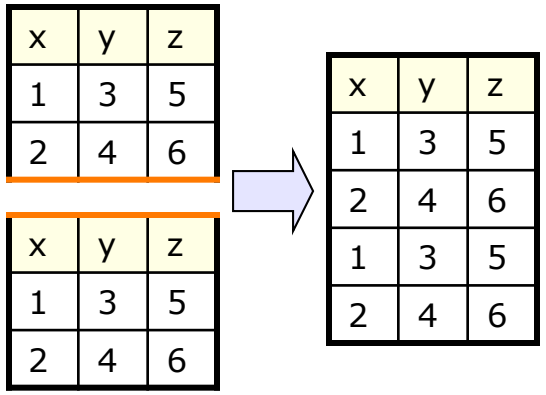
- From a **`RooDataSet`**

```
RooDataHist* binnedData = data->binnedClone() ;
```

Extending and reducing **unbinned** datasets

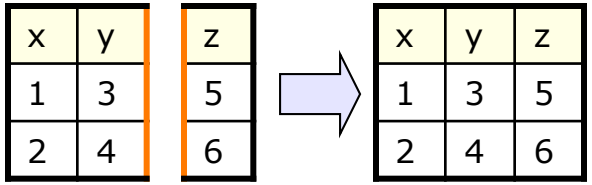
- Appending

```
RooDataSet d1("d1","d1",RooArgSet(x,y,z));
RooDataSet d2("d2","d2",RooArgSet(x,y,z));

d1.append(d2) ;
```



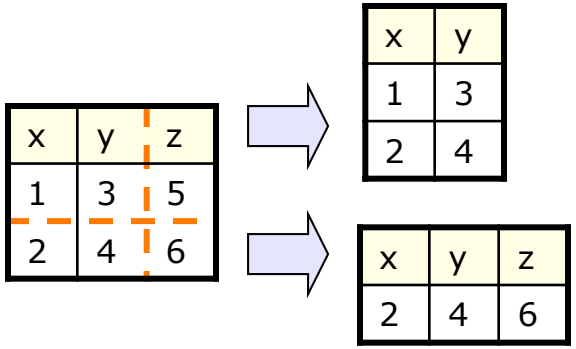
- Merging

```
RooDataSet d1("d1","d1",RooArgSet(x,y) ;
RooDataSet d2("d2","d2",RooArgSet(z)) ;

d1.merge(d2) ;
```



- Reducing

```
RooDataSet d1("d1","d1",RooArgSet(x,y,z) ;

RooDataSet* d2 = d1.reduce(RooArgSet(x,y));

RooDataSet* d3 = d1.reduce("x>1");
```



Adding and reducing **binned** datasets

- Adding

```
RooDataHist d1("d1","d1",
               RooArgSet(x,y));
RooDataHist d2("d2","d2",
               RooArgSet(x,y));

d1.add(d2) ;
```

w	y1	y2
x1	0	**1**
x2	**1**	0

w	y1	y2
x1	**1**	0
x2	0	**1**

w	y1	y2
x1	**1**	**1**
x2	**1**	**1**

- Reducing

```
RooDataHist d1("d1","d1",
               RooArgSet(x,y) ;

RooDataHist* d2 =
     d1.reduce(x);

RooDataHist* d3 =
     d1.reduce(y,"x>0");
//Creating 1-dimensional projection on y of d1 for bins with x>0
```

w	y1	y2
x1	0	1
x2	1	0

-	w
x1	**1**
x2	**1**

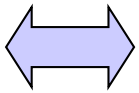
参考 tutorials/roofit/rf402_datahandling.C

Datasets and discrete observables

- Discrete observables play an important role in management of datasets
  - Useful to classify 'sub datasets' inside datasets
  - Can collapse multiple, logically separate datasets into a single dataset by adding them and labeling the source with a discrete observable
  - Allows to express operations such a simultaneous fits as operation on a single dataset

Dataset A
x
5.0
3.7
1.2
4.3

Dataset B
x
5.0
3.7
1.2



Dataset A+B	
x	**source**
5.0	A
3.7	A
1.2	A
4.3	A
5.0	B
3.7	B
1.2	B

Discrete variables in RooFit – RooCategory

- Properties of RooCategory variables
  - Finite set of named states → self documenting
  - Optional integer code associated with each state

At creation, a category has no states

Add states with a label *and index*

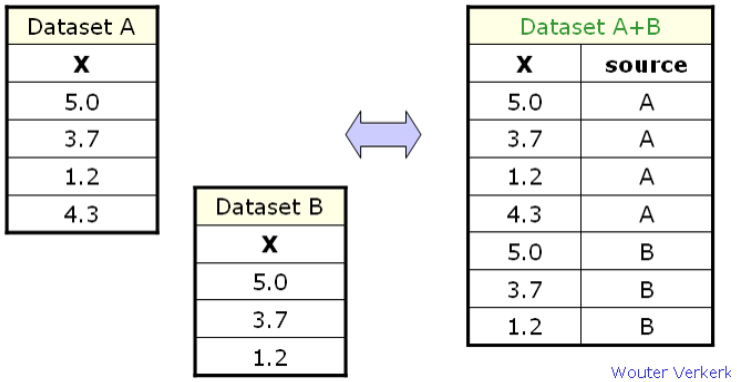
Add states with a label only. *Indices will be automatically assigned*

```
// Define a cat. with explicitly numbered states
RooCategory b0flav("b0flav","B0 flavour") ;
b0flav.defineType("B0",-1) ;
b0flav.defineType("B0bar",1) ;

// Define a category with labels only
RooCategory tagCat("tagCat","Tagging technique") ;
tagCat.defineType("Lepton") ;
tagCat.defineType("Kaon") ;
tagCat.defineType("NetTagger-1") ;
tagCat.defineType("NetTagger-2") ;
```

- Used for classification of data, or to describe occasional discrete fundamental observable (e.g. B^0 flavor)

Datasets and discrete observables – part 2

- Example of appending datasets with label attachment



Dataset A
x
5.0
3.7
1.2
4.3

Dataset B
x
5.0
3.7
1.2

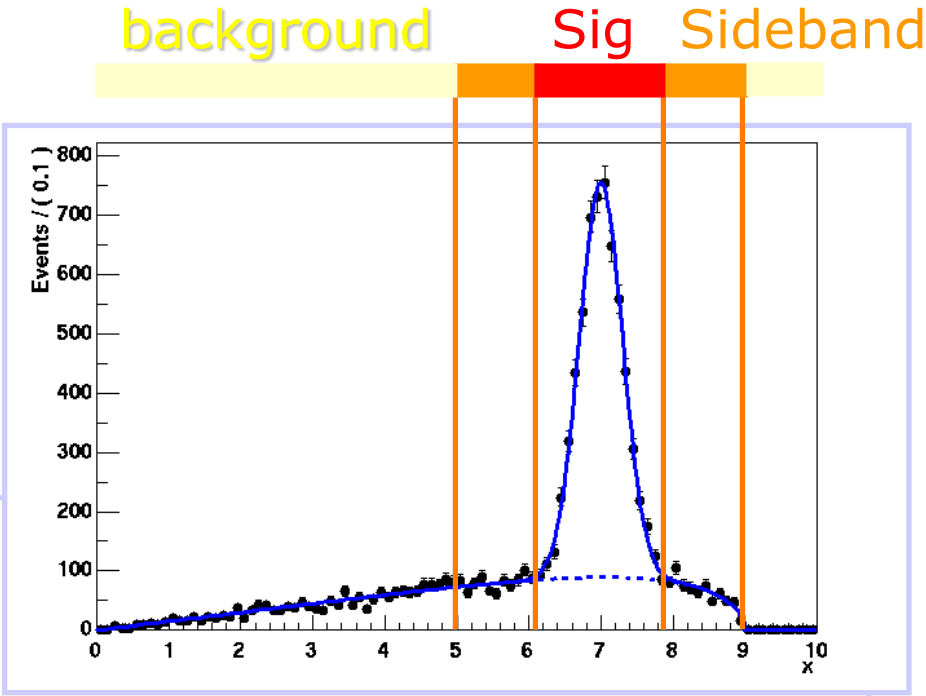
Dataset A+B	
x	**source**
5.0	A
3.7	A
1.2	A
4.3	A
5.0	B
3.7	B
1.2	B


```
RooCategory c("c","source")
c.defineType("A") ;
c.defineType("B") ;

// Add column with source label
c.setLabel("A") ; dA->addColumn(c) ;
c.setLabel("B") ; dB->addColumn(c) ;

// Make combined dataset
RooDataSet* dAB = dA->Clone("dAB") ;
dAB->append(*dB) ;
```

- But can also derive classification from info within dataset
  - E.g. (10<x<20 = "signal", 0<x<10 | 20<x<30 = "sideband")
  - Encode classification using real→discrete mapping functions

A universal real→discrete mapping function

- Class **RooThresholdCategory** maps ranges of input **RooRealVar** to states of a **RooCategory**




```
// Mass variable
RooRealVar m("m","mass,0,10.);

// Define threshold category
RooThresholdCategory region("region","Region of M",m,"Background");
region.addThreshold(9.0, "SideBand") ;
region.addThreshold(7.9, "Signal") ;
region.addThreshold(6.1,"SideBand") ;
region.addThreshold(5.0,"Background") ;
```

Default state

Define region boundaries

参考 tutorials/roofit/rf405_realtocatfuncs.C

Discrete→Discrete mapping function

- **`RooMappedCategory`** provides cat → cat mapping

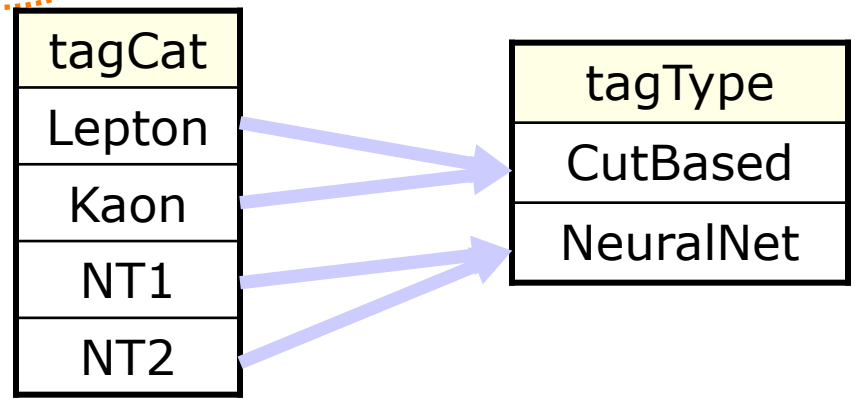
Define input category

Create mapped category

Add mapping rules

```
RooCategory tagCat("tagCat","Tagging category") ;
tagCat.defineType("Lepton") ;
tagCat.defineType("Kaon") ;
tagCat.defineType("NetTagger-1") ;
tagCat.defineType("NetTagger-2") ;

RooMappedCategory tagType("tagType","type",tagCat) ;

tagType.map("Lepton","CutBased") ;
tagType.map("Kaon","CutBased") ;
tagType.map("NT*","NeuralNet") ;
```

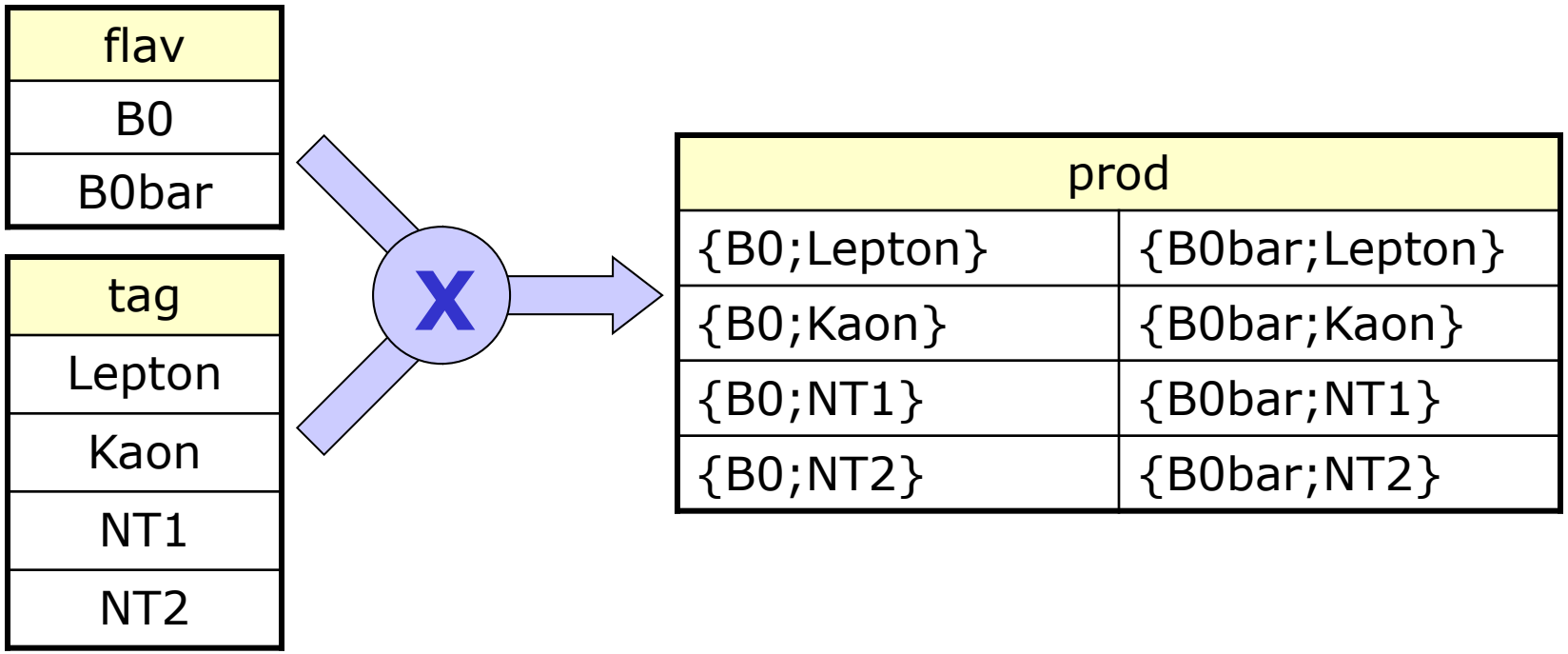
Wildcard expressions allowed



Discrete multiplication function

- **`RooSuperCategory/RooMultiCategory`** provides category multiplication

```
// Define 'product' of tagCat and runBlock
RooSuperCategory prod("prod","prod",RooArgSet(tag,flav))
```



Exploring discrete data

- Like real variables of a dataset can be plotted, discrete variables can be tabulated

Tabulate contents of dataset by category state

```
RooTable* table=data->table(b0flav) ;
table->Print() ;

Table b0flav : aData
  +-------+------+
  |    B0 | 4949 |
  | B0bar | 5051 |
  +-------+------+
```

Extract contents by label

```
Double_t nB0 = table->get("B0") ;
```

Extract contents fraction by label

```
Double_t b0Frac = table->getFrac("B0");
```

Tabulate contents of selected part of dataset

```
data->table(tagCat,"x>8.23")->Print() ;

  Table tagCat : aData(x>8.23)
  +-------------+-----+
  |      Lepton | 668 |
  |        Kaon | 717 |
  | NetTagger-1 | 632 |
  | NetTagger-2 | 616 |
  +-------------+-----+
```

Exploring discrete data

- *Discrete functions*, built from categories in a dataset can be tabulated likewise

Tabulate **RooSuperCategory** states

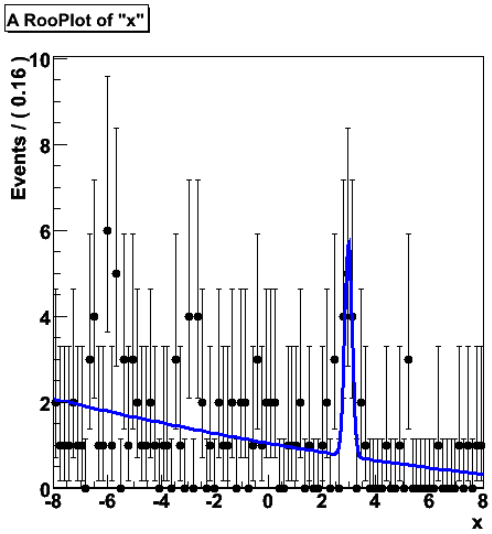
```
data->table(b0Xtcat)->Print() ;

  Table b0Xtcat : aData
  +----------------------+------+
  |          {B0;Lepton} | 1226 |
  |      {B0bar;Lepton}  | 1306 |
  |           {B0;Kaon}  | 1287 |
  |        {B0bar;Kaon}  | 1270 |
  |    {B0;NetTagger-1}  | 1213 |
  | {B0bar;NetTagger-1}  | 1261 |
  |    {B0;NetTagger-2}  | 1223 |
  | {B0bar;NetTagger-2}  | 1214 |
  +----------------------+------+
```

Tabulate **RooMappedCategory** states

```
data->table(tcatType)->Print() ;

  Table tcatType : aData
  +----------------+------+
  |        Unknown |    0 |
  |      Cut based | 5089 |
  | Neural Network | 4911 |
  +----------------+------+
```

Fitting multiple datasets simultaneously

- Simultaneous fitting efficient solution to incorporate information from control sample into signal sample
- Example problem: search rare decay
  - Signal dataset has small number entries.

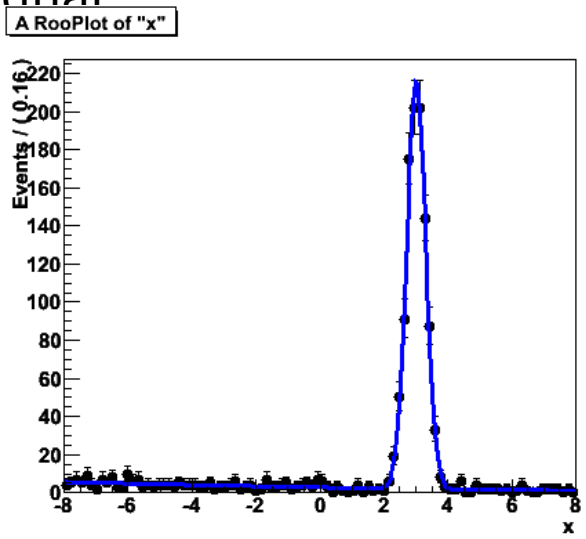


Par	FinalValue +/-	Error
a0	-1.0544e-01 +/-	2.88e-02
a1	2.2698e-03 +/-	4.92e-03
nbkg	1.0933e+02 +/-	1.07e+01
nsig	1.0680e+01 +/-	3.92e+00
mean	2.9787e+00 +/-	6.25e-02
width	1.3764e-01 +/-	6.29e-02

  - Statistical uncertainty on shape in fit contributes significantly to uncertainty on fitted number of signal events
  - However can constrain shape of signal from control sample (e.g. another decay with similar properties that is not rare), so no need to relay on simulations

Fitting multiple datasets simultaneously

- Fit to control sample yields accurate information on shape of signal



Par	FinalValue +/-	Error
a0	-9.9212e-02 +/-	1.75e-02
a1	3.3116e-03 +/-	3.57e-03
nbkg	3.0406e+02 +/-	1.83e+01
nsig	9.9594e+02 +/-	3.21e+01
m	3.0098e+00 +/-	9.83e-03
s	2.9891e-01 +/-	7.39e-03

- Q: What is the most practical way to combine shape measurement on control sample to measurement of signal on physics sample of interest

- A: Perform a simultaneous fit
  - Automatic propagation of errors & correlations
  - Combined measurement
    (i.e. error will reflect contributions from both physics sample and control sample

Discrete observable as data subset classifier

- Likelihood level definition of a simultaneous fit

$$-\log(L) = \sum_{i=1,n} -\log(PDF_A(D_A^i)) + \sum_{i=1,m} -\log(PDF_B(D_B^i))$$

- PDF level definition of a simultaneous fit

$$-\log(L) = \sum_{i=1,n} -\log(simPDF(D_{A+B}^i))$$

RooSimultaneous implements 'switch' PDF:

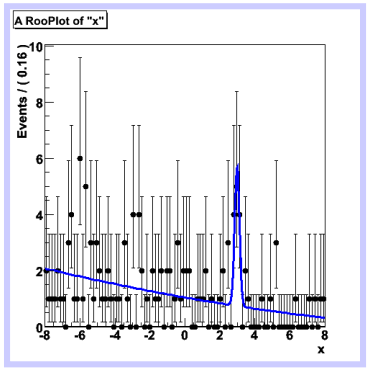
```
case (indexCat) {
  A: return pdfA ;
  B: return pdfB ;
}
```

Likelihood of switchPdf with composite dataset *automatically* constructs sum of likelihoods above

Dataset A+B	
X	**source**
5.0	A
3.7	A
1.2	A
4.3	A
5.0	B
3.7	B
1.2	B

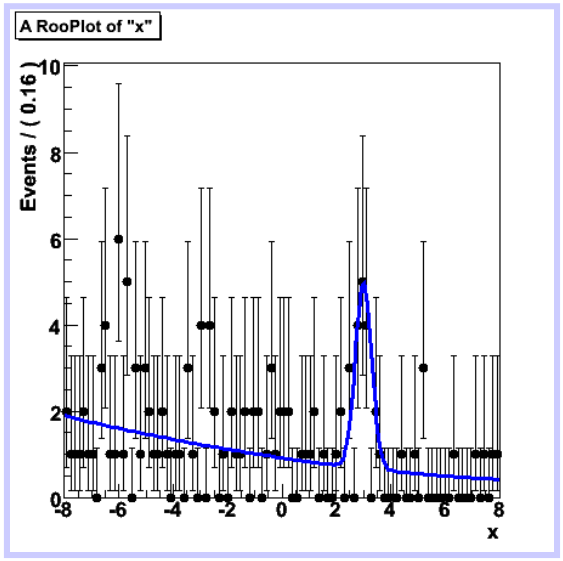
Using RooSimultaneous to implement preceding example

Fit to signal data




```
RooCategory c("c","c") ;
c.defineType("control") ;
c.defineType("physics") ;

RooSimultaneous sim_model("sim_model","",c) ;
sim_model.addPdf(model_phys,"physics") ;
sim_model.addPdf(model_ctrl,"control") ;

sim_model.fitTo(*d,Extended()) ;
```

Combined fit



Signal shape constrained from control sample

Parameter	FinalValue +/-	Error
a0_ctrl	-8.0947e-02 +/-	1.47e-02
a0_phys	-1.1825e-01 +/-	3.26e-02
a1_ctrl	2.1004e-04 +/-	3.12e-03
a1_phys	4.2259e-03 +/-	5.55e-03
nbkg_ctrl	3.1054e+02 +/-	1.86e+01
nbkg_phys	1.0633e+02 +/-	1.06e+01
nsig_ctrl	9.8946e+02 +/-	3.20e+01
nsig_phys	1.3647e+01 +/-	4.44e+00
m	2.9983e+00 +/-	9.69e-03
s	2.9255e-01 +/-	7.53e-03

Relative error on Nsig improved from 37% to 32%

Other scenarios in which simultaneous fits are useful

- Preceding example was 'asymmetric'
  - Very large control sample, small signal sample
  - Physics in each channel possibly different (but with some similar properties

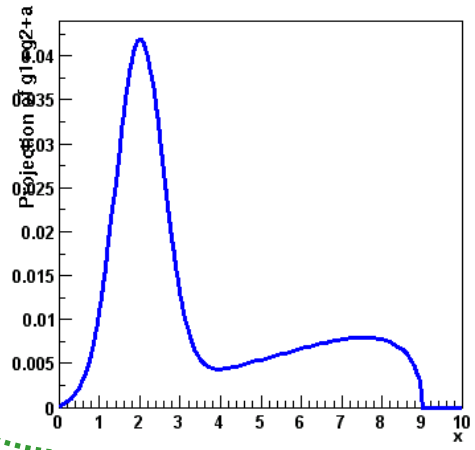
- There are also 'symmetric' use cases
  - Fit multiple data sets that are functionally equivalent, but have slightly different properties (e.g. purity)
  - Example: Split B physics data in block separated by flavor tagging technique (each technique results in a different sensitivity to CP physics parameters of interest).
  - Split data in block by data taking run, mass resolutions in each run may be slightly different
  - For symmetric use cases pdf-level definition of simultaneous fit very convenient as you usually start with a single dataset with subclassing formation derived from its observables

- By splitting data into subsamples with p.d.f.s that can be tuned to describe the (slightly) varying properties you can increase the statistical sensitivity of your measurement

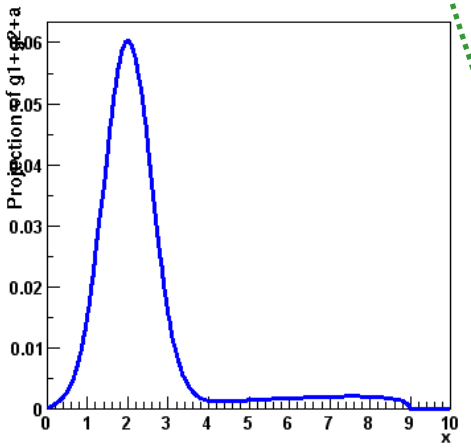
How to replicate and customize p.d.f – Cumbersome by hand…

x	type
0.73	A
0.42	A
0.33	A
1.52	B
0.29	B
0.98	B
0.54	B

fA *G(x;m,sA)+
(1-fA)*A(x;a,c)



fB *G(x;m,sB)+
(1-fB)*A(x;a,c)




```
RooRealVar m("m","mean of gaussian",-10,10) ;
RooRealVar s_A("s_A","sigma of gaussian A",0,20) ;
RooGaussian gauss_A("gauss_A","gaussian A",X,m,s_A) ;
RooRealVar s_B("s_B","sigma of gaussian B",0,20) ;
RooGaussian gauss_B("gauss_B","gaussian B",X,m,s_B) ;

RooRealVar k ("k","ArgusBG kappa parameter",-50,0) ;
RooRealVar xm("xm","ArgusBG cutoff point",9.0) ;
RooArgusBG argus_A ("argus","argus background",X,k,xm) ;
RooArgusBG argus_B … (省略)

RooRealVar f_A("f_A","fraction of gaussian A",0.,1.) ;
RooAddPdf pdf_A("pdf_A","gauss_A+argus",RooArgList(gauss_A,argus_A),f_A) ;
RooRealVar f_B("f_A","fraction of gaussian A",0.,1.) ;
RooAddPdf pdf_B("pdf_B","gauss_B+argus",RooArgList(gauss_B,argus_B),f_B) ;

RooSimultaneous simPdf("simPdf","simPdf",type) ;
simPdf.addPdf(pdf_A,"A") ;
simPdf.addPdf(pdf_B,"B") ;
```

参考
$ROOTSYS/tutorials/roofit/rf501_simultaneouspdf.C

Simultaneous fit with different ranges

也同时拟合两个不同的区间，稍微改动 rf501_simultaneouspdf.C

```
[dongly@lxslc603 ~]$ diff rf501_simultaneouspdf.C $ROOTSYS/tutorials/roofit/rf501_simultaneouspdf.C

39,40d38
<   x.setRange("fitRange_control",-6,2);
<   x.setRange("fitRange_physics",-4,4);

113c112
<   simPdf.fitTo(combData,Range("fitRange"),SplitRange(kTRUE)) ;
---
>   simPdf.fitTo(combData) ;

121c120
<   RooPlot* frame1 = x.frame(Bins(30),Title("Physics sample"),Range("fitRange_physics")) ;
---
>   RooPlot* frame1 = x.frame(Bins(30),Title("Physics sample")) ;

134,135c133
<   RooPlot* frame2 = x.frame(Bins(30),Title("Control sample"),Range("fitRange_control")) ;
---
>   RooPlot* frame2 = x.frame(Bins(30),Title("Control sample")) ;
```

6 Likelihood calculation & minimization

- *Details on the likelihood calculation*
- *Using MINUIT and RooMinuit*
- *Profile likelihoods*

Fitting and likelihood minimization

- What happens when you do **`pdf->fitTo(*data)`**
  - 1) Construct object representing –log of (extended) likelihood
  - 2) Minimize likelihood w.r.t floating parameters using MINUIT

- Can also do these two steps explicitly by hand

```
// Construct function object representing -log(L)
RooNLLVar nll("nll","nll",pdf,data) ;

// Minimize nll w.r.t its parameters
RooMinuit m(nll) ;
m.migrad() ;
m.hesse() ;
```

Constructing the likelihood function

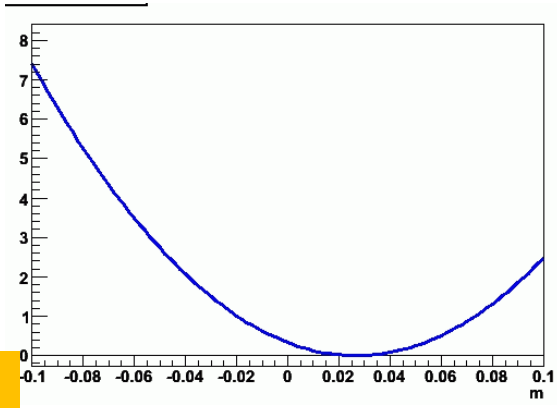
- Class **RooNLLVar** works universally for all p.d.f.s and all types of data
  - Binned data → Binned likelihood
  - Unbinned data → Unbinned likelihood

- Can add named arguments to constructor to control details of likelihood definition and mode of calculation

```
RooNLLVar nll("nll","nll",pdf,data,Extended()) ;
```

  - Works like a regular RooFit function object, i.e. can retrieve value and make plots as usual

```
Double_t val = nll.getVal() ;
RooArgSet* vars = nll.getVariables()

RooPlot* frame = p.frame() ;
nll.plotOn(frame) ;
```



参考 tutorials/roofit/rf606_nllerrorhandling.C

Constructing the likelihood function

- All of the following RooNLLVar options are available under identical name in **`pdf->fitTo()`**

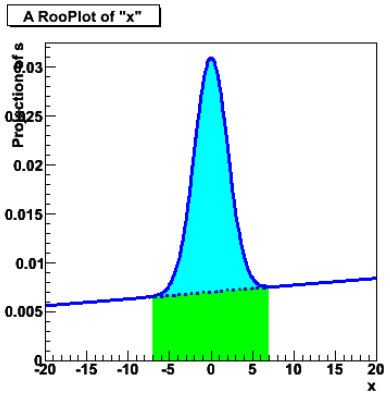
- Definition options
  - **`Extended()`** – Add extended likelihood term with N_{exp} taken from p.d.f and N_{obs} taken from data
  - **`ConditionalObservable(obs)`** – Treat given observables of pdf as conditional observables

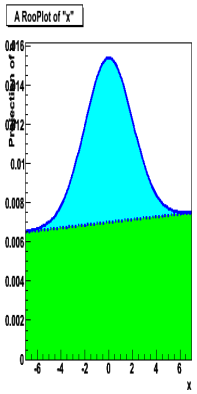
- Mode of calculation options
  - **`Verbose()`** – Additional information is printed on how the likelihood calculation is set up
  - **`NumCPU(N)`** – Parallelize calculation of likelihood over N processes. Nice if you have a dual-quad core box (actual speedup is about factor 7.6 for N=8)

Constructing the likelihood function

- Range options
  - **`Range("name")`** – Restrict likelihood to events in dataset that are within given named range definition. Note that if a Range() is fitted, all **`RooAddPdf`** components will keep their fraction coefficient interpretation in the full domain of the observables (unless they have a prior fixed definition)

This is default





With SumCoefRange("name") as well

  - **`SumCoefRange("name")`** – Instruct all `RooAddPdf` component of the p.d.f to interprett their fraction coefficients in the given range. Particularly useful in conjunction with Range()
  - **`SplitRange("name")`** – For use with simultaneous p.d.f.s. If given name of range applied will be "**`name_state`**" where state is the name of the index category of the top level **`RooSimultaneous`**

Constructing a χ^2 function

- Along similar lines it is also possible to construct a χ^2 function
  - Only takes binned datasets (class **RooDataHist**)
  - Normalized p.d.f is multiplied by Ndata to obtain χ^2

```
// Construct function object representing -log(L)
RooNLLVar chi2("chi2","chi2",pdf,data) ;

// Minimize nll w.r.t its parameters
RooMinuit m(chi2) ;
m.migrad() ;
m.hesse() ;
```

  - MINUIT error definition for χ^2 automatically adjusted to 1 (it is 0.5 for likelihoods) as default error level is supplied through virtual method of function base class **RooAbsReal**

Automatic optimizations in the calculation of the likelihood

- Several automatic computational optimizations are applied the calculation of likelihoods inside RooNLLVar
  - Components that have all constant parameters are pre-calculated
  - Dataset variables not used by the PDF are dropped
  - PDF normalization integrals are only recalculated when the ranges of their observables or the value of their parameters are changed
  - Simultaneous fits: When a parameters changes only parts of the total likelihood that depend on that parameter are recalculated
    - Lazy evaluation: calculation only done when intergal value is requested
- Applicability of optimization techniques is re-evaluated for each use
  - Maximum benefit for each use case
- 'Typical' large-scale fits see significant speed increase
  - Factor of 3x – 10x not uncommon.

Features of class RooMinuit

- Class RooMinuit is an *interface* to the ROOT implementation of the MINUIT minimization and error analysis package.

- RooMinuit takes care of
  - Passing value of miminized RooFit function to MINUIT
  - Propagated changes in parameters both from **`RooRealVar`** to MINUIT and back from MINUIT to **`RooRealVar`**, i.e. it keeps the state of RooFit objects synchronous with the MINUIT internal state
  - Propagate error analysis information back to **`RooRealVar`** parameters objects
  - Exposing high-level MINUIT operations to RooFit uses (MIGRAD,HESSE,MINOS) etc…
  - Making optional snapshots of complete MINUIT information (e.g. convergence state, full error matrix etc)

A brief description of MINUIT functionality

- MIGRAD
  - Find function minimum. Calculates function gradient, follow to (local) minimum, recalculate gradient, iterate until minimum found
    - To see what MIGRAD does, it is very instructive to do RooMinuit::setVerbose(1). It will print a line for each step through parameter space
  - Number of function calls required depends greatly on number of floating parameters, distance from function minimum and shape of function
- HESSE
  - Calculation of error matrix from 2nd derivatives at minimum
  - Gives symmetric error. Valid in assumption that likelihood is (locally parabolic)

$$\hat{\sigma}(p)^2 = \hat{V}(p) = \left(\frac{d^2 \ln L}{d^2 p} \right)^{-1}$$

  - Requires roughly N^2 likelihood evaluations (with N = number of floating parameters)

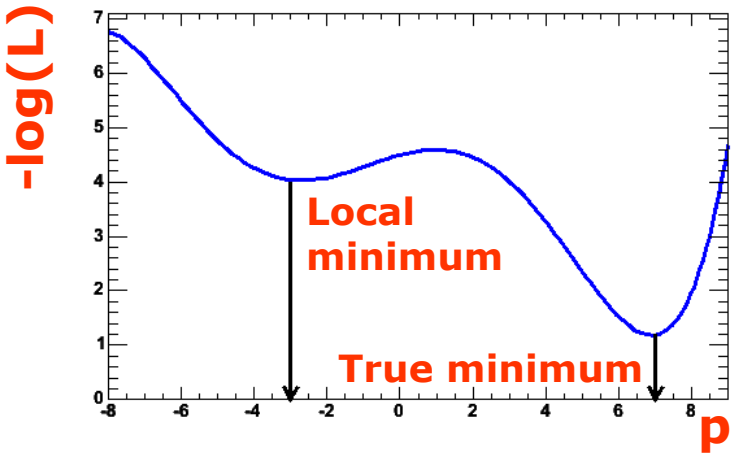
A brief description of MINUIT functionality

- MINOS
  - Calculate errors by explicit finding points (or contour for >1D) where Δ-log(L)=0.5
  - Reported errors can be asymmetric
  - Can be very expensive in with large number of floating parameters

- CONTOUR
  - Find contours of equal Δ-log(L) in two parameters and draw corresponding shape
  - Mostly an interactive analysis tool

Note of MIGRAD function minimization

- For all but the most trivial scenarios it is not possible to automatically find reasonable starting values of parameters
  - So you need to supply 'reasonable' starting values for your parameters



Reason: There may exist multiple (local) minima in the likelihood or χ^2

  - You may also need to supply 'reasonable' initial step size in parameters. (A step size 10x the range of the above plot is clearly unhelpful)
  - Using RooMinuit, the initial step size is the value of `RooRealVar::getError()`, so you can control this by supplying initial error values

Illustration of difference between HESSE and MINOS errors

- 'Pathological' example likelihood with multiple minima and non-parabolic behavior

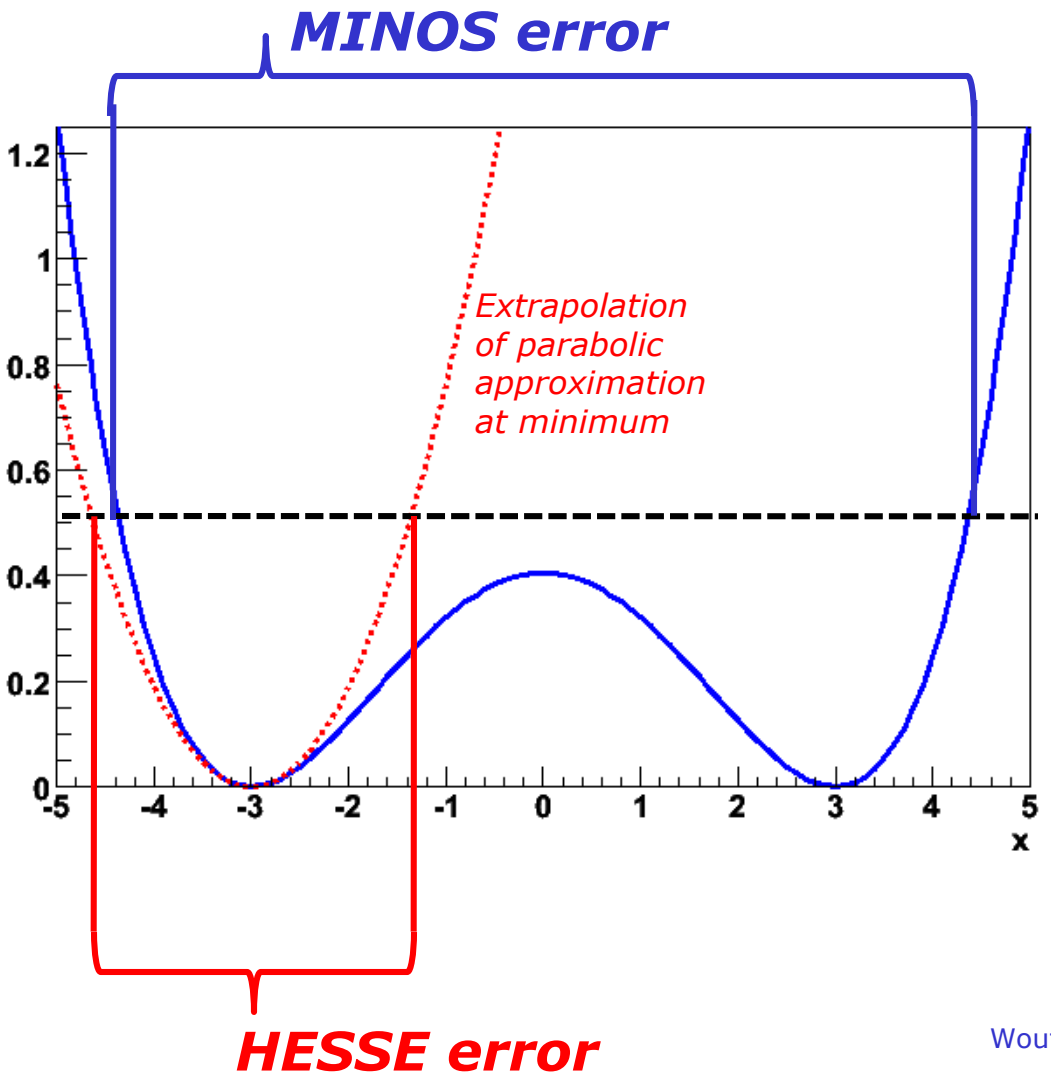
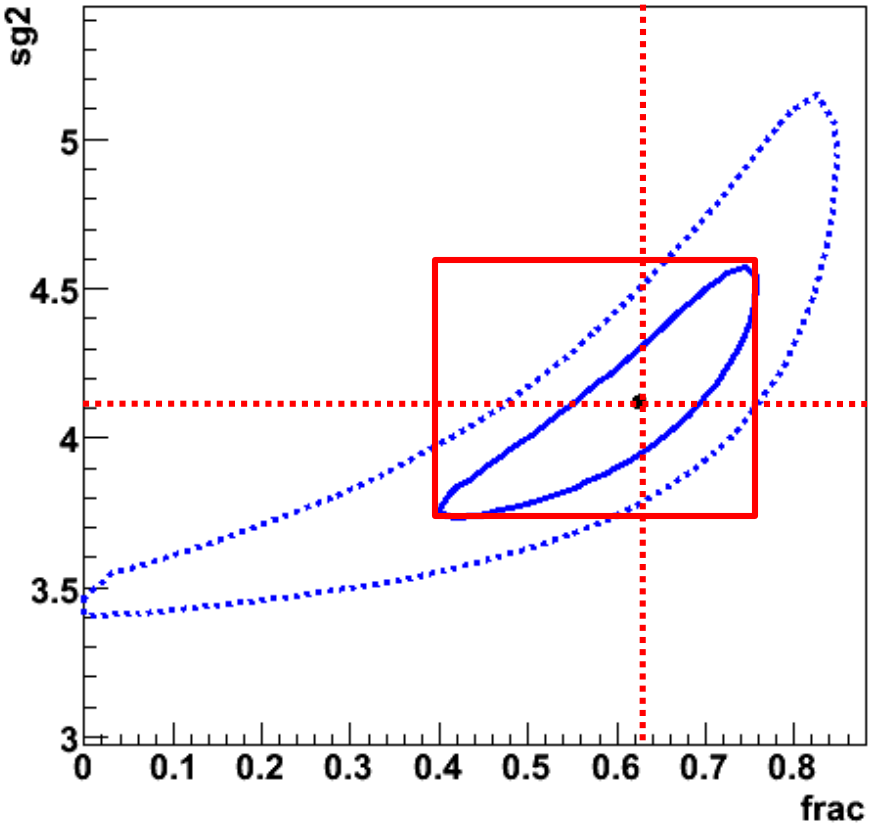


Illustration of MINOS errors in 2 dimensions

- Now we have a contour of Δnll instead of two points
- MINOS errors on px,py now defined by box enclosing contour



Demonstration of RooMinuit use

```
// Start Minuit session on above nll
RooMinuit m(nll) ;

// MIGRAD likelihood minimization
m.migrad() ;

// Run HESSE error analysis
m.hesse() ;

// Set sx to 3, keep fixed in fit
sx.setVal(3) ;
sx.setConstant(kTRUE) ;

// MIGRAD likelihood minimization
m.migrad() ;

// Run MINOS error analysis
m.minos()

// Draw 1,2,3 'sigma' contours in sx,sy
m.contour(sx,sy) ;
```

Minuit function MIGRAD

- Purpose: find minimum

```
**********
**    13 **MIGRAD        1000           1
**********
(some output omitted)
MIGRAD MINIMIZATION HAS CONVERGED.
MIGRAD WILL VERIFY CONVERGENCE AND ERROR MATRIX.
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=257.304 FROM MIGRAD    STATUS=CONVERGED      31 CALLS          32 TOTAL
                     EDM=2.36773e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  mean        8.84225e-02   3.23862e-01   3.58344e-04  -2.24755e-02
  2  sigma       3.20763e+00   2.39540e-01   2.78628e-04  -5.34724e-02
                               ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 1.049e-01  3.338e-04
 3.338e-04  5.739e-02
PARAMETER  CORRELATION COEFFICIENTS
      NO.   GLOBAL      1      2
       1  0.00430   1.000  0.004
       2  0.00430   0.004  1.000
```

Progress information, watch for errors here

Parameter values and approximate errors reported by MINUIT

Error definition (in this case 0.5 for a likelihood fit)

Minuit function MIGRAD

- Purpose: find minimum

```
**********
**    13 **MIGR
**********
 (some output o
MIGRAD MINIMIZ
MIGRAD WILL VERIF                    ENCE AND ERROR MATRIX.
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=257.304 FROM MIGRAD    STATUS=CONVERGED      31 CALLS          32 TOTAL
                    EDM=2.36773e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  mean         8.84225e-02   3.23862e-01   3.58344e-04  -2.24755e-02
   2  sigma        3.20763e+00   2.39540e-01   2.78628e-04  -5.34724e-02
                                ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 1.049e-01  3.338e-04
 3.338e-04  5.739e-02
PARAMETER  CORRELATION COEFFICIENTS
       NO.  GLOBAL      1      2
        1  0.00430   1.000  0.004
        2  0.00430   0.004  1.000
```

Value of $\chi 2$ or likelihood at minimum

(NB: χ^2 values are not divided by $N_{d.o.f}$)

Approximate Error matrix And covariance matrix

Minuit function MIGRAD

- Purpose: find minimu

***Status*:**
Should be 'converged' but can be 'failed'

Estimated Distance to Minimum
should be small O(10^{-6})

Error Matrix Quality
should be 'accurate', but can be 'approximate' in case of trouble

```
**********
**   13 **MIGRAD        1000
**********
(some output omitted)
MIGRAD MINIMIZATION HAS CONVERGED
MIGRAD WILL VERIFY CONVERGENCE AND E     MATRIX.
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=257.304 FROM MIGRAD    STATUS=CONVERGED      31 CALLS          32 TOTAL
                    EDM=2.36773e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  mean         8.84225e-02   3.23862e-01   3.58344e-04  -2.24755e-02
  2  sigma        3.20763e+00   2.39540e-01   2.78628e-04  -5.34724e-02
                               ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 1.049e-01  3.338e-04
 3.338e-04  5.739e-02
PARAMETER  CORRELATION COEFFICIENTS
      NO.  GLOBAL      1      2
       1  0.00430   1.000  0.004
       2  0.00430   0.004  1.000
```

Minuit function HESSE

- Purpose: calculate error matrix from $\frac{d^2L}{dp^2}$

```
**********
**    18 **HESSE        1000
**********
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=257.304 FROM HESSE     STATUS=OK                          OTAL
                    EDM=2.36534e-06    STRAT                CURATE
 EXT PARAMETER                                   TERNAL      INTERNAL
 NO.   NAME      VALUE            ERROR          STEP SIZE       VALUE
  1  mean          8.84225e-02   3.23861e-01   7.16689e-05   8.84237e-03
  2  sigma         3.20763e+00   2.39539e-01   5.57256e-05   3.26535e-01
                               ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 1.049e-01  2.780e-04
 2.780e-04  5.739e-02
PARAMETER  CORRELATION COEFFICIENTS
      NO.   GLOBAL      1      2
       1  0.00358   1.000  0.004
       2  0.00358   0.004  1.000
```

Symmetric errors calculated from 2nd derivative of –ln(L) or χ^2

Minuit function HESSE

- Purpose: calculate error matrix from $\frac{d^2L}{dp^2}$

```
**********
**
***
COV                                    UCCESSFULLY
FCN                                    TUS=OK              10 CALLS          42 TOTAL
                                       4e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
 EX                                                  INTERNAL      INTERNAL
 NO                                      ERROR       STEP SIZE       VALUE
  1                                   3.23861e-01    7.16689e-05    8.84237e-03
  2   sig                3.20763e+00  2.39539e-01    5.57256e-05    3.26535e-01
                                    ERR DEF= 0.5
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=0.5
 1.049e-01  2.780e-04
 2.780e-04  5.739e-02
PARAMETER  CORRELATION COEFFICIENTS
       NO.   GLOBAL      1      2
        1   0.00358   1.000  0.004
        2   0.00358   0.004  1.000
```

Error matrix (Covariance Matrix) calculated from

$$V_{ij} = \left(\frac{d^2(-\ln L)}{dp_i dp_j} \right)^{-1}$$

Minuit function HESSE

- Purpose: calculate error matrix from $\frac{d^2L}{dp^2}$

```
**********
**   18 **HESSE        1000
**********
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=257.304 FROM HESSE     STATUS=OK             10 CALLS          42 TOTAL
                    EDM=2.36534e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   INTERNAL      INTERNAL
 NO.   NAME      VALUE            ERROR          STEP SIZE       VALUE
  1   mean         8.84225e-02                                  8.84237e-03
  2   sigma        3.20763e+00                                  3.26535e-01

EXTERNAL ERROR MATRIX.    NDIM
 1.049e-01  2.780e-04
 2.780e-04  5.739e-02
PARAMETER  CORRELATION COEFFICIENT
      NO.   GLOBAL      1      2
        1   0.00358   1.000  0.004
        2   0.00358   0.004  1.000
```

F=0.5

Correlation matrix ρ_{ij} calculated from

$$V_{ij} = \sigma_i \sigma_j \rho_{ij}$$

Minuit function HESSE

- Purpose: calculate error matrix from $\frac{d^2L}{dp^2}$

```
**********
**   18 **HESSE        1000
**********
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=257.304 FROM HESSE     STATUS=OK             10 CALLS          42 TOTAL
                    EDM=2.36534e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                   INTERNAL      INTERNAL
 NO.   NAME      VALUE            ERROR          STEP SIZE       VALUE
  1   mean                                      7.16689e-05   8.84237e-03
  2   sigma                                     5.57256e-05   3.26535e-01

EXTERNAL ERROR                                  2    ERR DEF=0.5
 1.049e-01  2.7
 2.780e-04  5.739e-
PARAMETER  CORRELA    N COEFFICIENTS
      NO.   GLOBAL          1       2
        1   0.00358     1.000   0.004
        2   0.00358     0.004   1.000
```

Global correlation vector: correlation of each parameter with *all other* parameters

Minuit function MINOS

- Error analysis through Δnll contour finding

```
**********
**    23 **MINOS         1000
**********
FCN=257.304 FROM MINOS      STATUS=SUCCESSFUL      52 CALLS           94 TOTAL
                     EDM=2.36534e-06    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                          PARABOLIC        MINOS ERRORS
 NO.   NAME       VALUE                    ERROR      NEGATIVE      POSITIVE
  1   mean          8.84225e-02   3.23861e-01   -3.24688e-01    3.25391e-01
  2   sigma         3.20763e+00   2.39539e-01   -2.23321e-01    2.58893e-01
                                ERR DEF= 0.5
```

Symmetric error

(repeated result from HESSE)

MINOS error
Can be asymmetric

(in this example the 'sigma' error is slightly asymmetric)

What happens if there are problems in the NLL calculation

- Sometimes the likelihood cannot be evaluated do due an error condition.
  - PDF Probability is zero, or less than zero at coordinate where there is a data point 'infinitely improbable'
  - Normalization integral of PDF evaluates to zero
- Most problematic during MINUIT operations. How to handle error condition
  - All error conditions are gather and reported in consolidated way by RooMinuit
  - Since MINUIT has no interface deal with such situations, RooMinuit passes instead a large value to MINUIT to force it to retreat from the region of parameter space in which the problem occurred

```
[#0] WARNING:Minization -- RooFitGlue: Minimized function has error status.
Returning maximum FCN so far (99876) to force MIGRAD to back out of this region.
Error log follows. Parameter values: m=-7.397
RooGaussian::gx[ x=x mean=m sigma=sx ] has 3 errors
```

What happens if there are problems in the NLL calculation

- Can request more verbose error logging to debug problem
  - Add PrintEvalError(N) with N>1

```
[#0] WARNING:Minization -- RooFitGlue: Minimized function has error status.
Returning maximum FCN so far (-1e+30) to force MIGRAD to back out of this region.
Error log follows
Parameter values: m=-7.397
RooGaussian::gx[ x=x mean=m sigma=sx ]
     getLogVal() top-level p.d.f evaluates to zero or negative number
                 @ x=x=9.09989, mean=m=-7.39713, sigma=sx=0.1
     getLogVal() top-level p.d.f evaluates to zero or negative number
                 @ x=x=6.04652, mean=m=-7.39713, sigma=sx=0.1
     getLogVal() top-level p.d.f evaluates to zero or negative number
                 @ x=x=2.48563, mean=m=-7.39713, sigma=sx=0.1
```

Practical estimation – Fit converge problems

- Sometimes fits don't converge because, e.g.
  - MIGRAD unable to find minimum
  - HESSE finds negative second derivatives (which would imply negative errors)
- Reason is usually numerical precision and stability problems, but
  - The underlying cause of fit stability problems is usually by **highly correlated parameters** in fit
- HESSE correlation matrix in primary investigative tool

PARAMETER NO.	CORRELATION COEFFICIENTS GLOBAL	1	2
1	0.99835	1.000	0.998
2	0.99835	0.998	1.000

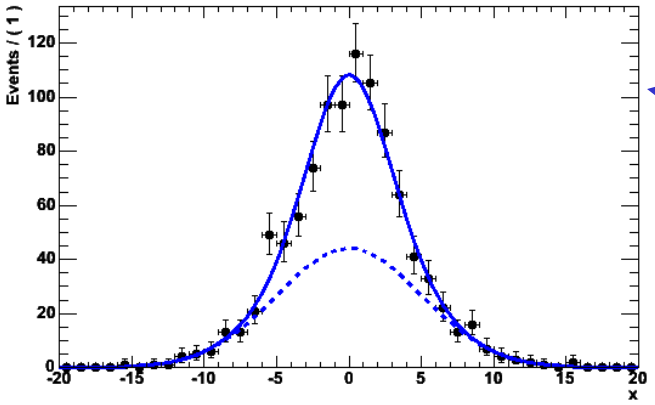
Signs of trouble…

  - In limit of 100% correlation, the usual point solution becomes a line solution (or surface solution) in parameter space. Minimization problem is no longer well defined

Mitigating fit stability problems

- Strategy I – More orthogonal choice of parameters
  - Example: fitting sum of 2 Gaussians of similar width

$$F(x; f, m, s_1, s_2) = fG_1(x; s_1, m) + (1-f)G_2(x; s_2, m)$$



HESSE correlation matrix

PARAMETER	CORRELATION COEFFICIENTS				
NO.	GLOBAL	[f]	[m]	[s1]	[s2]
[f]	0.96973	1.000	-0.135	0.918	0.915
[m]	0.14407	-0.135	1.000	-0.144	-0.114
[s1]	0.92762	0.918	-0.144	1.000	0.786
[s2]	0.92486	0.915	-0.114	0.786	1.000

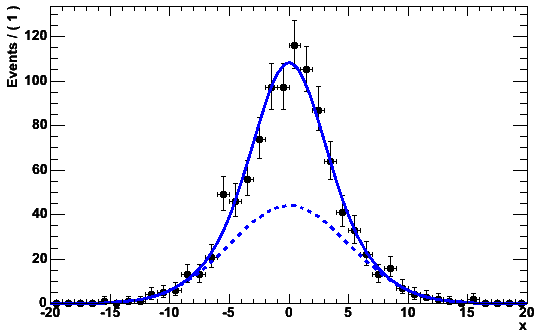
Widths s_1,s_2 strongly correlated fraction f

Mitigating fit stability problems

- Different parameterization:

$$fG_1(x;s_1,m_1)+(1-f)G_2(x;s_1\cdot s_2,m_2)$$

```
PARAMETER  CORRELATION COEFFICIENTS
      NO.  GLOBAL      [f]     [m]    [s1]    [s2]
 [ f]  0.96951   1.000 -0.134  0.917 -0.681
 [ m]  0.14312  -0.134  1.000 -0.143  0.127
 [s1]  0.98879   0.917 -0.143  1.000 -0.895
 [s2]  0.96156  -0.681  0.127 -0.895  1.000
```

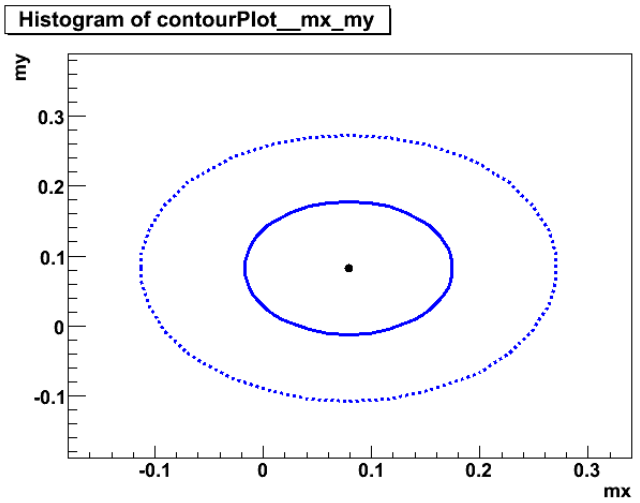


- Correlation of width s2 and fraction f reduced from 0.92 to 0.68
- Choice of parameterization matters!

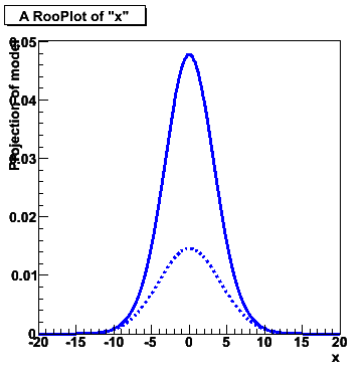
- Strategy II – Fix all but one of the correlated parameters
  - If floating parameters are highly correlated, some of them may be redundant and not contribute to additional degrees of freedom in your model

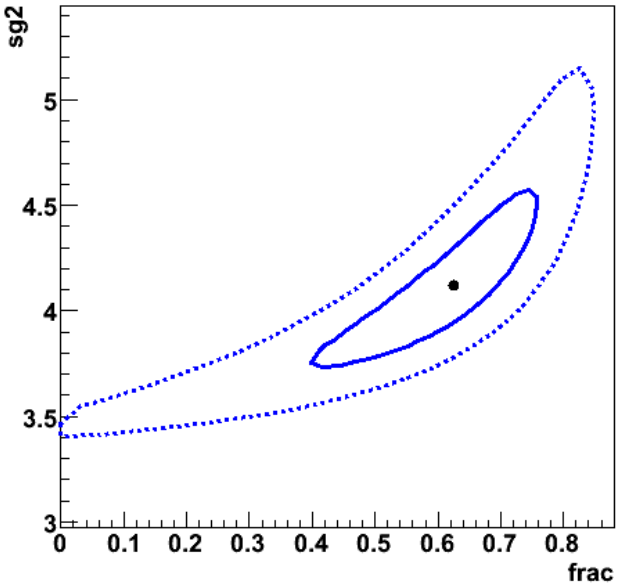
Minuit CONTOUR tool also useful to examine 'bad' correlations

- Example of 1,2 sigma contour of two uncorrelated variables
  - Elliptical shape. In this example parameters are uncorrelation



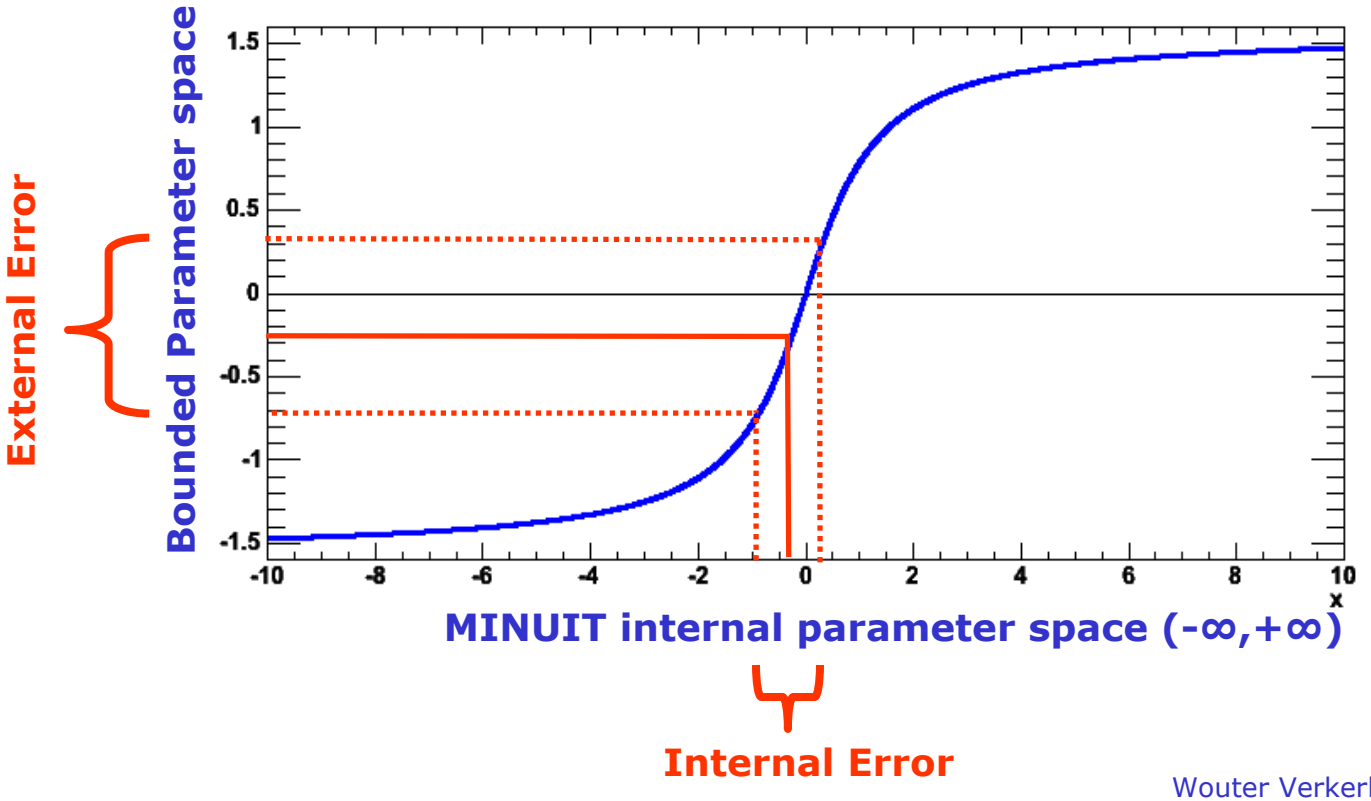
- Example of 1,2 sigma contour of two variables with problematic correlation
  - Pdf = f·G1(x,0,3)+(1-f)·G2(x,0,s) with s=4 in data





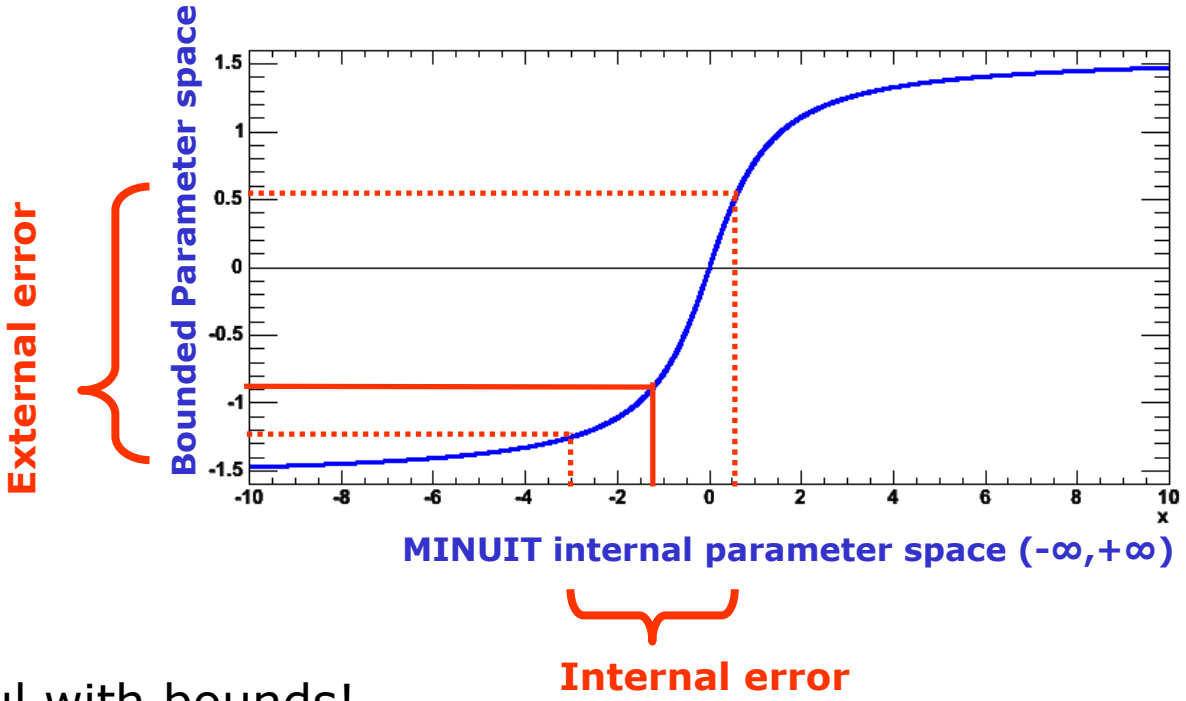
Practical estimation – Bounding fit parameters

- Sometimes is it desirable to bound the allowed range of parameters in a fit
  - Example: a fraction parameter is only defined in the range [0,1]
  - MINUIT option 'B' maps finite range parameter to an internal infinite range using an arcsin(x) transformation:



Practical estimation – Bounding fit parameters

- If fitted parameter values is close to boundary, errors will become asymmetric (and possible incorrect)



- So be careful with bounds!
  - If boundaries are imposed to avoid region of instability, look into other parameterizations that naturally avoid that region
  - If boundaries are imposed to avoid 'unphysical', but statistically valid results, consider not imposing the limit and dealing with the 'unphysical' interpretation in a later stage

Browsing fit results with `RooFitResult`

- As fits grow in complexity (e.g. 45 floating parameters), number of output variables increases
  - Need better way to navigate output that MINUIT screen dump
- **`RooFitResult`** holds complete snapshot of fit results
  - Constant parameters
  - Initial and final values of floating parameters
  - Global correlations & full correlation matrix
  - Returned from **`RooAbsPdf::fitTo()`** when "**`r`**" option is supplied
- Compact & verbose printing mode

Compact Mode

Constant parameters omitted in compact mode

Alphabetical parameter listing

```
fitres->Print() ;

  RooFitResult: min. NLL value: 1.6e+04, est. distance to min: 1.2e-05

    Floating Parameter    FinalValue +/-  Error
  --------------------  --------------------------
                argpar   -4.6855e-01 +/-  7.11e-02
                g2frac    3.0652e-01 +/-  5.10e-03
                 mean1    7.0022e+00 +/-  7.11e-03
                 mean2    1.9971e+00 +/-  6.27e-03
                 sigma    2.9803e-01 +/-  4.00e-03
```

Browsing fit results with **RooFitResult**

Verbose printing mode

```
fitres->Print("v") ;

  RooFitResult: min. NLL value: 1.6e+04, est. distance to min: 1.2e-05

  Constant Parameter    Value
  --------------------  ------------
                cutoff    9.0000e+00
                g1frac    3.0000e-01

    Floating Parameter  InitialValue    FinalValue +/-  Error      GblCorr.
  --------------------  ------------  --------------------------  --------
                argpar   -5.0000e-01   -4.6855e-01 +/-  7.11e-02  0.191895
                g2frac    3.0000e-01    3.0652e-01 +/-  5.10e-03  0.293455
                 mean1    7.0000e+00    7.0022e+00 +/-  7.11e-03  0.113253
                 mean2    2.0000e+00    1.9971e+00 +/-  6.27e-03  0.100026
                 sigma    3.0000e-01    2.9803e-01 +/-  4.00e-03  0.276640
```

Constant parameters listed separately

Initial,final value and global corr. listed side-by-side

Correlation matrix accessed separately

Browsing fit results with `RooFitResult`

- Easy navigation of correlation matrix
  - Select single element or complete row by parameter name

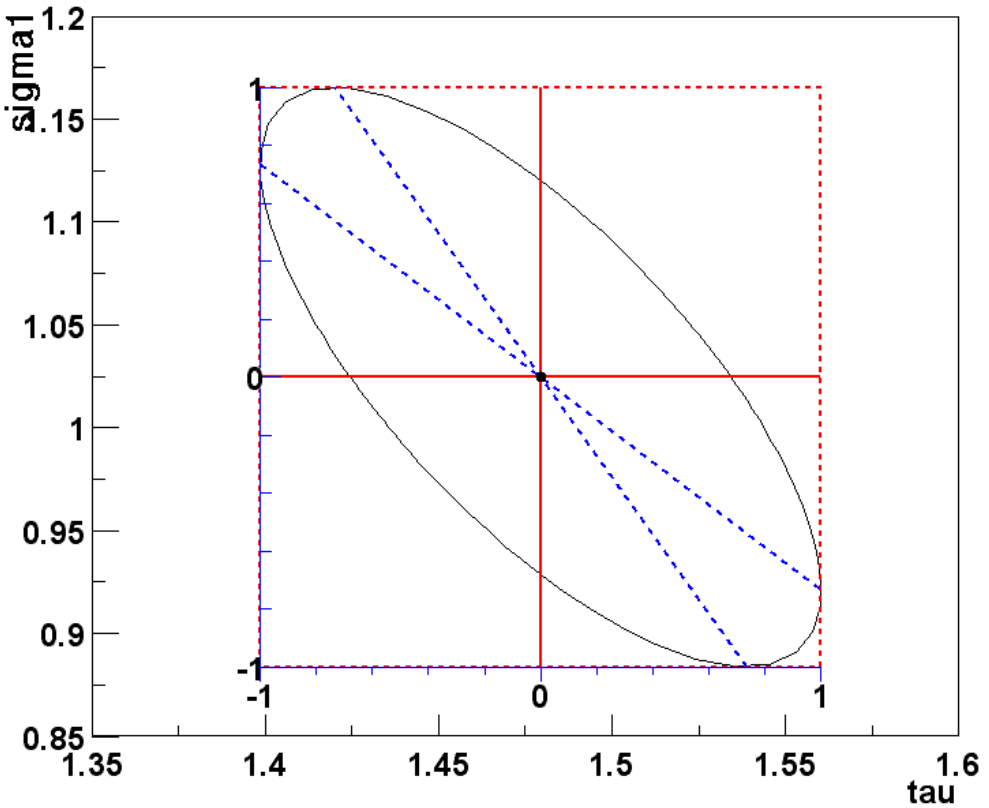
```
r->correlation("argpar","sigma")
(const Double_t)(-9.2560641200591084e-02)

r->correlation("mean1")->Print("v")
RooArgList::C[mean1,*]: (Owning contents)
  1) RooRealVar::C[mean1,argpar] :  0.11064 C
  2) RooRealVar::C[mean1,g2frac] : -0.0262487 C
  3) RooRealVar::C[mean1,mean1]  :  1.0000 C
  4) RooRealVar::C[mean1,mean2]  : -0.00632847 C
  5) RooRealVar::C[mean1,sigma]  : -0.0339814 C
```

- **`RooFitResult`** persistable with ROOT I/O
  - Save your batch fit results in a ROOT file and navigate your results just as easy afterwards

Visualize errors and correlation matrix elements

```
RooFitResult* r = pdf->fitTo(data,"mhvr") ;
RooPlot* f = new RooPlot(tau,sigma1,1.35,1.6,0.85,1.20) ;
r->plotOn(f,tau,sigma1,"ME12VHB") ;
f->Draw() ;
```

Works on any `RooFitResult`,
Also after persistence



MINUIT contour scan
is also possible with
a separate interface

Other uses of likelihood

- A likelihood may be considered the ultimate publication of a measurement
- Interesting to be able to digitally publish actual likelihood rather than
  - Parabolic version (i.e. you publish your measurement and an error)
  - Some parameterized form. Cumbersome in >1 dimension. No standard protocol for exchanging this time of information
- You can do this now in RooFit
  - You can persist your data (previously possible) and your actual p.d.f or likelihood into a ROOT file that anyone can read and use
- Many potential applications
  - Combining of Higgs channels, Heavy flavor averaging (CKMfitter) etc…

Using the Workspace concept

- Up to now, to share with colleagues need to distribute *both* a data file and a ROOT macro that builds the RooFit p.d.f
- Now add the Workspace – Persistent container for both data and functions

```
RooAbsPdf& g ; // from preceding example
RooAbsData& d ; // from preceding example
```

Create the workspace container object

```
RooWorkspace w("w","my workspace") ;
w.import(g) ;
w.import(d) ;
```

Use standard ROOT I/O to store wspace

```
TFile f("myresult.root","RECREATE") ;
w.Write() ;
f.Close() ;
```

- Both data and p.d.f. are now stored in file!

A look at the workspace

- What is in the workspace?

```
w.Print() ;
RooWorkspace(w) my workspace contents

variables
---------
(x,m,s)

p.d.f.s
-------
RooGaussian::g[ x=x mean=m sigma=s ] = 0

datasets
--------
RooDataSet::d(x)
```

```
RooRealVar* x = w.get("x") ;
```

```
RooAbsPdf* g = w.pdf("g") ;
```

```
RooAbsData* d = w.data("d") ;
```

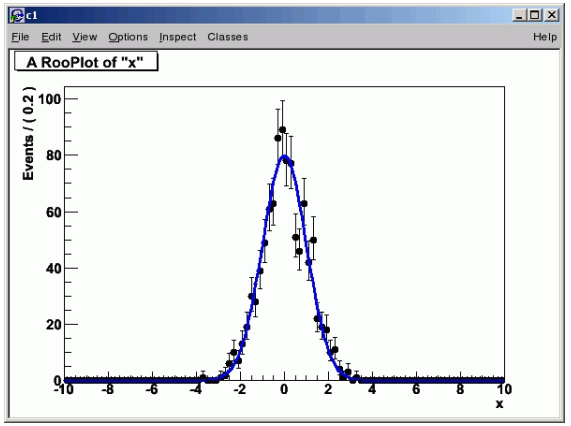
Using persisted p.d.f.s.

- Using both model & p.d.f from file

```
TFile f("myresults.root") ;
RooWorkspace* w = f.Get("w") ;
```

Make plot of data and p.d.f

```
RooPlot* xframe = w->var("x")->frame() ;
w->data("d")->plotOn(xframe) ;
w->pdf("g")->plotOn(xframe) ;
```

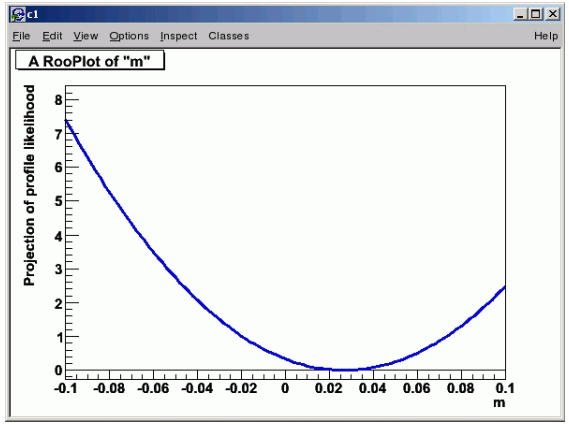


Construct likelihood & profile LH

```
RooNLLVar nll("nll","nll",*w->pdf("g"),*w->data("d")) ;
RooProfileLL pll("pll","pll", nll,*w->var("m")) ;
```

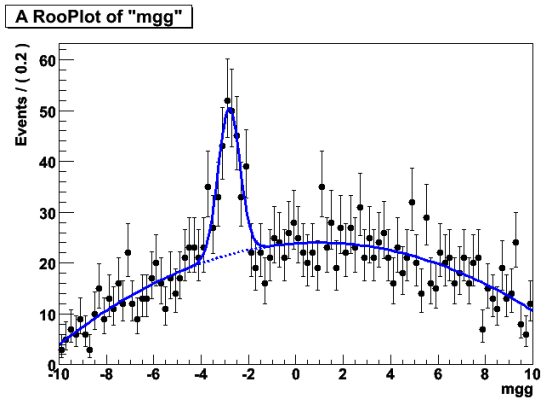
Draw profile LH

```
RooPlot* mframe = w->var("m")->frame(-1,1) ;
pll.plotOn(mframe) ;
mframe->Draw()
```



- **Note that above code is independent of actual p.d.f in file → e.g. full Higgs combination would work with identical code**

A combination example



- Code constructing 'ATLAS' p.d.f with signal and background and data
  - PDF is Gaussian + Chebychev

Create signal p.d.f

```
RooRealVar mgg("mgg","mgg",-10,10) ;
RooRealVar mHiggs("mHiggs","mHiggs",-3,-10,10) ;
RooRealVar sHiggs("sHiggs","sHiggs",0.5,0.01,10) ;
RooGaussian sig("sig","sig",mgg,mHiggs,sHiggs) ;
```

Create bkg p.d.f

```
RooRealVar a0("a0","a0",0.2,-1,1) ;
RooRealVar a1("a1","a1",-0.5,-1,1) ;
RooRealVar a2("a2","a2",0.02,-1,1) ;
RooChebychev bkg("bkg","bkg",mgg,RooArgList(a0,a1,a2)) ;
```

Create combined p.d.f

```
RooRealVar nHiggs("nHiggs","nHiggs",500,-100.,1000.) ;
RooRealVar nBkg("nBkg","nBkg",5000,-100.,10000.) ;
RooAddPdf model("model","model",RooArgList(sig,bkg),
                                RooArgList(nHiggs,nBkg)) ;
```

Generate toy data

```
RooDataSet* data = model.generate(mgg,NumEvents(2000),Name("data")) ;
```

A combination example

- Code writing 'ATLAS' p.d.f and data into Workspace
  - Variation of toy Gauss example:
    also create & persist likelihood function here

Fit model to data (skip MINOS)

```
model.fitTo(*data,Extended(),Minos(kFALSE)) ;
```

Create likelihood function

```
RooNLLVar nll("nll","nll",model,*data,Extended()) ;
```

Create the workspace container object

```
RooWorkspace atlas("atlas","atlas") ;
atlas.import(model) ;
atlas.import(*data) ;
atlas.import(nll) ;
```

Use standard ROOT I/O to store wspace

```
TFile f("atlas.root","RECREATE") ;
atlas.Write() ;
f.Close() ;
```

A combination example

- Contents of the ATLAS workspace

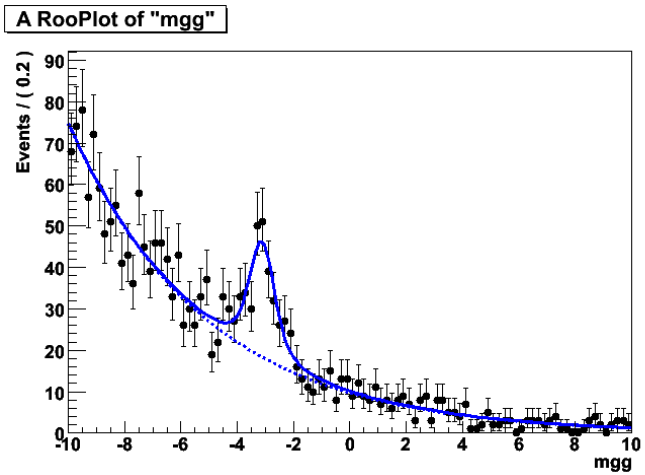
```
root [4] atlas->Print()

RooWorkspace(atlas) atlas contents

variables
---------
(mgg,mHiggs,sHiggs,nHiggs,a0,a1,a2,nBkg)

p.d.f.s
-------
RooAddPdf::model[ pdfs=(sig,bkg) coefficients=(nHiggs,nBkg) ] = 0
RooGaussian::sig[ x=mgg mean=mHiggs sigma=sHiggs ] = 0
RooChebychev::bkg[ x=mgg coefList=(a0,a1,a2) ] = 0

functions
---------
RooNLLVar::nll[ params=(mHiggs,sHiggs,a0,a1,a2,nHiggs,nBkg) ] = 0

datasets
--------
RooDataSet::data(mgg)
```

All component pdfs are visible as well

A combination example

- Similar code for 'CMS' – p.d.f. is Voigtian + Exponential

```
RooRealVar mgg("mgg","mgg",-10,10) ;
RooRealVar mHiggs("mHiggs","mHiggs",-3,-10,10) ;
RooRealVar wHiggs("wHiggs","wHiggs",0.8,0.1,10) ;
RooRealVar sHiggs("sHiggs","sHiggs",0.3) ;
RooVoigtian sig("sig","sig",mgg,mHiggs,wHiggs,sHiggs) ;
RooRealVar slope("slope","slope",-0.2,-100,1) ;
RooExponential bkg("bkg","bkg",mgg,slope) ;
RooRealVar nHiggs("nHiggs","nHiggs",500,-500.,10000.) ;
RooRealVar nBkg("nBkg","nBkg",5000,0.,10000.) ;
RooAddPdf model("model","model",RooArgList(sig,bkg),RooArgList(nHiggs,nBkg)) ;

RooDataSet* data = model.generate(mgg,NumEvents(2000),Name("data")) ;
model.fitTo(*data,Extended(),Minos(kFALSE)) ;
RooNLLVar nll("nll","nll",model,*data,Extended()) ;

RooWorkspace cms("cms","cms") ;
cms.import(model) ;
cms.import(*data) ;
cms.import(nll) ;
cms.Print() ;

TFile f("cms.root","RECREATE") ;
cms.Write() ;
f.Close() ;
```



A combination example

- Combining 'ATLAS' and 'CMS' result from persisted workspaces

Read ATLAS workspace

```
TFile* f = new TFile("atlas.root") ;
RooWorkspace *atlas = f->Get("atlas") ;
```

Read CMS workspace

```
TFile* f = new TFile("cms.root") ;
RooWorkspace *cms = f->Get("cms") ;
```

Construct combined LH

```
RooAddition nllCombi("nllCombi","nll CMS&ATLAS",
            RooArgSet(*cms->function("nll"),*atlas->function("nll"))) ;
```

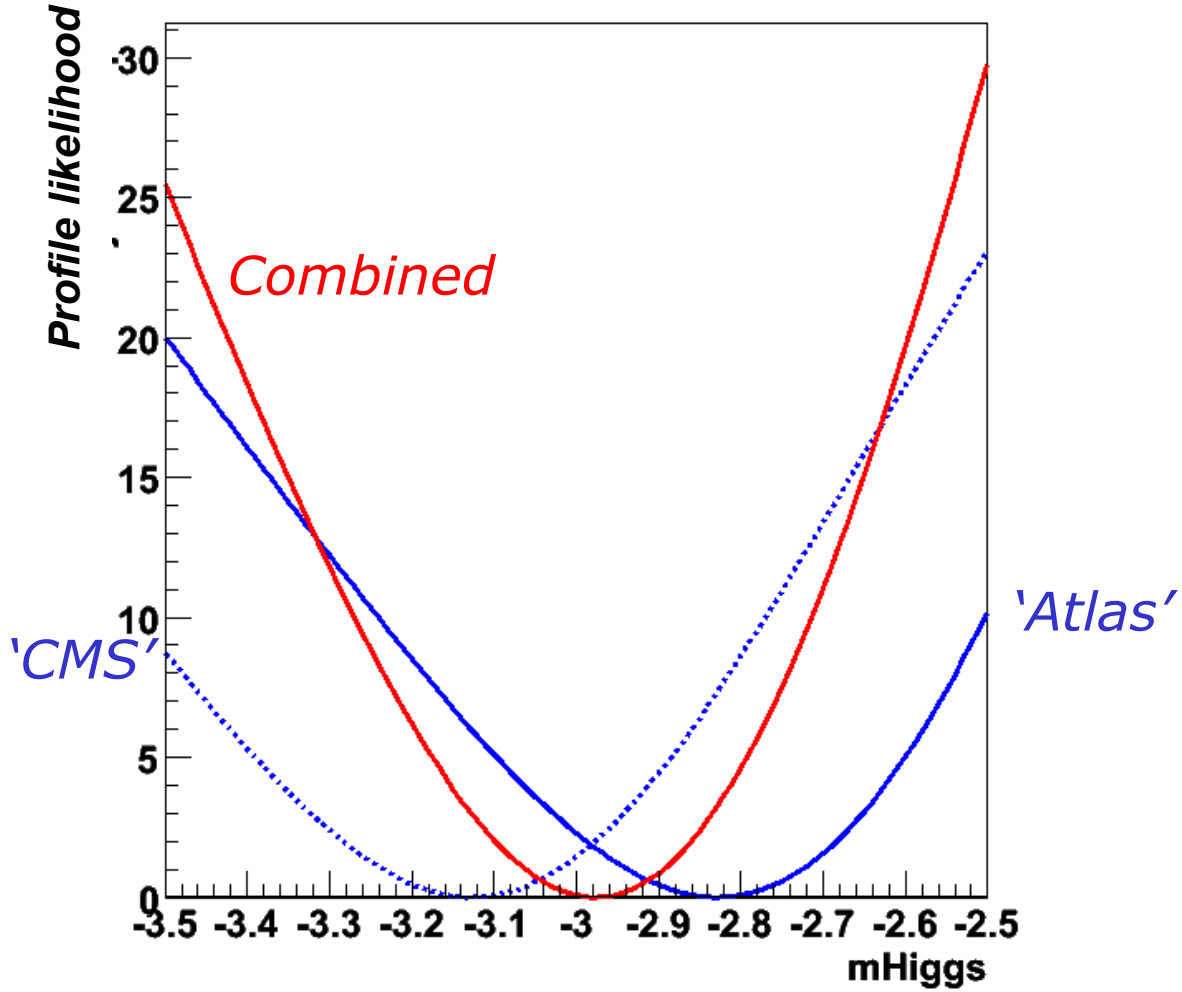
Construct profile LH in mHiggs

```
RooProfileLL pllCombi("pllCombi","pll",nllCombi,*atlas->var("mHiggs")) ;
```

Plot Atlas,CMS, combined profile LH

```
RooPlot* mframe = atlas->var("mHiggs")->frame(-3.5,-2.5) ;
atlas->function("nll")->plotOn(mframe)) ;
cms->function("nll")->plotOn(mframe),LineStyle(kDashed)) ;
pllCombi.plotOn(mframe,LineColor(kRed)) ;
mframe->Draw() ; // result on next slide
```

A combination example



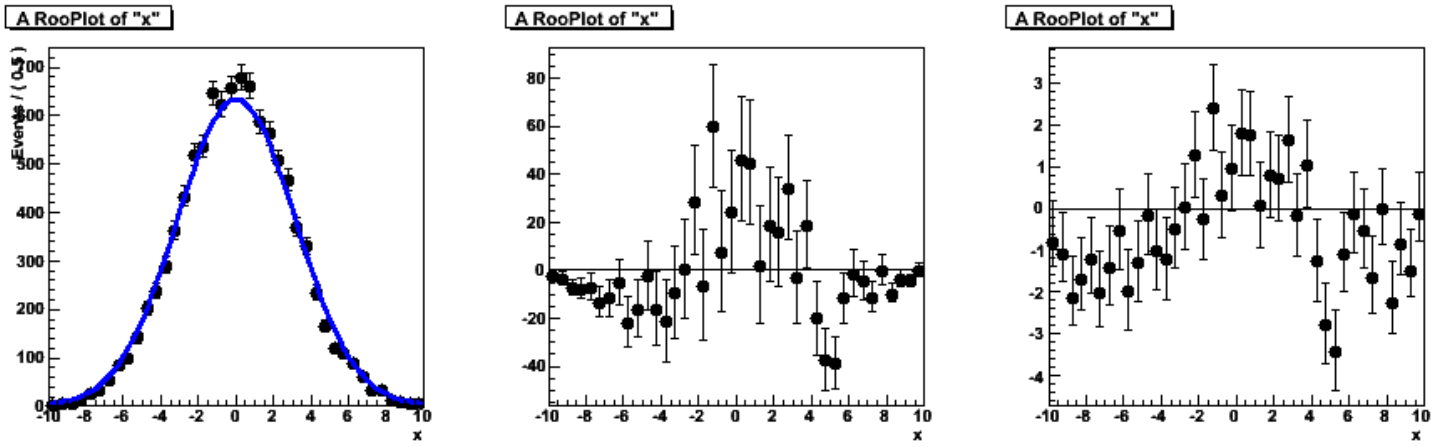
7

Fit validation, Toy MC studies

- *Goodness-of-fit, c2*
- *Toy Monte Carlo studies for fit validation*

How do you know if your fit was 'good'

- Goodness-of-fit broad issue in statistics in general, will just focus on a few specific tools implemented in RooFit here
- For one-dimensional fits, a χ^2 is usually the right thing to do
  - Some tools implemented in RooPlot to be able to calculate χ^2/ndf of curve w.r.t data



  - Also tools exists to plot residual and pull distributions from curve and histogram in a RooPlot

```
frame->makePullHist() ;
frame->makeResidHist() ;
```

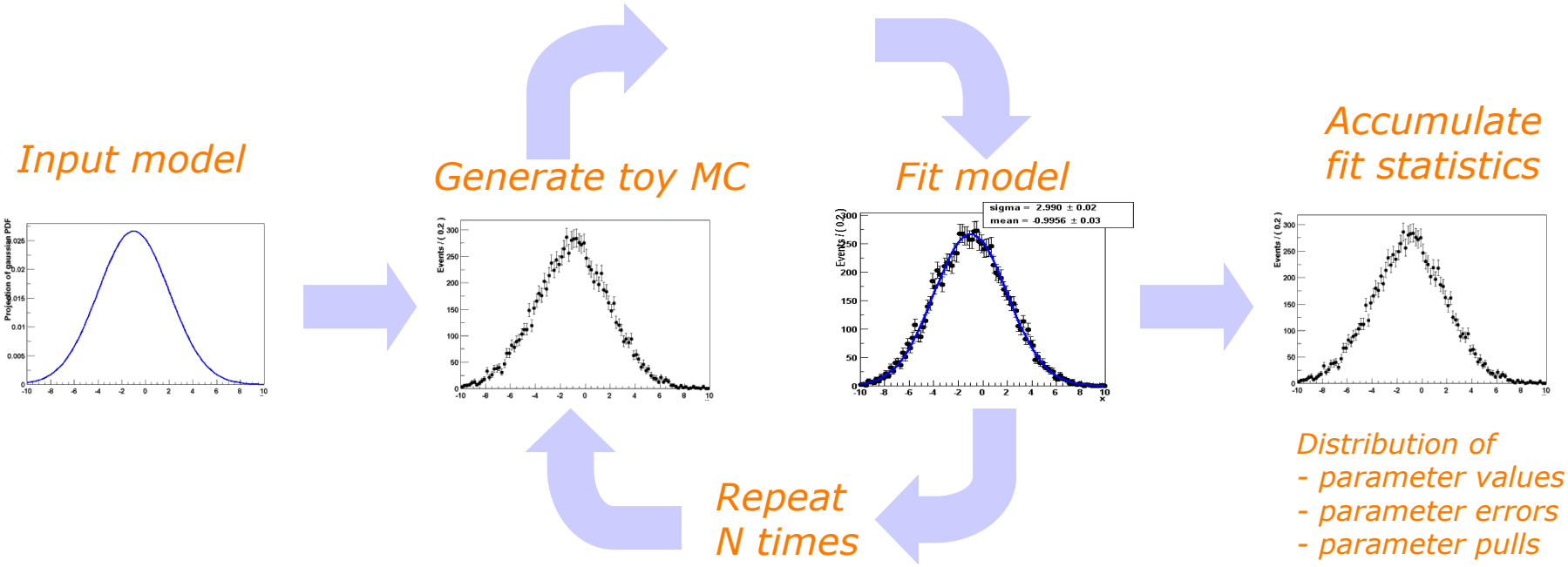
参考 tutorials/roofit/rf109_chi2residpull.C

GOF in >1D, other aspects of fit validity

- No special tools for >1 dimensional goodness-of-fit
  - A χ^2 usually doesn't work because empty bins proliferate with dimensions
  - But if you have ideas you'd like to try, there exists generic base classes for implementation that provide the same level of computational optimization and parallelization as is done for likelihoods (**`RooAbsOptTestStatistic`**)
- But you can study many other aspect of your fit validity
  - Is your fit unbiased?
  - Does it (often) have convergence problems?
- You can answer these with a toy Monte Carlo study
  - I.e. generate 10000 samples from your p.d.f., fit them all and collect and analyze the statistics of these 10000 fits.
  - The **`RooMCStudy`** class helps out with the logistics

Advanced features – Task automation

- Support for routine task automation, e.g. goodness-of-fit study




```
// Instantiate MC study manager
RooMCStudy mgr(inputModel) ;

// Generate and fit 100 samples of 1000 events
mgr.generateAndFit(100,1000) ;

// Plot distribution of sigma parameter
mgr.plotParam(sigma)->Draw()
```

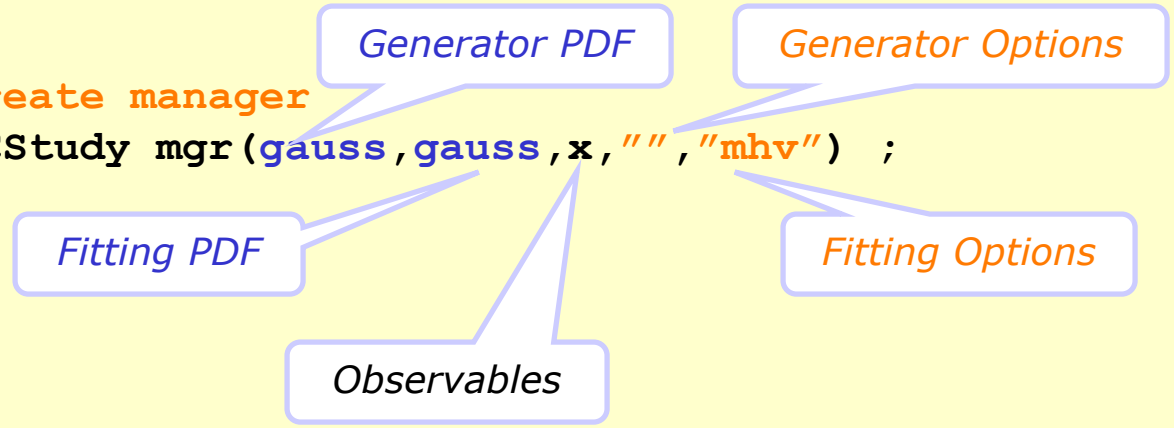
How to *efficiently* generate multiple sets of ToyMC?

- Use **`RooMCStudy`** class to manage generation and fitting
- Generating features
  - *Generator overhead only incurred once*
    → Efficient for large number of small samples
  - Optional Poisson distribution for #events of generated experiments
  - Optional automatic creation of ASCII data files
- Fitting
  - Fit with generator PDF or different PDF
  - Fit *results* (floating parameters & NLL) automatically *collected in summary dataset*
- Plotting
  - Automated plotting for distribution of parameters, parameter errors, pulls and NLL
- Add-in modules for optional modifications of procedure
  - Concrete tools for variation of generation parameters, calculation of likelihood ratios for each experiment
  - Easy to write your own. You can intervene at any stage and offer proprietary data to be aggregated with fit results

A RooMCStudy example

- Generating and fitting a simple PDF

```
// Setup PDF
RooRealVar x("x","x",-5,15) ;
RooRealVar mean("mean","mean of gaussian",-1) ;
RooRealVar sigma("sigma","width of gaussian",4) ;
RooGaussian gauss("gauss","gaussian PDF",x,mean,sigma) ;


// Create manager
RooMCStudy mgr(gauss,gauss,x,"","mhv") ;


// Generate and fit 1000 experiments of 100 events each
mgr.generateAndFit(1000,100) ;
RooMCStudy::run: Generating and fitting sample 999
RooMCStudy::run: Generating and fitting sample 998
RooMCStudy::run: Generating and fitting sample 997
…
```



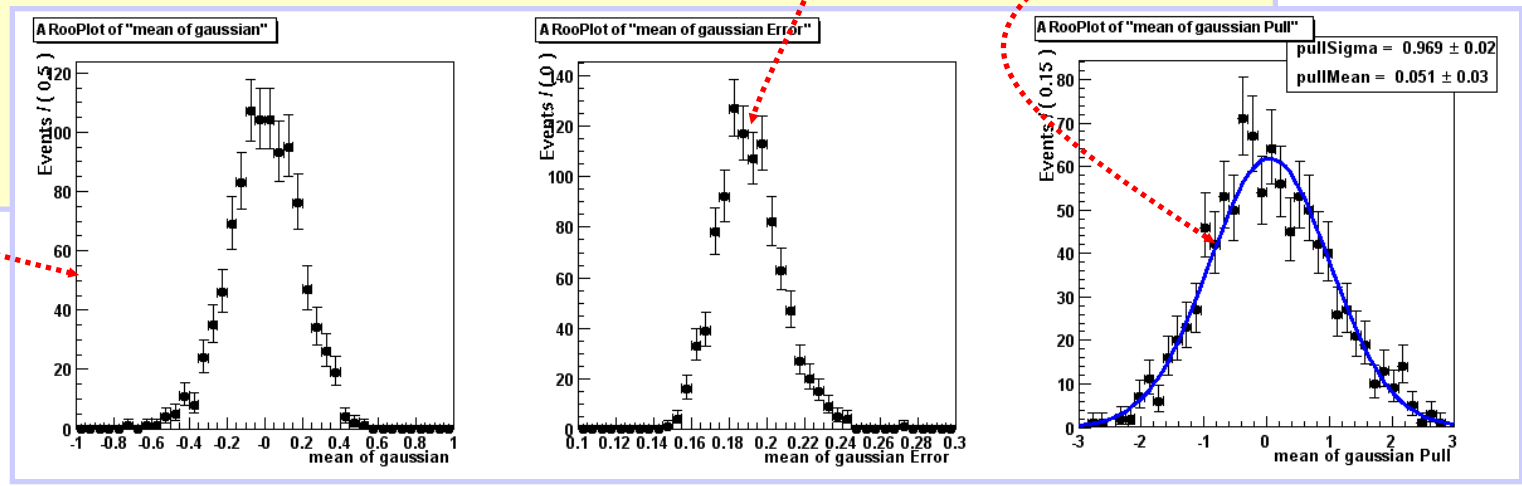
参考 tutorials/roofit/rf801_mcstudy.C

A RooMCStudy example

- Plot the distribution of the value, error and pull of *mean*

```
// Plot the distrution of the value
RooPlot* mframe = mean.frame(-1,1) ;
mgr.plotParamOn(mframe) ;
mframe->Draw() ;

// Plot the distrution of the error
RooPlot* meframe = mgr.plotError(mean,0.,0.1) ;
meframe->Draw() ;

// Plot the distrution of the pull
RooPlot* mpframe = mgr.plotPull(mean,-3,3,40,kTRUE) ;
mpframe->Draw() ;
```

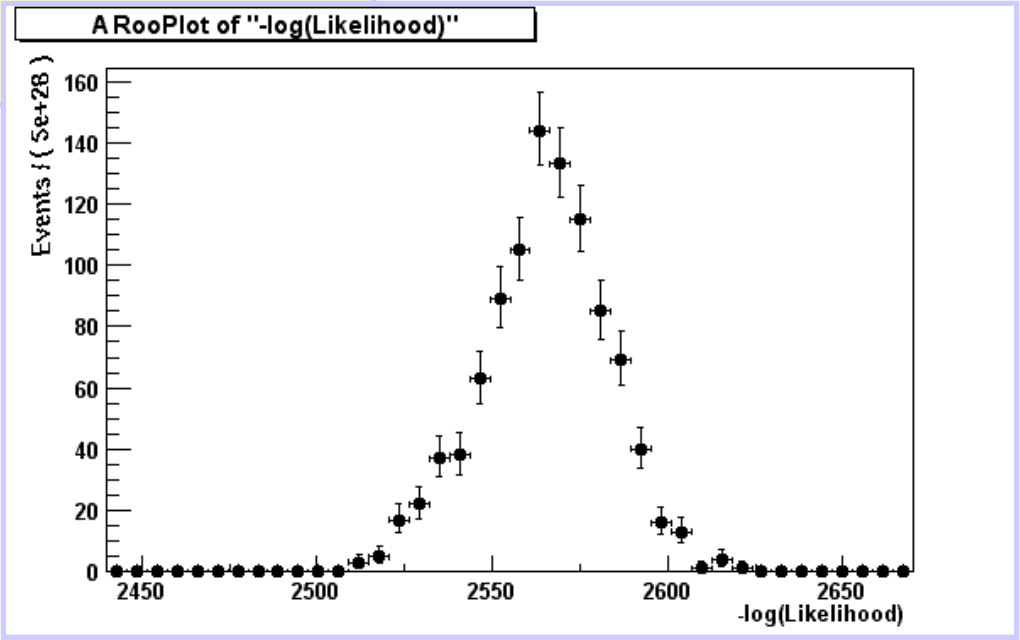
Add Gaussian fit



A RooMCStudy example

- Plot the distribution of –log(L)

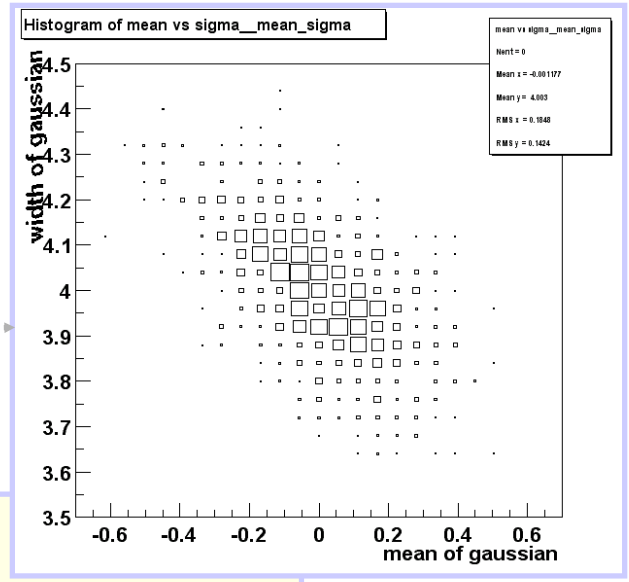
```
// Plot the distribution of the NLL
mgr.plotNLL(mframe) ;
mframe->Draw() ;
```



- NB: likelihood distributions cannot be used to deduce goodness-of-fit information!

A RooMCStudy example

- For other uses, use summarized fit results in **RooDataSet** form



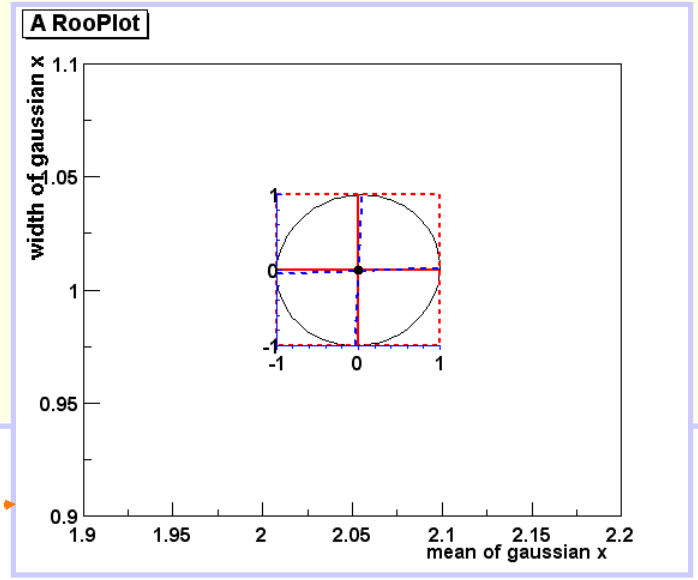

```
mgr.fitParDataSet().get(10)->Print("v") ;
RooArgSet:::
  1) RooRealVar::mean       :  0.14814 +/- 0.191 L(-10 - 10)
  2) RooRealVar::sigma      :  4.0619 +/- 0.143 L(0 - 20)
  3) RooRealVar::NLL        :  2585.1 C
  4) RooRealVar::meanerr    :  0.19064 C
  5) RooRealVar::meanpull   :  0.77704 C
  6) RooRealVar::sigmaerr   :  0.14338 C
  7) RooRealVar::sigmapull  :  0.43199 C

TH2* h = mean.createHistogram("mean vs sigma",sigma) ;
mgr.fitParDataSet().fillHistogram(h,RooArgList(mean,sigma)) ;
h->Draw("BOX") ;
```

Pulls and errors have separate entries for easy access and plotting

A RooMCStudy example

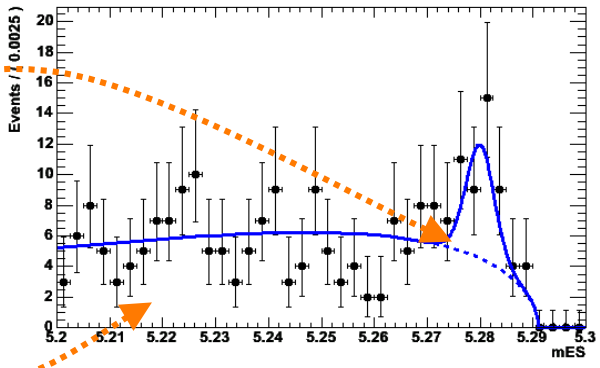
- If the "r" fit option is supplied
  the **RooFitResult** output of each fit is be saved

```
mgr.fitResult(10)->Print("v") ;

  RooFitResult: minimized NLL value: 2585.13, estimated distance to minimum: 3.18389e-06

    Floating Parameter  InitialValue    FinalValue +/-  Error     GblCorr.
  --------------------  ------------  --------------------------  --------
                  mean    0.0000e+00    1.4814e-01 +/-  1.91e-01  0.597596  <none>
                 sigma    4.0000e+00    4.0619e+00 +/-  1.43e-01  0.597596  <none>

mgr.fitResult(10)->correlation("sigma")->Print("v") ;

RooArgList::C[sigma,*]: (Owning contents)
  1) RooRealVar::C[sigma,mean]   : -0.597596 C
  2) RooRealVar::C[sigma,sigma] :  1.0000 C


RooPlot* frame = new RooPlot(…)
mgr.fitResult(10)->plotOn(frame,meanx,
                          sigmax,"ME12VHB") ;
```



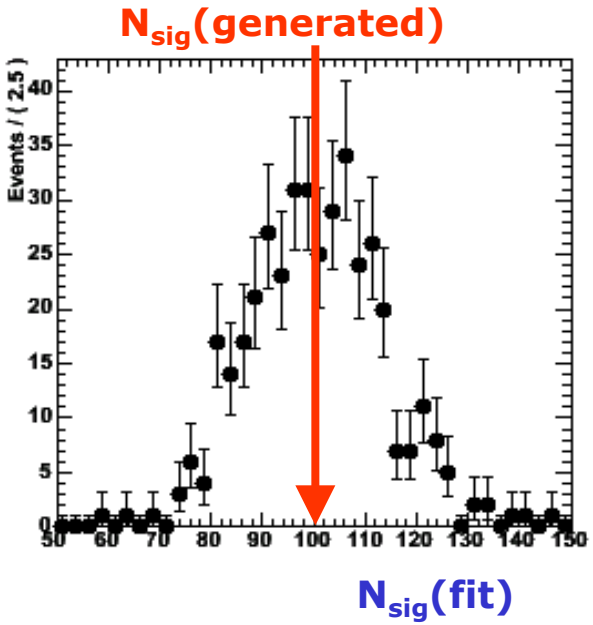
Fit Validation Study – Practical example

- Example fit model in 1-D (B mass)
  - Signal component is Gaussian centered at B mass
  - Background component is Argus function (models phase space near kinematic limit)



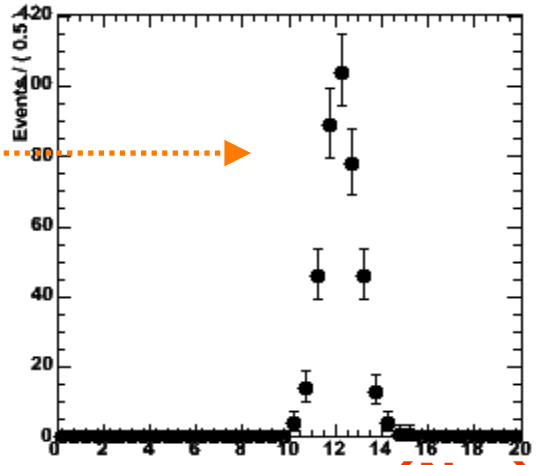
$$F(m; N_{\text{sig}}, N_{\text{bkg}}, \vec{p}_S, \vec{p}_B) = N_{\text{sig}} \cdot G(m; p_S) + N_{\text{bkg}} \cdot A(m; p_B)$$

- Fit parameter under study: N_{sig}
  - Results of simulation study: 1000 experiments with $\mathbf{N_{SIG}(gen)=100}$, $\mathbf{N_{BKG}(gen)=200}$
  - Distribution of N_{sig}(fit)
  - This particular fit looks unbiased…



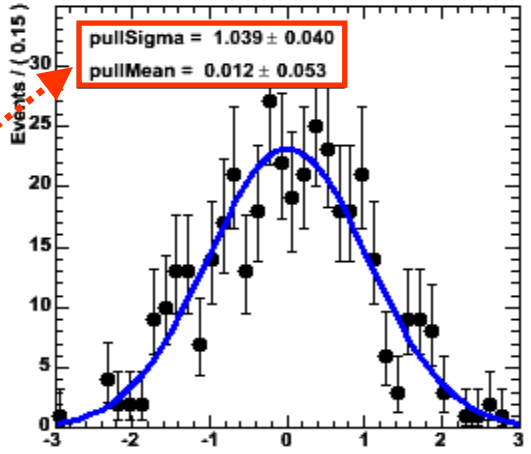
Fit Validation Study – The pull distribution

- What about the validity of the error?
  - Distribution of error from simulated experiments is difficult to interpret…
  - We don't have equivalent of N_{sig}(generated) for the error



$\sigma(N_{sig})$

- Solution: look at the ***pull distribution***
  - Definition: $$\text{pull}(N_{sig}) = \frac{N_{sig}^{fit} - N_{sig}^{true}}{\sigma_N^{fit}}$$
  - Properties of pull:
    - **Mean is 0 if there is no bias**
    - **Width is 1 if error is correct**
  - In this example: no bias, correct error within statistical precision of study



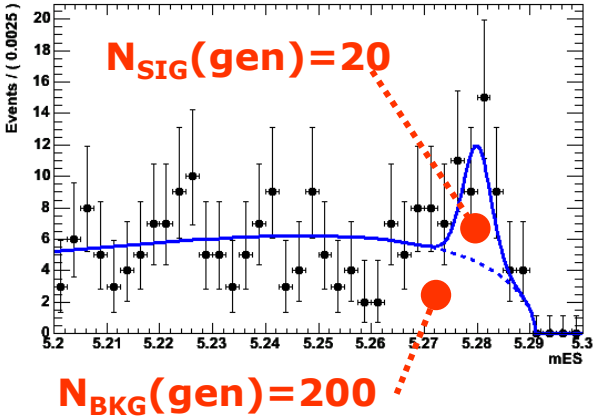
pull(N_{sig})

Fit Validation Study – Low statistics example

- Special care should be taken when fitting small data samples
  - Also if fitting for small signal component in large sample
- Possible causes of trouble
  - χ^2 estimators may become approximate as Gaussian approximation of Poisson statistics becomes inaccurate
  - ML estimators may no longer be efficient
    → error estimate from 2nd derivative may become inaccurate
  - Bias term proportional to 1/N of ML and χ^2 estimators may no longer be small compared to 1/sqrt(N)
- In general, absence of bias, correctness of error can not be assumed. How to proceed?
  - Use unbinned ML fits only – most robust at low statistics
  - **Explicitly verify the validity of your fit**

Demonstration of fit bias at low N – pull distributions

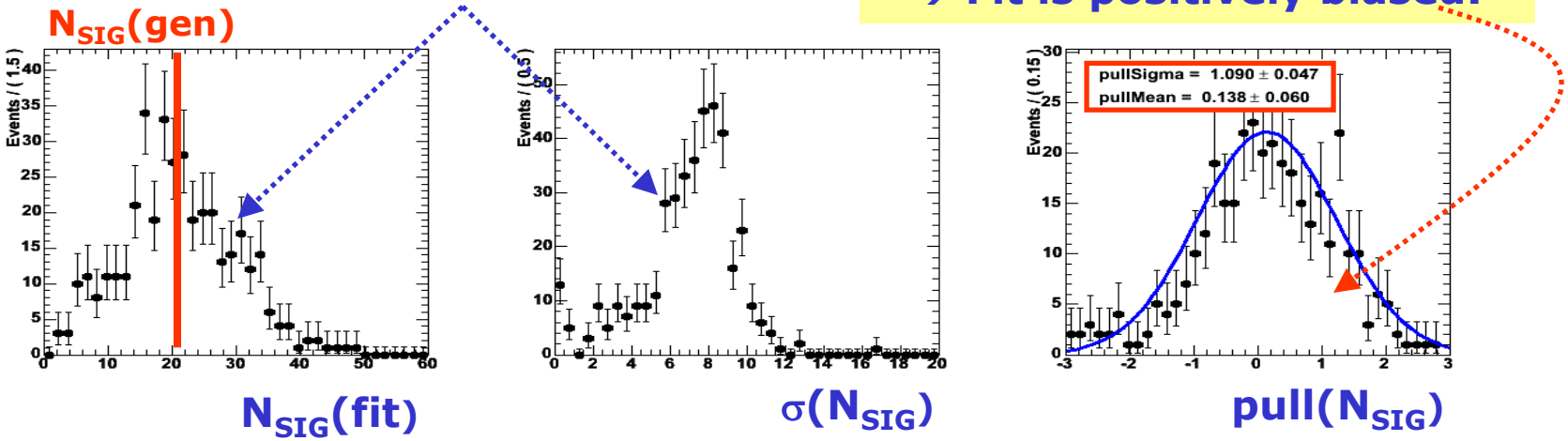
- Low statistics example:
  - Scenario as before but now with 200 bkg events and **only 20 signal events** (instead of 100)



- Results of simulation study

Distributions become asymmetric at low statistics

Pull mean ~2σ away from 0 → Fit is positively biased!



- *Absence of bias, correct error at low statistics not obvious*

10 Code examples

Implementing a RooAbsReal Providing analytical integrals Implementing a RooAbsPdf Providing an internal generator

Writing a real-valued function – class RooAbsReal

- Class declaration

Real-valued functions inherit from **RooAbsReal**

```
class RooUserFunc : public RooAbsReal {
public:
  RooUserFunc(const cha      e, const char *title,
              RooA       , RooAbsReal& _mean,
                    _sigma);
                         er,
  virtual TObject* clone(const char* newname) const {
          return new RooUserFunc(*this,newname);
  }
  inline virtual ~RooUserFunc() { }

protected:
  RooRealProxy x ;
  RooRealProxy mean ;
  RooRealProxy sigma ;

  Double_t evaluate() const ;

private:
  ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

Writing a function – class RooAbsReal

- Mandatory methods

- Constructor
- Copy constructor
- Clone
- Destructor
- **evaluate**

Calculates your PDF return value

```
class RooUserFunc : public RooAbsPdf {
public:
  RooUserFunc(const char *name, const char *title,
              RooAbsReal& _x, RooAbsReal& _mean,
              RooAbsReal& _sigma);
  RooUserFunc(const RooUserFunc& other,
              const char* name=0) ;
  virtual TObject* clone(const char* newname) const {
          return new RooUserFunc(*this,newname);
  }
  inline virtual ~RooUserFunc() { }

protected:
  RooRealProxy x ;
  RooRealProxy mean ;
  RooRealProxy sigma ;

  Double_t evaluate() const ;

private:
  ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

Use copy ctor in `clone()`

Writing a function – class RooAbsReal

- Constructor arguments

```
class RooUserFunc : public RooAbsPdf {
public:
  RooUserFunc(const char *name, const char *title,
              RooAbsReal& _x, RooAbsReal& _mean,
              RooAbsReal& _sigma);
  RooU          (const RooUserFunc& other,
                 const char* name=0) ;
                                                  const {
```

Try to be as generic as possible, i.e.

Use `RooAbsReal&` to receive real-valued arguments
Use `RooAbsCategory&` to receive discrete-valued arguments

Allows user to plug in either
a variable (**RooRealVar**) or a function (**RooAbsReal**)

```
private:
  ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

Writing a function – class RooAbsReal

- Storing **RooAbsArg** references

Always use proxies to store RooAbsArg references:

RooRealProxy	for	RooAbsReal
RooCategoryProxy	for	RooAbsCategory
RooSetProxy	for	a set of RooAbsArgs
RooListProxy	for	a list of RooAbsArgs

```
                                                    r *title,
                                                    & _mean,


                                                    wname) const {
                return new RooUserFunc(*this,newname);

        line virtual ~RooUserFunc() { }

pro  cted:
  RooRealProxy x ;
  RooRealProxy mean ;
  RooRealProxy sigma ;

  Double_t evaluate() const ;

private:
  ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

Storing references in proxies allows RooFit to adjust pointers

This is essential for cloning of composite objects

Writing a function – class RooAbsReal

- ROOT-CINT dictionary methods

```
class RooUserFunc : public RooAbsPdf {
public:
  RooUserFunc(const char *name, const char *title,
              RooAbsReal& _x, RooAbsReal& _mean,
              RooAbsReal& _sigma);
  RooUserFunc(const RooUserFunc& other,
              const char* name=0) ;
  virtual TObject* clone(const char* newname) const {
          return new RooUserFunc(*this,newname);
  }
  inline virtual ~RooUserFunc()

protected:
  RooRealProxy x ;

                                    ;

private:
  ClassDef(RooUserFunc,1) // Gaussian PDF
};
```

Don't forget ROOT `ClassDef` macro
No semi-colon at end of line!

Description here will be used in auto-generated `THtml` documentation

Writing a function – class RooAbsReal

- Constructor implementation

```
RooUserFunc::RooUserFunc(const char *name, const char *title,
                         RooAbsReal& _x, RooAbsReal& _mean,
                         RooAbsReal& _sigma) :
  RooAbsPdf(name,title),
  x("x","Dependent",this,_x),
  mean("mean","Mean",this,_mean),
  sigma("sigma","Width",this,_sigma)
{
```

Initialize the proxies from the **RooAbsArg** method arguments

Pointer to owning object is needed to register proxy

Name and title are for description only

```
() const
{
  Double_t arg= x - mean;
  return exp(-0.5*arg*arg/(sigma*sigma)) ;
}
```

Writing a function – class RooAbsReal

- **Implement a copy constructor!**

```
RooUserFunc::RooUserFunc(const char *name, const char *title,
                         RooAbsReal& _x, RooAbsReal& _mean,
                         RooAbsReal& _sigma) :
  RooAbsPdf(name,title),
```

In the class copy constructor, call all *proxy copy constructors*

```
}

RooUserFunc::RooU      ic(const RooUserFunc& other,
                          const char* name) :
  RooAbsPdf(other,name),
  x(this,other.x),
  mean(this,other.mean),
  sigma(this,other.sigma)
{
}

Double_t RooU
{
  Double_t ar
  return exp(-      gma)) ;
}
```

Pointer to owning object is (again) needed to register proxy

Writing a function – class RooAbsReal

- Write evaluate function

```
RooUserFunc::RooUserFunc(const char *name, const char *title,
                         RooAbsReal& _x, RooAbsReal& _mean,
                         RooAbsReal& _sigma) :
  RooAbsPdf(name,title),
  x("x","Dependent",this,_x),
  mean("mean","Mean",this,_mean),
  sigma("sigma","Width",this,_sigma)
{
}

RooUserFunc::RooUserFunc(const RooUserFunc& other,
                         const char* name) :
  RooAbsPdf(other,name),
  x("x",this,other.x),
  mean("mean",this,other.mean),
```

In `evaluate()`, calculate and return the function value

```
Double_t RooUserFunc::evaluate() const
{
  Double_t arg= x - mean;
  return exp(-0.5*arg*arg/(sigma*sigma)) ;
}
```

Working with proxies

- **`RooRealProxy/RooCategoryProxy`**
  objects automatically cast to the value type they represent
  - Use as if they were fundamental data types

```
RooRealProxy x ;
Double_t func = x*x ;
```

Use as `Double_t`

```
RooCategoryProxy c ;
if (c=="bogus") {…}
```

Use as `const char*`

- To access the proxied **`RooAbsReal/RooAbsCategory`** object use the **`arg()`** method

```
RooRealProxy x ;
RooCategoryProxy c ;

RooAbsReal& xarg = x.arg() ;
RooAbsCategory& carg = c.arg() ;
```

NB: the value or `arg()` may change during the lifetime of the object (e.g. if a composite cloning operation was performed)

- Set and list proxy operation completely transparent
  - Use as if they were **`RooArgSet/RooArgList`** objects

Lazy function evaluation & caching

- Method getVal() does not always call evaluate()
  - Each RooAbsReal object caches its last calculated function value
  - If none of the dependent values changed, no need to recalculate

- Proxies are used to track changes in objects
  - Whenever a RooAbsArg changes value,
    it notifies all its client objects that recalculation is needed
  - Messages passed via client/server links that are installed by proxies
  - Only if recalculate flag is set getVal() will call evaluate()

- Redundant calculations are automatically avoided
  - Efficient optimization technique for expensive objects like integrals
  - No need to hand-code similar optimization in function code:
    evaluate() is only called when necessary

Writing a function – analytical integrals

- **Analytical integrals are optional!**
- Implementation of analytical integrals is separated in two steps
  - Advertisement of available (partial) integrals:
  - Implementation of partial integrals
- Advertising integrals: **`getAnalyticalIntegral()`**

Integration is requested over all variables in set `allVars`

```
Int_t RooUserFunc::getAnalyticalInteg
                   RooArgSet& allVars, RooArgSet& analVars) const
{
  if (matchArgs(allVars,analVars,x)) return 1 ;
  return 0 ;
}
```

Task of `getAnalyticalIntegral()`:

1) find the *largest subset* that function can integrate analytically
2) Copy largest subset into analVars
3) Return unique identification code for this integral

Writing a function – advertising integrals

Task of **getAnalyticalIntegral()**:

1) find the *largest subset* that function can integrate analytically
2) Copy largest subset into analVars
3) Return unique identification code for this integral

```
Int_t RooUserFunc::getAnalyticalIntegral(
                    RooArgSet& allVars, RooArgSet& analVars) const
{
  if (matchArgs(allVars,analVars,x)) return 1 ;
  return 0 ;
}
```

Utility method **matchArgs()** does all the work for you:

If **allVars** contains the variable held in proxy **x**
variable is copied to **analVars** and **matchArgs()** returns **kTRUE**
If not, it returns **kFALSE**

Writing a function – advertising multiple integrals

```
Int_t RooUserFunc::getAnalyticalIntegral(
                    RooArgSet& allVars, RooArgSet& analVars) const
{
 if (matchArgs(allVars,analVars,x,m)) return 3 ;
 if (matchArgs(allVars,analVars,m)) return 2 ;
 if (matchArgs(allVars,analVars,x)) return 1 ;
  return 0 ;
}
```

If multiple integrals are advertised, test for the largest one first

You may advertise analytical integrals for both *real-valued* and *discrete-valued* integrands

Writing a function – implementing integrals

- Implementing integrals: **`analyticalIntegral()`**
  - One entry point for *all* advertised integrals

Integral identification code assigned by `getAnalyticalIntegral()`

```
Double_t RooGaussian::analyticalIntegral(Int_t code) const
{
  static const Double_t root2 = sqrt(2) ;
  static const Double_t rootPiBy2 = sqrt(atan2(0.0,-1.0)/2.0);

  Double_t xscale = root2*sigma;
  return rootPiBy2*sigma*
         (erf((x.max()-mean)/xscale)-erf((x.min()-mean)/xscale));
}
```

Integration limits for real-valued integrands can be accessed via the `min()` and `max()` method of each proxy

Discrete-valued integrands are always summed over *all* states

Calculating integrals – behind the scenes

- Integrals are calculated by **RooRealIntegral**
  - To create an **RooRealIntegral** for a **RooAbsReal**

```
RooAbsReal* f; // f(x)
RooAbsReal* int_f = f.createIntegral(x) ;

RooAbsReal* g ; // g(x,y)
RooAbsReal* inty_g = g.createIntegral(y) ;
RooAbsReal* intxy_g = g.createIntegral(RooArgSet(x,y)) ;
```

A **RooRealIntegral** is also a **RooAbsReal**

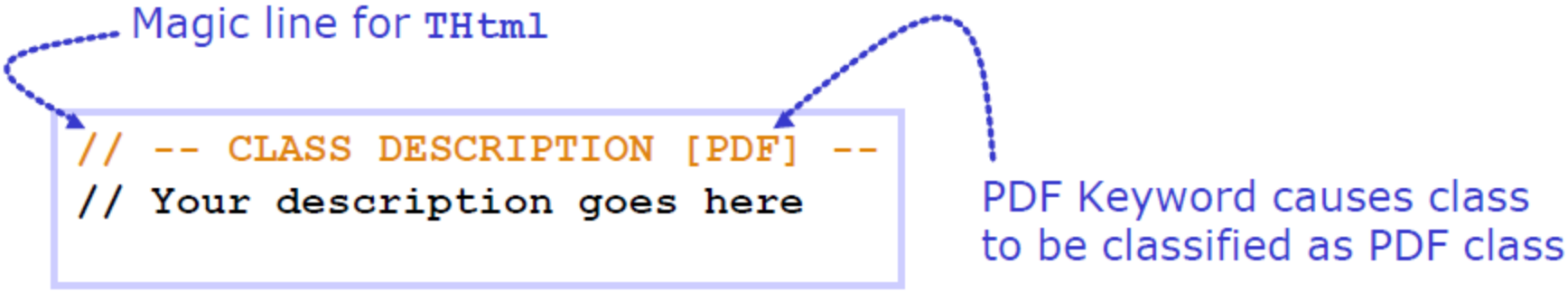
- Tasks of **RooRealIntegral**
  - Structural analysis of composite
  - Negotiate analytical integration with components PDF
  - Provide numerical integration where needed

RooRealIntegral is RooFits most complex class!

- **RooRealIntegral** works universally on simple and composite objects

Class documentation

- General description of the class functionality should be provided at the beginning of your .cc file



- First comment block in each function will be picked up by **THtml** as the description of that member function
  - Put some general, sensible description here

Writing a PDF – Normalization

- Do not ***under any circumstances*** attempt to ***normalize*** your PDF in `evaluate()` via ***explicit*** or ***implicit integration***
- You do not know over what variables to normalize
  - In RooFit, parameter/observable distinction is dynamic, a PDF does not have a unique normalization/return value
- You don't even now know how to integrate yourself!
  - Your PDF may be part of a larger composite structure. Variables may be functions, your internal representation may have a difference number of dimensions that the actual composite object.
  - `RooRealIntegral` takes proper care of all this
- But you can help!
  - Advertise all partial integrals that you can calculate
  - They will be used in the normalization when appropriate
    - Function calling overhead is minimal

Writing a PDF – advertising an internal generator

Task of `getGenerator()`:

1) find the *largest subset* of observables PDF can generate internally
2) Copy largest subset into dirVars
3) Return unique identification code for this integral

```
Int_t RooUserFunc::getGenerator(
  RooArgSet& allVars, RooArgSet& dirVars, Bool_t staticOK) const
{
  if (matchArgs(allVars,dirVars,x)) return 1 ;
  return 0 ;
}
```

Utility method **`matchArgs()`** does all the work for you:

If **`allVars`** contains the variable held in proxy **`x`**
variable is copied to **`dirVars`** and **`matchArgs()`** returns **`kTRUE`**
If not, it returns **`kFALSE`**

Writing a PDF – advertising an internal generator

- For certain internal generator implementations it can be efficient to do a one-time initialization for each set of generated events
  - Example: precalculate fractions for discrete variables
- Caveat: one-time initialization only safe if no observables are generated from a prototype dataset
  - Only advertise such techniques if staticOK flag is true

```
Int_t RooBMixDecay::getGenerator(const RooArgSet& directVars,
             RooArgSet &generateVars, Bool_t staticInitOK) const
{
  if (staticInitOK) {
    if (matchArgs(directVars,generateVars,t,mix,tag)) return 4 ;
    if (matchArgs(directVars,generateVars,t,mix)) return 3 ;
    if (matchArgs(directVars,generateVars,t,tag)) return 2 ;
  }

  if (matchArgs(directVars,generateVars,_t)) return 1 ;
  return 0 ;
}
```

If you advertise multiple configurations, try the most extensive one first

Writing a PDF – implementing an internal generator

- Implementing a generator: **`generateEvent()`**
  - One entry point for *all* advertised event generators

Generator identification code assigned by `getGenerator()`

```
void RooGaussian::generateEvent(Int_t code)
{
  Double_t xgen ;
  while(1) {
    xgen = RooRandom::randomGenerator()->Gaus(mean,sigma);
    if (xgen<x.max() && xgen>x.min()) {
      x = xgen ;
      break;
    }
  }
  return;
}
```

Return generated value by assigning it to the proxy

Writing a PDF – implementing an internal generator

- Static generator initialization: **`initGenerator()`**
  - This function is guaranteed to be call once before each series of **`generateEvent()`** calls with the same configuration

Generator identification code assigned by **`getGenerator()`**

```
void RooBMixDecay::initGenerator(Int_t code)
{
  switch (code) {
  case 2:
    {
      // Calculate the fraction of B0bar events to generate
      Double_t sumInt = RooRealIntegral(…).getVal() ;
      _tagFlav = 1 ; // B0
      Double_t flavInt = RooRealIntegral(…).getVal() ;
      _genFlavFrac = flavInt/sumInt ;
      break ;
    }
  }
}
```

Store your precalculated values in data members

11 Documentation

Documentation, Forum and Release plans

- Some aspects were not covered for which tools exists
  - E.g. constrained fits, fitting efficiencies, limit calculations
- Sources of documentation
  - Home page
    http://roofit.sourceforge.net/
  - Class documentation
    https://root.cern.ch/doc/master/group__Roofit.html
  - Users Manual
    https://root.cern.ch/root-user-guides-and-manuals
  - Tutorial macros ($ROOTSYS/tutorial/roofit)
- Where to ask (technical) questions
  - ROOT users Forum