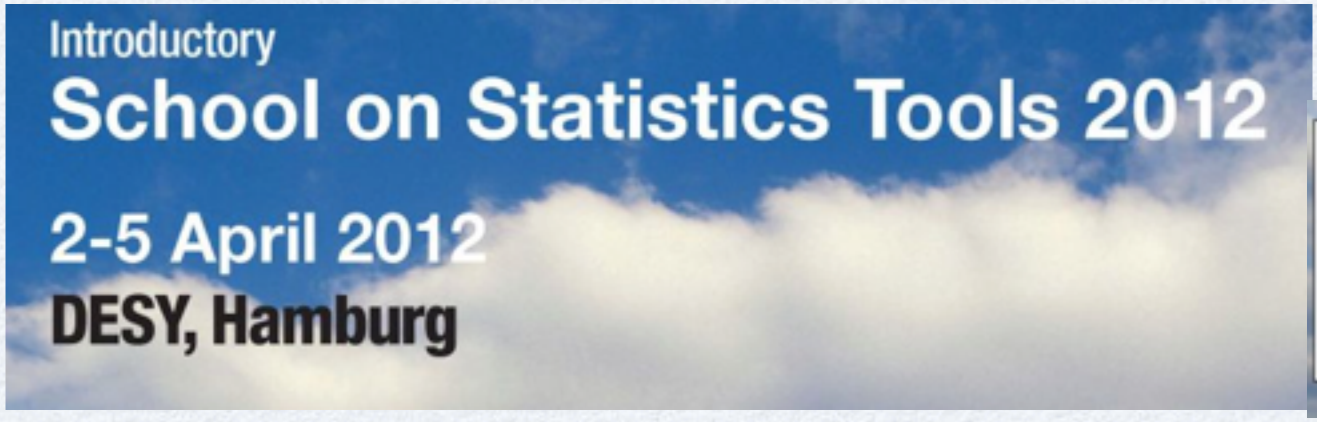
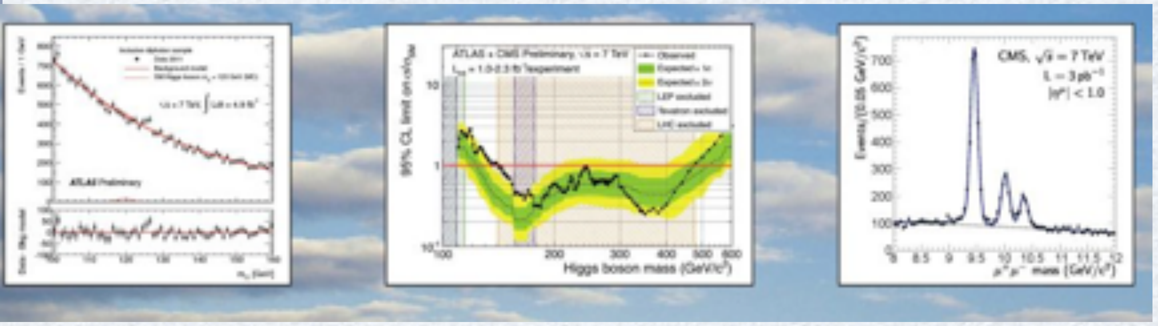
RooFit/RooStats Tutorials

Lorenzo Moneta (CERN)
Sven Kreiss (NYU)


Introductory
School on Statistics Tools 2012
2-5 April 2012
DESY, Hamburg



Outline

- RooFit
  - Introduction and overview of basic functionality
  - Composite models building
  - Advance functionality (e.g. working with likelihood)
- RooFit Exercises

- Introduction to RooStats
- RooStats Exercises

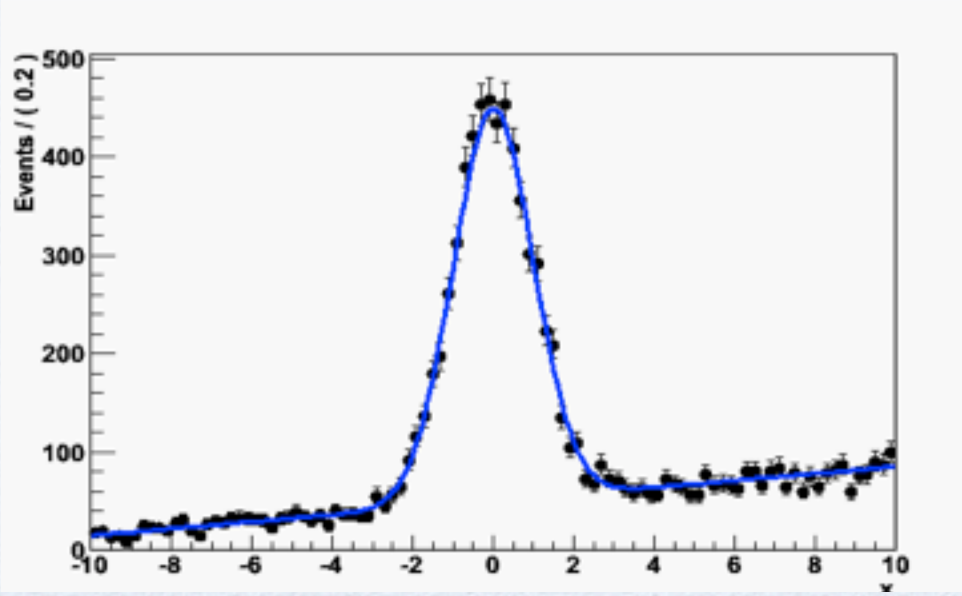
CREDITS:

- RooFit sides and example from material prepared by W. Verkerke (NIKHEF)
- more information and slides from Wouter available at

http://indico.in2p3.fr/getFile.py/access?contribId=15&resId=0&materialId=slides&confId=750

RooFit

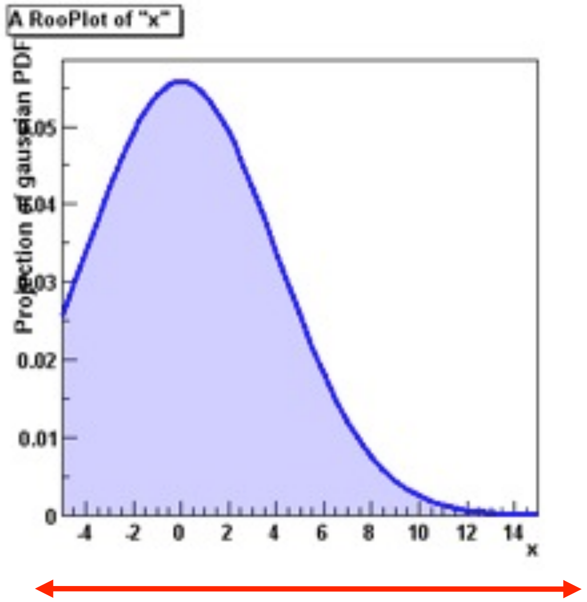
- Toolkit for data modeling
  - developed by *W. Verkerke and D. Kirkby*
- model distribution of observable ***x*** in terms of parameters ***p***
  - probability density function (pdf): $\mathcal{P}$`(x;p)`
- pdf are normalized over allowed range of observables ***x*** with respect to the parameters ***p***



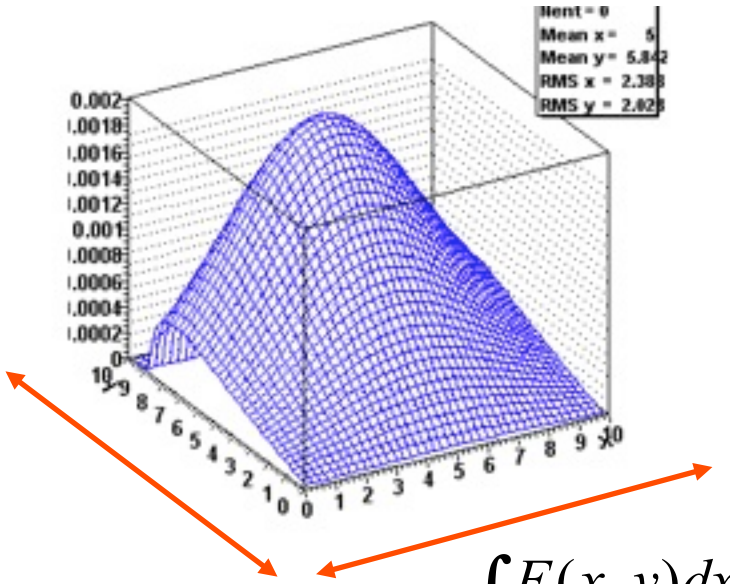
Mathematic – Probability density functions

- 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

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



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



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

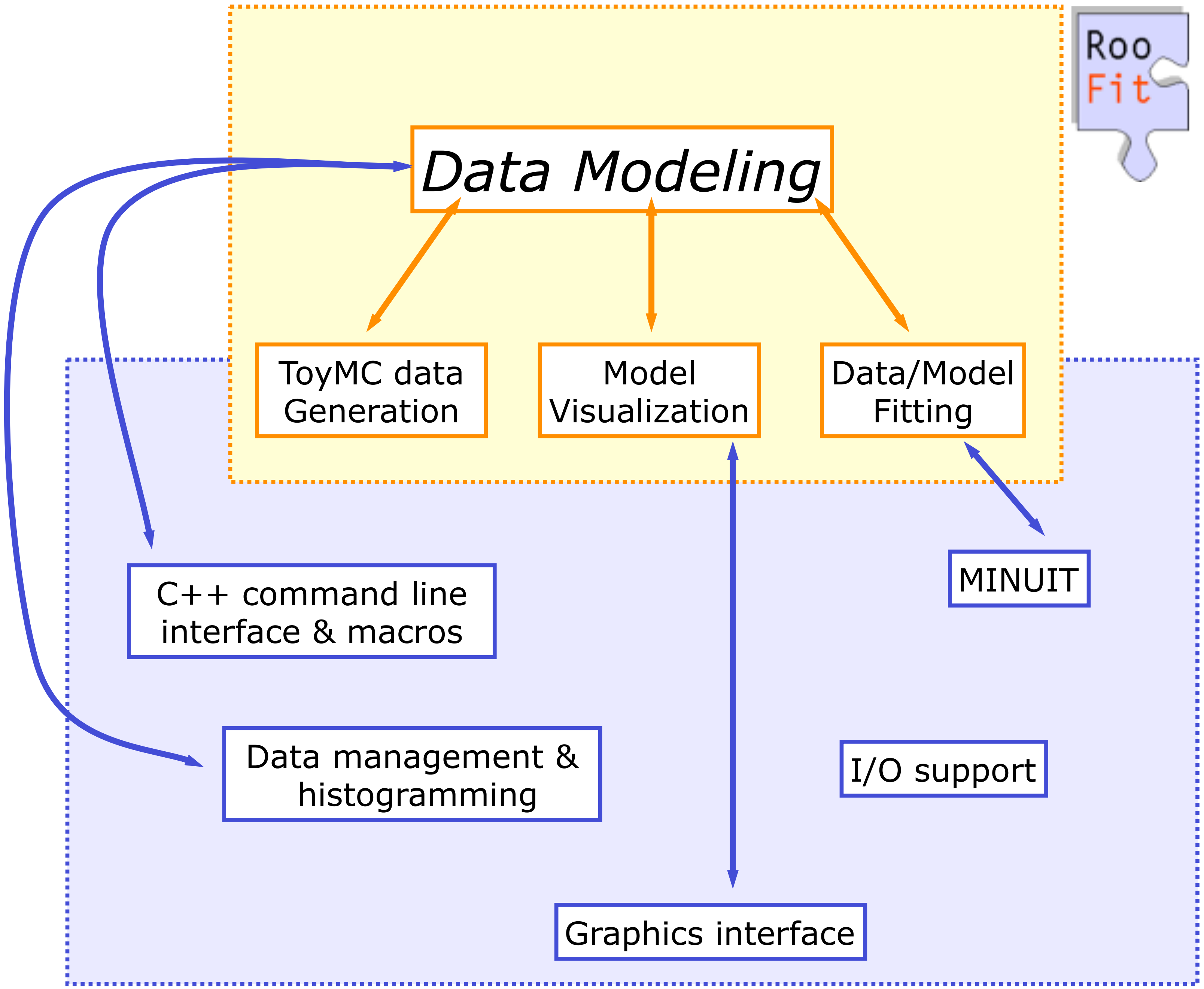
- Note distinction in role between *parameters* (p) and *observables* (x)
  - Observables are measured quantities
  - Parameters are degrees of freedom in your model

RooFit

- RooFit provides functionality for building the pdf's
  - complex model building from standard components
  - composition with addition product and convolution
- All models provide the functionality for
  - maximum likelihood fitting
  - toy MC generator
  - visualization
- Extension of ROOT functionality

Introduction – Relation to ROOT

Extension to ROOT – (Almost) no overlap with existing functionality



Data Modeling

ToyMC data Generation

Model Visualization

Data/Model Fitting

C++ command line interface & macros

MINUIT

Data management & histogramming

I/O support

Graphics interface

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	$\int_{x_{\mathrm{min}}}^{x_{\mathrm{max}}} f(x)dx$	**`RooRealIntegral`**
list of space points		**`RooAbsData`**

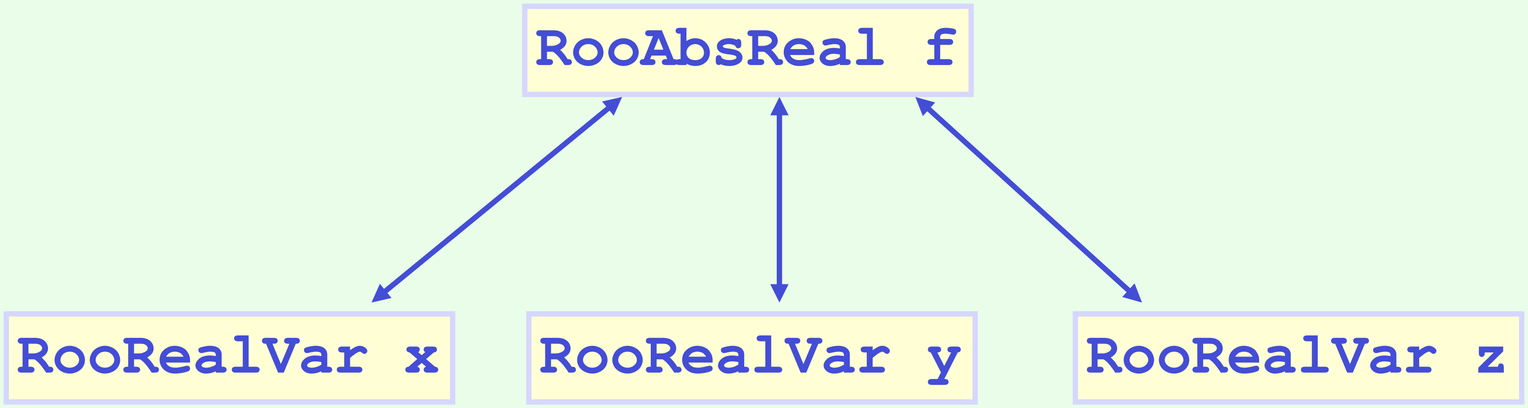
RooFit core design philosophy

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

Math

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

RooFit diagram

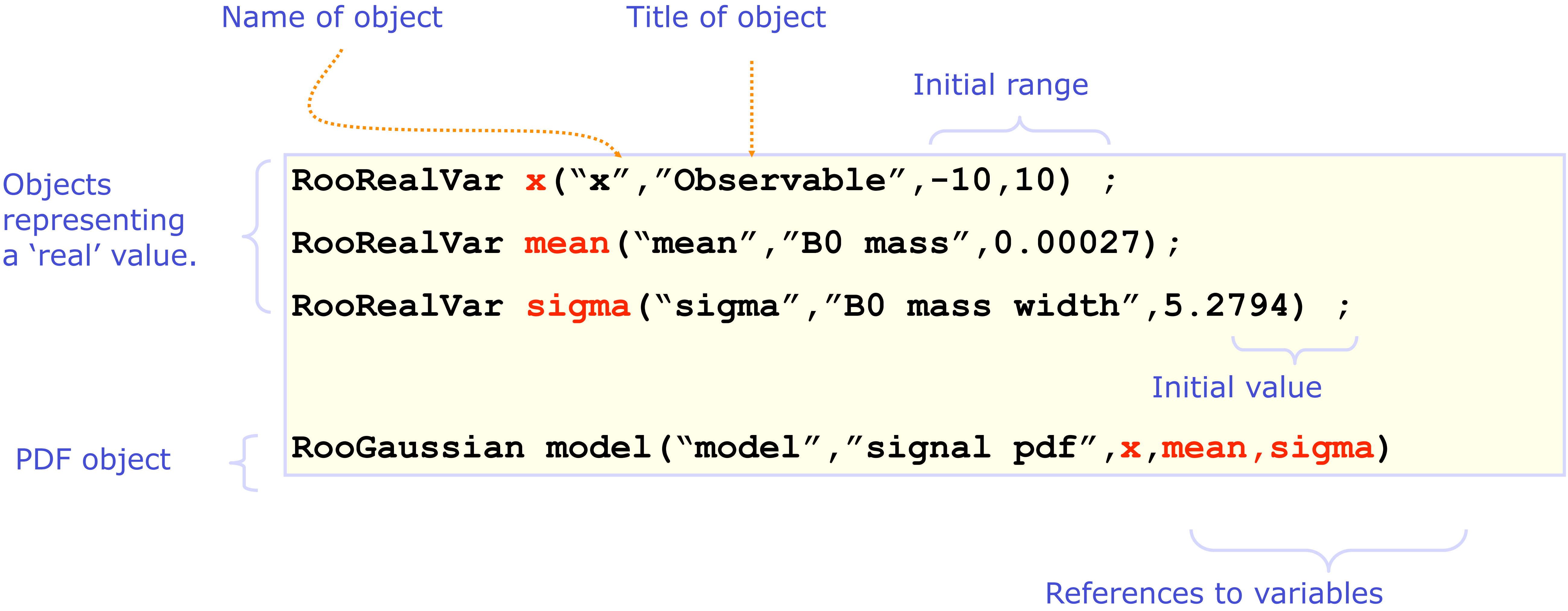


RooFit code

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

The simplest possible example

- We make a Gaussian p.d.f. with three variables: mass, mean and sigma



Name of object

Title of object

Initial range

Objects representing a ‘real’ value.

```
RooRealVar x(“x”,”Observable”,-10,10) ;
RooRealVar mean(“mean”,”B0 mass”,0.00027);
RooRealVar sigma(“sigma”,”B0 mass width”,5.2794) ;
```

Initial value

PDF object

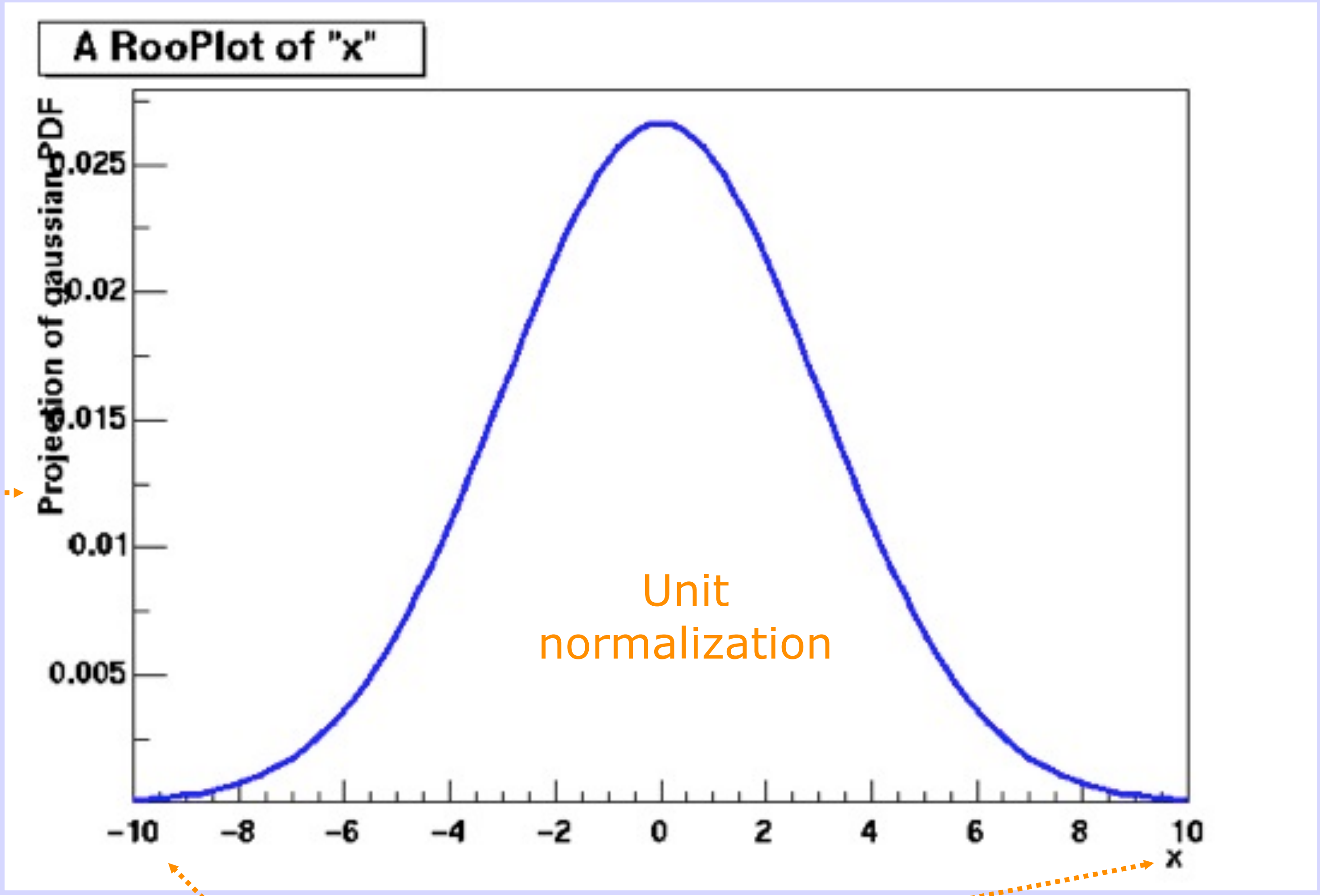
```
RooGaussian model(“model”,”signal pdf”,x,mean,sigma)
```

References to variables

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

Setup gaussian PDF and plot

```
// Create an empty plot frame
RooPlot* xframe = x.frame() ;

// Plot model on frame
model.plotOn(xframe) ;

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



Axis label from **gauss** title

Unit normalization

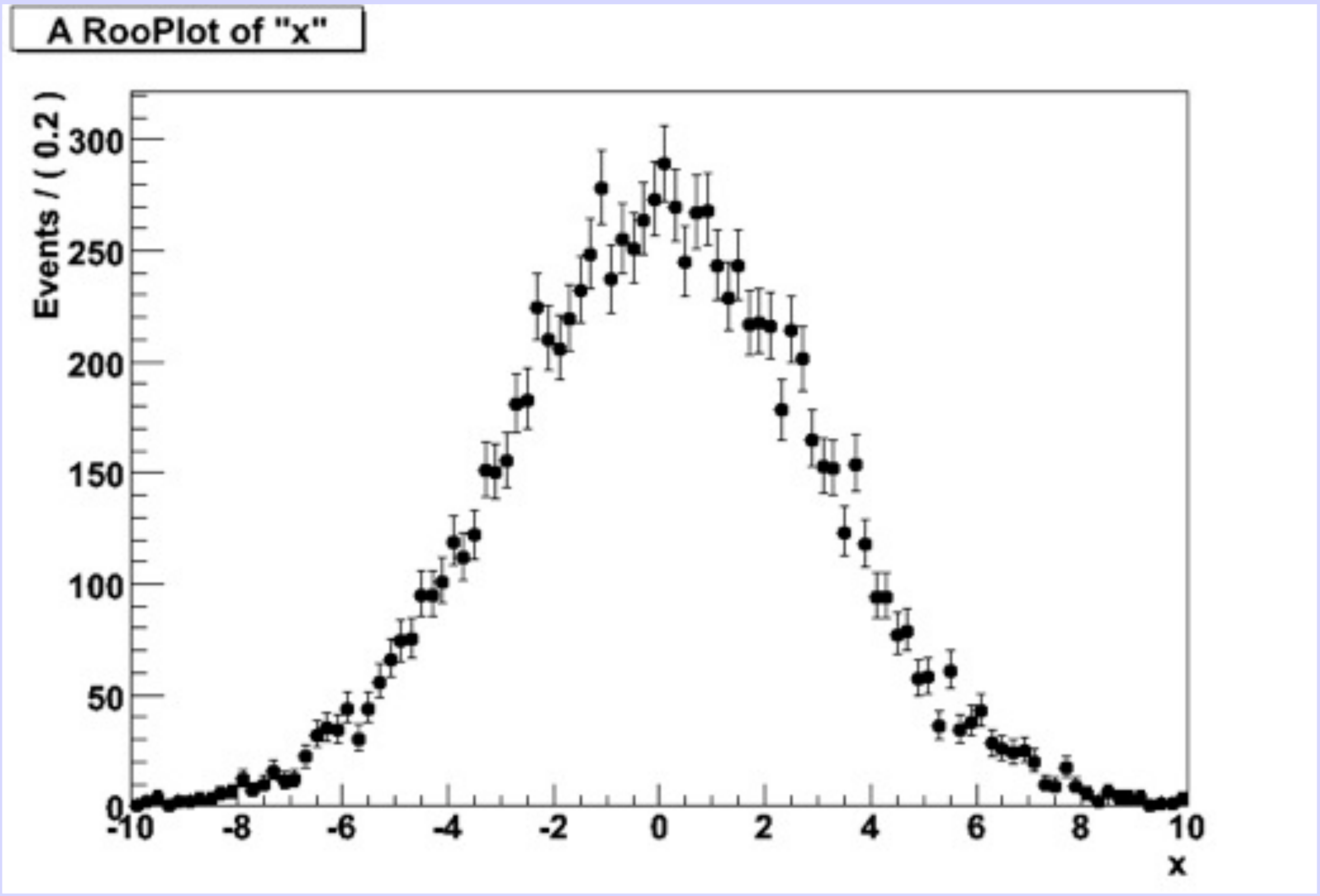
A **RooPlot** is an empty frame capable of holding anything plotted versus it variable

Plot range taken from limits of **x**

Basics – Generating toy MC events

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

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

// Generate an binned toy MC set
RooDataHist* data = gauss.generateBinned(x,10000) ;

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



Can generate both binned and unbinned datasets

Basics – Importing data

- Unbinned data can also be imported from ROOT **`TTrees`**

```
// Import unbinned data
RooDataSet data(“data”,”data”,x,Import(*myTree)) ;
```

  - Imports **`TTree`** branch named “x”.
  - Can be of type **`Double_t`**, **`Float_t`**, **`Int_t`** or **`UInt_t`**. All data is converted to Double_t internally
  - Specify a **`RooArgSet`** of multiple observables to import multiple observables

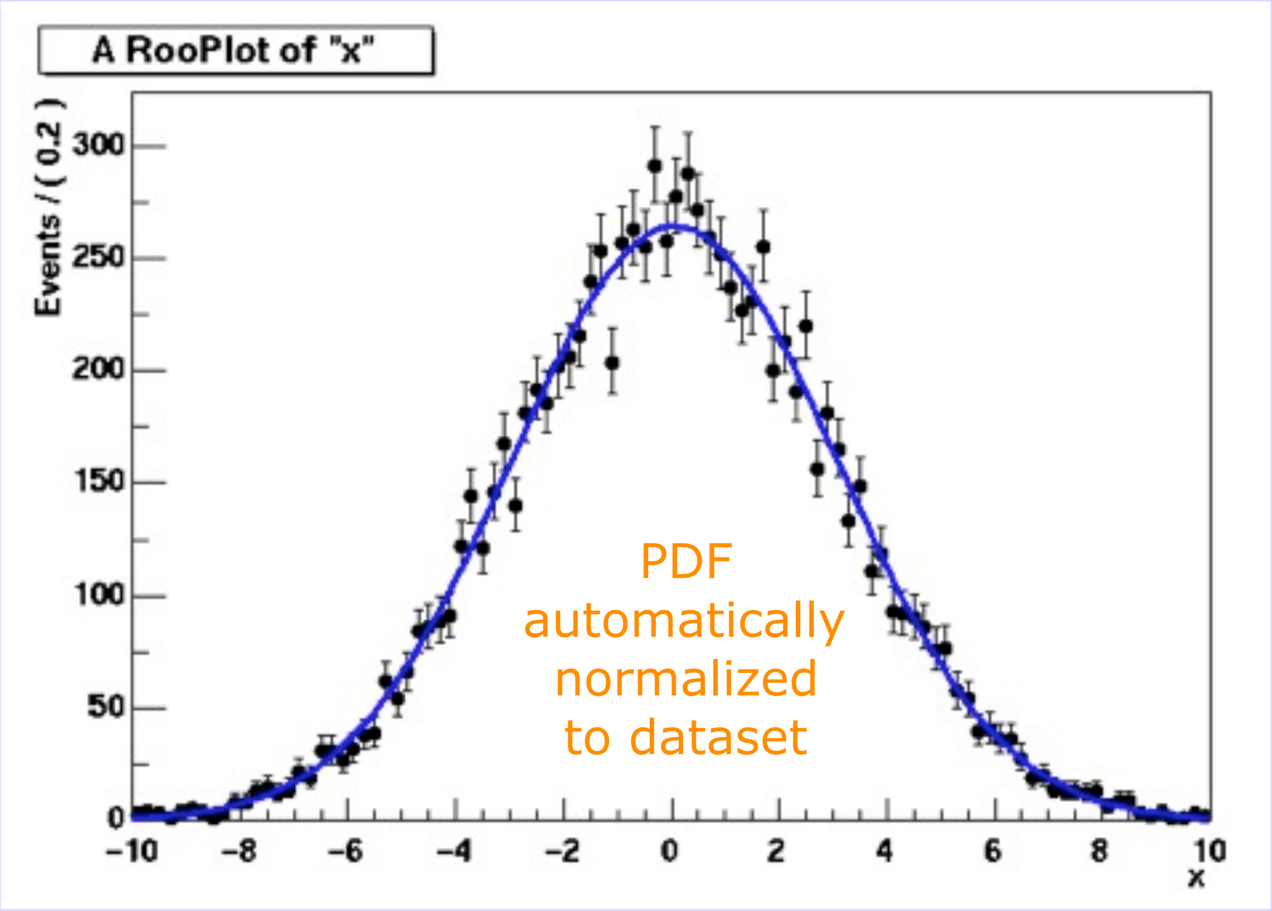
- Binned data can be imported from ROOT **`THx`** histograms

```
// Import unbinned data
RooDataHist data(“data”,”data”,x,Import(*myTH1)) ;
```

  - Imports values, binning definition *and* SumW2 errors (if defined)
  - Specify a **`RooArgList`** of observables when importing a TH2/3.

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

```
// 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* xframe = x.frame() ;
data->plotOn(xframe) ;
gauss.plotOn(xframe) ;
```



PDF automatically normalized to dataset

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

- Can also choose to save full detail of fit

```
RooFitResult* r = gauss.fitTo(*data,Save()) ;

r->Print() ;
  RooFitResult: minimized FCN value: 25055.6,
                estimated distance to minimum: 7.27598e-08
                coviarance matrix quality:
                Full, accurate covariance matrix

    Floating Parameter    FinalValue +/-  Error
  --------------------  --------------------------
                  mean    1.7233e-02 +/-  3.00e-02
                 sigma    2.9809e+00 +/-  2.17e-02


r->correlationMatrix().Print() ;

2x2 matrix is as follows

     |      0    |      1    |
-------------------------------
   0 |          1   0.0005869
   1 |  0.0005869           1
```

RooFit Factory

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

Provides a factory to auto-generates objects from a math-like language

```
RooWorkspace w;
w.factory(“Gaussian::g(x[2,-10,10],m[0],s[3])”)
```

We will work in the example and exercises using the workspace factory to build models

RooWorkspace

- Workspace class in RooFit (**`RooWorkSpace`**) with:
  - full model configuration
    - PDF and parameter/observables descriptions
    - uncertainty/shape of nuisance parameters
  - (multiple) data sets
- Maintain a complete description of all the model
  - possibility to save entire model in a ROOT file
- Combination of results joining workspaces in a single one
- All information is available for further analysis
  - common format for combining and sharing physics results

```
RooWorkspace workspace(“Example_workspace”);
workspace.import(*data);
workspace.import(*pdf);
workspace.defineSet(“obs”,”x”);
workspace.defineSet(“poi”,”mu”);
workspace.importClassCode();
workspace.writeToFile(“myWorkspace”)
```

Using the workspace

- Workspace
  - A generic container class for all RooFit objects of your project
  - Helps to organize analysis projects
- Creating a workspace

```
RooWorkspace w(“w”) ;
```

- Putting variables and function into a workspace
  - When importing a function or pdf, all its components (variables) are automatically imported too

```
RooRealVar x(“x”,”x”,-10,10) ;
RooRealVar mean(“mean”,”mean”,5) ;
RooRealVar sigma(“sigma”,”sigma”,3)  ;
RooGaussian f(“f”,”f”,x,mean,sigma) ;

// imports f,x,mean and sigma
w.import(f) ;
```

Using the workspace

- Looking into a workspace

```
w.Print() ;

variables
---------
(mean,sigma,x)

p.d.f.s
-------
RooGaussian::f[ x=x mean=mean sigma=sigma ] = 0.249352
```

- Getting variables and functions out of a workspace

```
// Variety of accessors available
RooPlot* frame = w.var("x")->frame() ;
w.pdf("f")->plotOn(frame) ;
```

Using the workspace

- Alternative access to contents through namespace
  - Uses CINT extension of C++, works in interpreted code only
  - (Alternatively construct workspace with kTRUE as 2nd arg)

```
// Variety of accessors available
w.exportToCint() ;
RooPlot* frame = w::x.frame() ;
w::f.plotOn(frame) ;
```

- Writing workspace and contents to file

```
w.writeToFile("wspace.root") ;
```

Using the workspace

- Organizing your code –
  Separate construction and use of models

```
void driver() {
  RooWorkspace w(“w”) ;
  makeModel(w) ;
  useModel(w) ;
}


void makeModel(RooWorkspace& w) {
  // Construct model here
}


void useModel(RooWorkspace& w) {
  // Make fit, plots etc here
}
```

Factory and Workspace

- *One C++ object per math symbol* provides ultimate level of control over each objects functionality, but results in lengthy user code for even simple macros
- Solution: add factory that auto-generates objects from a math-like language. Accessed through factory() method of workspace
- Example: reduce construction of Gaussian pdf and its parameters from 4 to 1 line of code

```
w.factory(“Gaussian::f(x[-10,10],mean[5],sigma[3])”) ;
```

```
RooRealVar x(“x”,”x”,-10,10) ;
RooRealVar mean(“mean”,”mean”,5) ;
RooRealVar sigma(“sigma”,”sigma”,3)  ;
RooGaussian f(“f”,”f”,x,mean,sigma) ;
```

Factory syntax

- Rule #1 – Create a variable

```
x[-10,10]    // Create variable with given range
x[5,-10,10]  // Create variable with initial value and range
x[5]         // Create initially constant variable
```

- Rule #2 – Create a function or pdf object

```
ClassName::Objectname(arg1,[arg2],...)
```

  - Leading ‘Roo’ in class name can be omitted
  - Arguments are names of objects that already exist in the workspace
  - Named objects must be of correct type, if not factory issues error
  - Set and List arguments can be constructed with brackets {}

```
Gaussian::g(x,mean,sigma)
          → RooGaussian(“g”,”g”,x,mean,sigma)

Polynomial::p(x,{a0,a1})
                → RooPolynomial(“p”,”p”,x”,RooArgList(a0,a1));
```

Factory syntax

- Rule #3 – Each creation expression returns the name of the object created
  - Allows to create input arguments to functions ‘in place’ rather than in advance

```
Gaussian::g(x[-10,10],mean[-10,10],sigma[3])
    →   x[-10,10]
        mean[-10,10]
        sigma[3]
        Gaussian::g(x,mean,sigma)
```

- Miscellaneous points
  - You can always use numeric literals where values or functions are expected

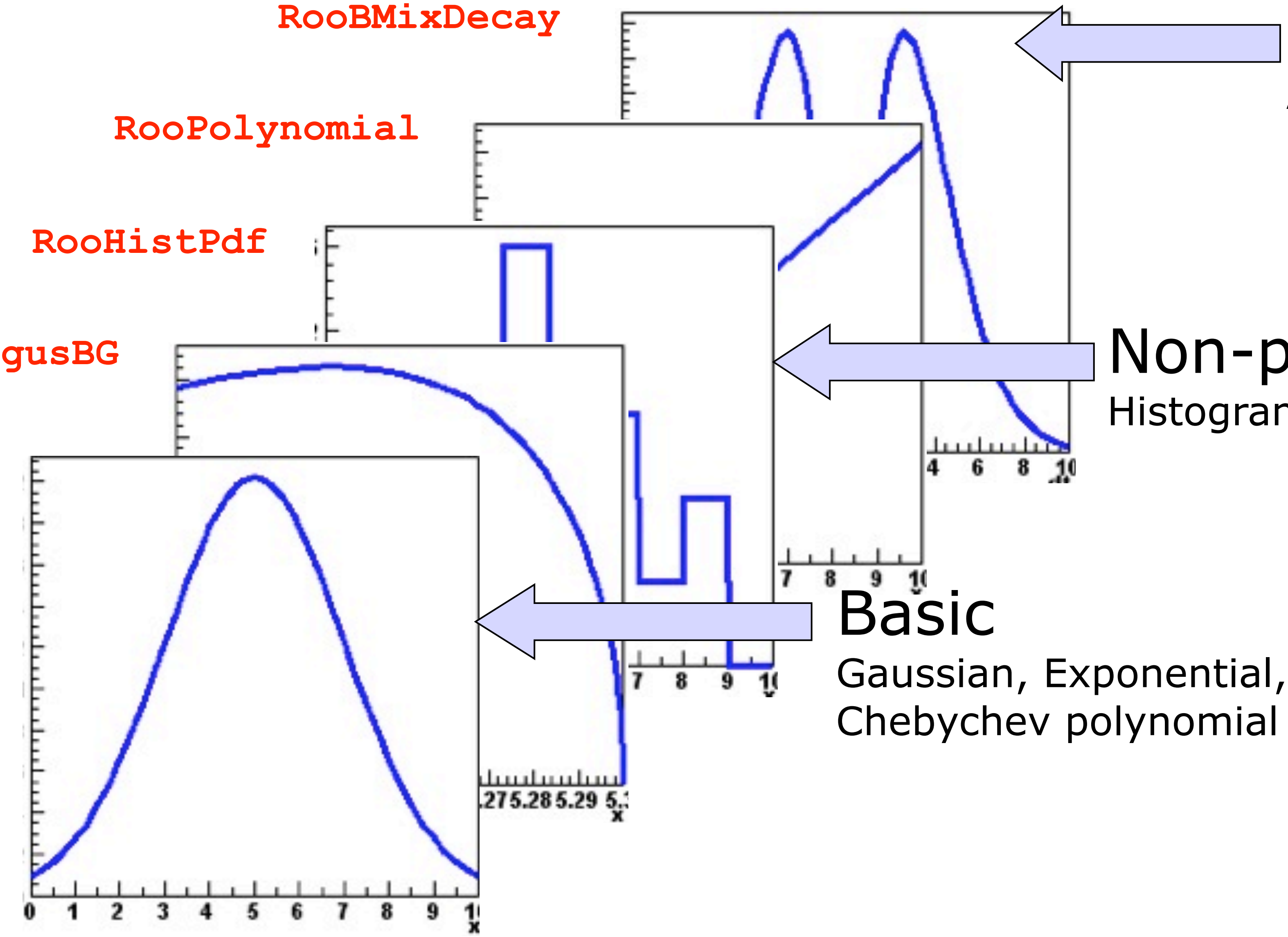
```
Gaussian::g(x[-10,10],0,3)
```

  - It is not required to give component objects a name, e.g.

```
SUM::model(0.5*Gaussian(x[-10,10],0,3),Uniform(x)) ;
```

Model building – (Re)using standard components

- RooFit provides a collection of compiled standard PDF classes



RooBMixDecay

RooPolynomial

RooHistPdf

RooArgusBG

RooGaussian

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

Model building – (Re)using standard components

- List of most frequently used pdfs and their factory spec

Gaussian	`Gaussian::g(x,mean,sigma)`
Breit-Wigner	`BreitWigner::bw(x,mean,gamma)`
Landau	`Landau::l(x,mean,sigma)`
Exponential	`Exponential::e(x,alpha)`
Polynomial	`Polynomial::p(x,{a0,a1,a2})`
Chebychev	`Chebychev::p(x,{a0,a1,a2})`
Kernel Estimation	`KeysPdf::k(x,dataSet)`
Poisson	`Poisson::p(x,mu)`
Voigtian (=BW⊗G)	`Voigtian::v(x,mean,gamma,sigma)`

Model building – Making your own

- Interpreted expressions

```
w.factory(“EXPR::mypdf(‘sqrt(a*x)+b’,x,a,b)”) ;
```

- Customized class, compiled and linked on the fly

```
w.factory(“CEXPR::mypdf(‘sqrt(a*x)+b’,x,a,b)”) ;
```

- Custom class written by you
  - Offer option of providing analytical integrals, custom handling of toy MC generation (details in RooFit Manual)
- Compiled classes are faster in use, but require O(1-2) seconds startup overhead
  - Best choice depends on use context

Model building – Adjusting parameterization

- RooFit pdf classes do not require their parameter arguments to be variables, one can plug in functions as well
- Simplest tool perform reparameterization is interpreted formula expression

```
w.factory(“expr::w(‘(1-D)/2’,D[0,1])”) ;
```

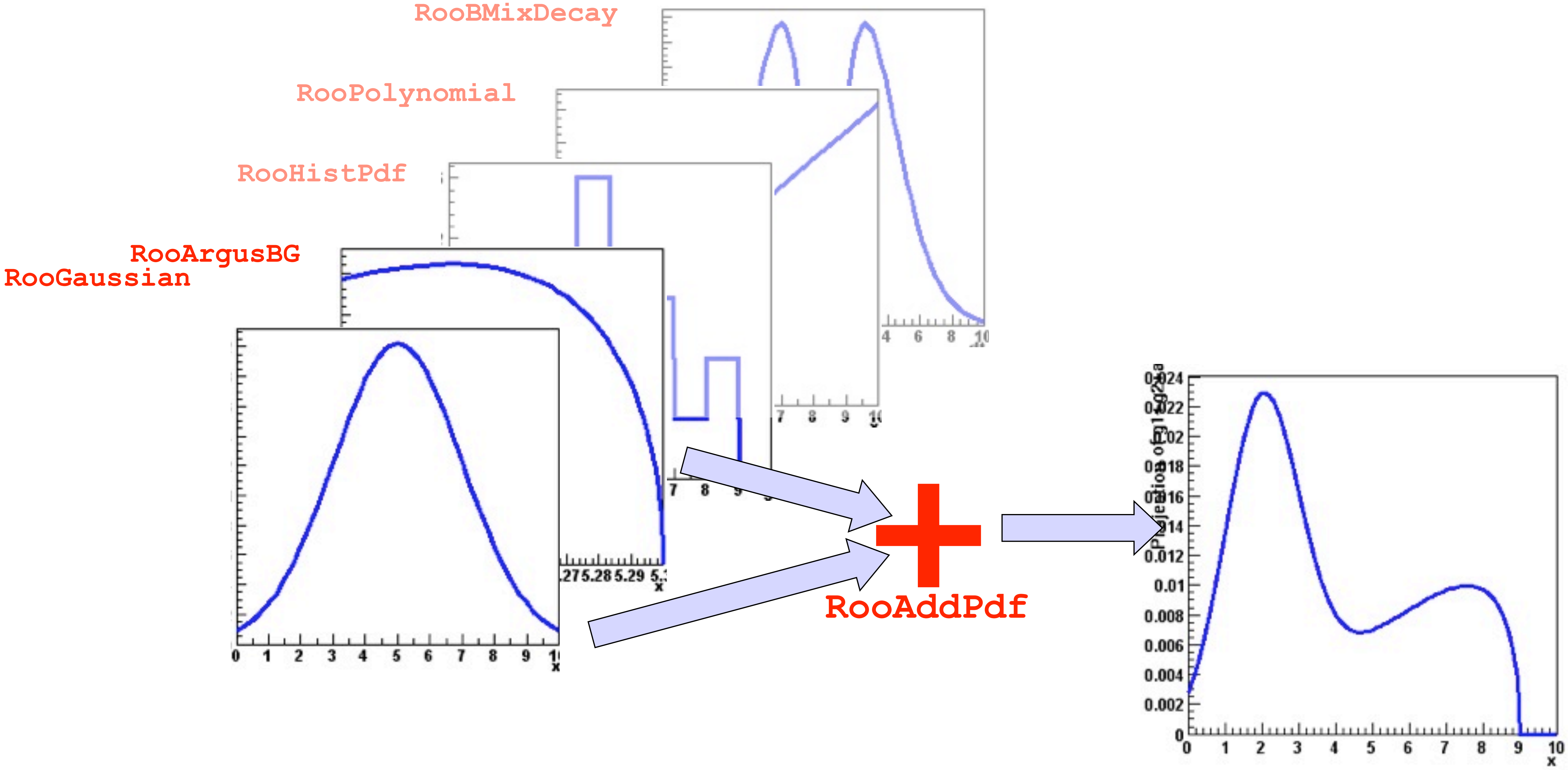
  - Note lower case: **expr** builds function, **EXPR** builds pdf

- Example: Reparameterize pdf that expects mistag rate in terms of dilution

```
w.factory(“BMixDecay::bmix(t,mixState,tagFlav,
                             tau,expr(‘(1-D)/2’,D[0,1]),dw,....”) ;
```

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`**



`RooBMixDecay`

`RooPolynomial`

`RooHistPdf`

`RooArgusBG`

`RooGaussian`

`RooAddPdf`

Adding p.d.f.s – Factory syntax

- Additions created through a SUM expression

```
SUM::name(frac1*PDF1,PDFN)
```

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

```
SUM::name(frac1*PDF1,frac2*PDF2,...,PDFN)
```

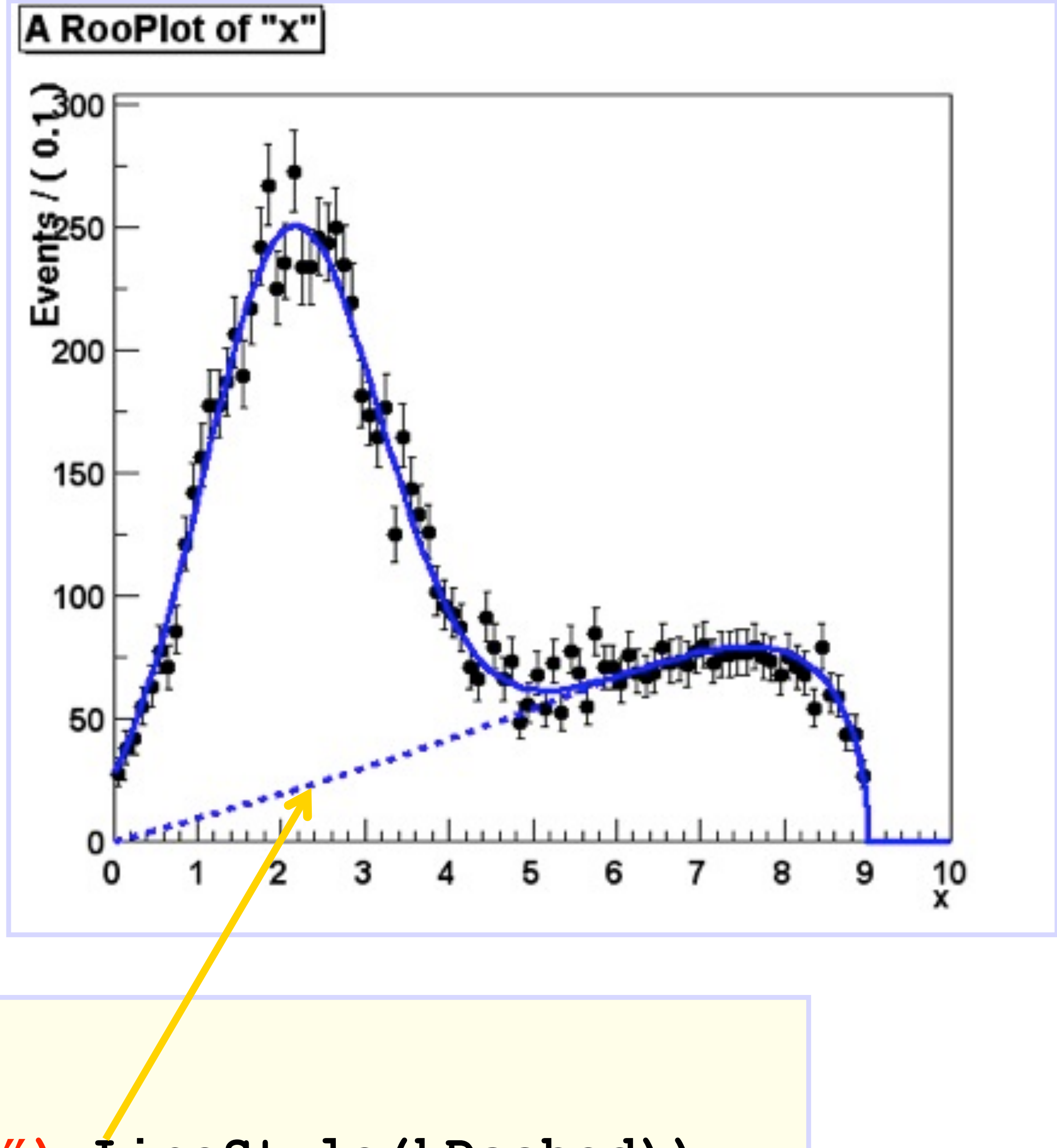
  - Note that last PDF does not have an associated fraction

- Complete example

```
w.factory(“Gaussian::gauss1(x[0,10],mean1[2],sigma[1]”) ;
w.factory(“Gaussian::gauss2(x,mean2[3],sigma)”) ;
w.factory(“ArgusBG::argus(x,k[-1],9.0)”) ;

w.factory(“SUM::sum(g1frac[0.5]*gauss1, g2frac[0.1]*gauss2, argus)”)
```

Component plotting - Introduction

- Plotting, toy event generation and fitting works identically for composite p.d.f.s
  - Several optimizations applied behind the scenes that are specific to composite models (e.g. delegate event generation to components)
- Extra plotting functionality specific to composite pdfs
  - Component plotting



```
// Plot only argus components
w::sum.plotOn(frame,Components(“argus”),LineStyle(kDashed)) ;

// Wildcards allowed
w::sum.plotOn(frame,Components(“gauss*”),LineStyle(kDashed)) ;
```

Extended ML fits

- In an extended ML fit, an extra term is added to the likelihood

  $L(x \mid p) \rightarrow L(x|p)\mathrm{Poisson}(N_{obs},N_{exp})$

- This is most useful in combination with a composite pdf

shape *normalization*

$$F(x) = f \times S(x) + (1-f)B(x) \quad ; \quad N_{\exp} = N$$

$f, N \Rightarrow N_S, N_B$

$$F(x) = \frac{N_S}{N_S+N_B} \times S(x) + \frac{N_B}{N_S+N_B} B(x) \quad ; \quad N_{\exp} = N_S + N_B$$

Write like this,
extended term automatically included in –log(L)

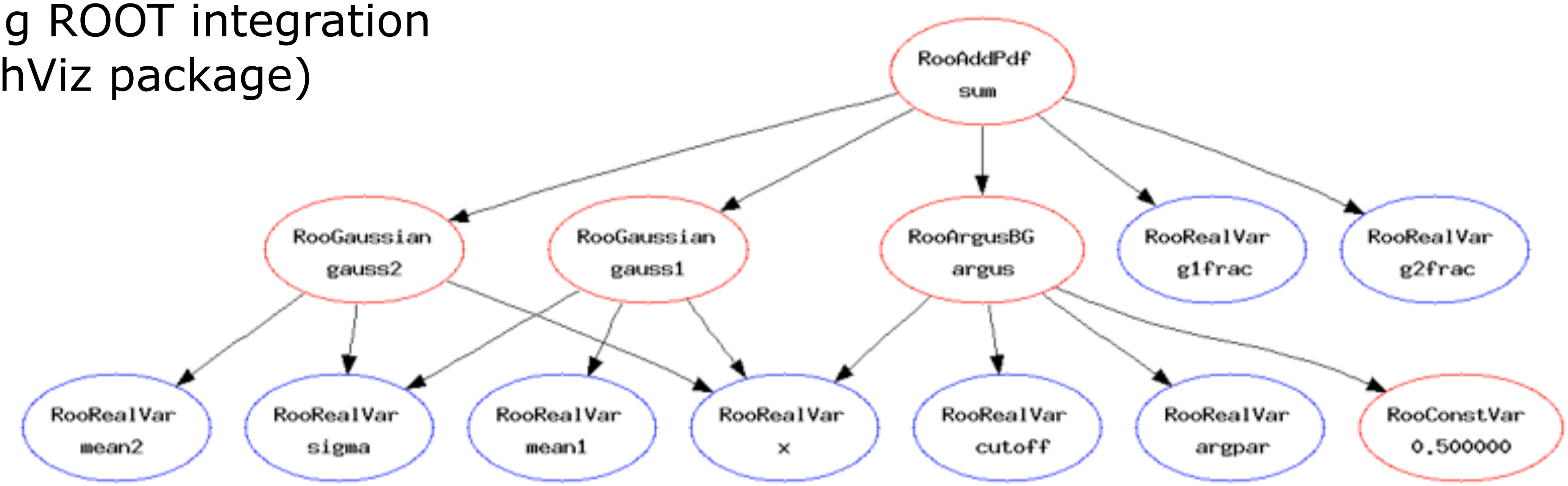
```
SUM::name(Nsig*S,Nbkg*B)
```

Operations on specific to composite pdfs

- Tree printing mode of workspace reveals component structure – **`w.Print("t")`**

```
RooAddPdf::sum[ g1frac * g1 + g2frac * g2 + [%] * argus ] = 0.0687785
    RooGaussian::g1[ x=x mean=mean1 sigma=sigma ] = 0.135335
    RooGaussian::g2[ x=x mean=mean2 sigma=sigma ] = 0.011109
    RooArgusBG::argus[ m=x m0=k c=9 p=0.5 ] = 0
```

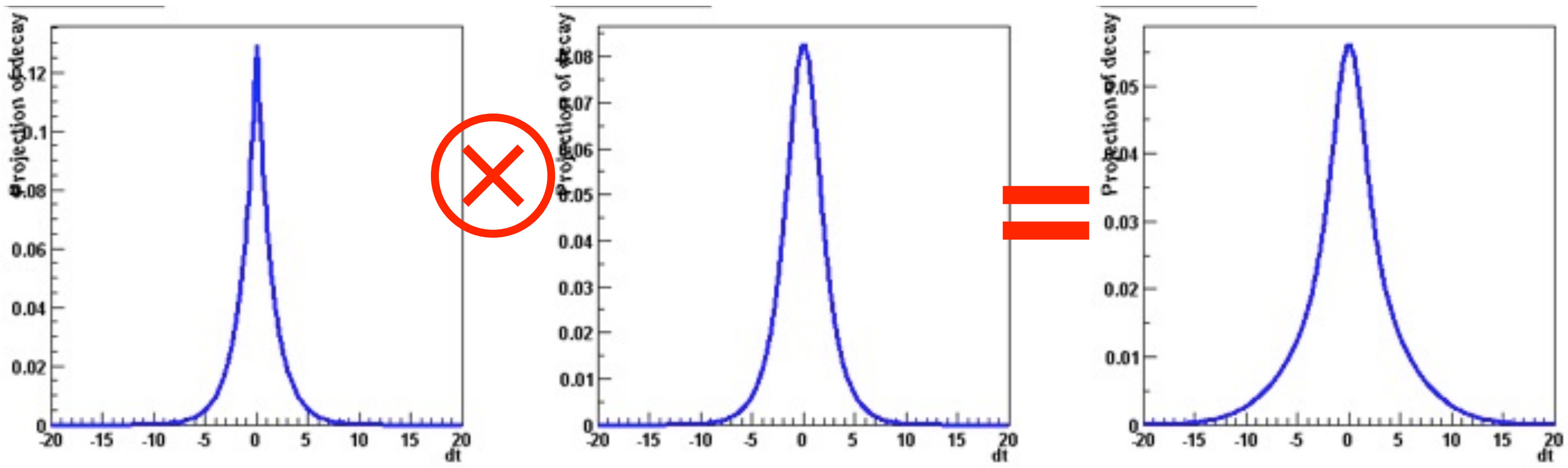
  - Can also make input files for GraphViz visualization (**`w::sum.graphVizTree("myfile.dot")`**)
  - Graph output on ROOT Canvas in near future (pending ROOT integration of GraphViz package)



Convolution

- Model representing a convolution of a theory model and a resolution model often useful

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



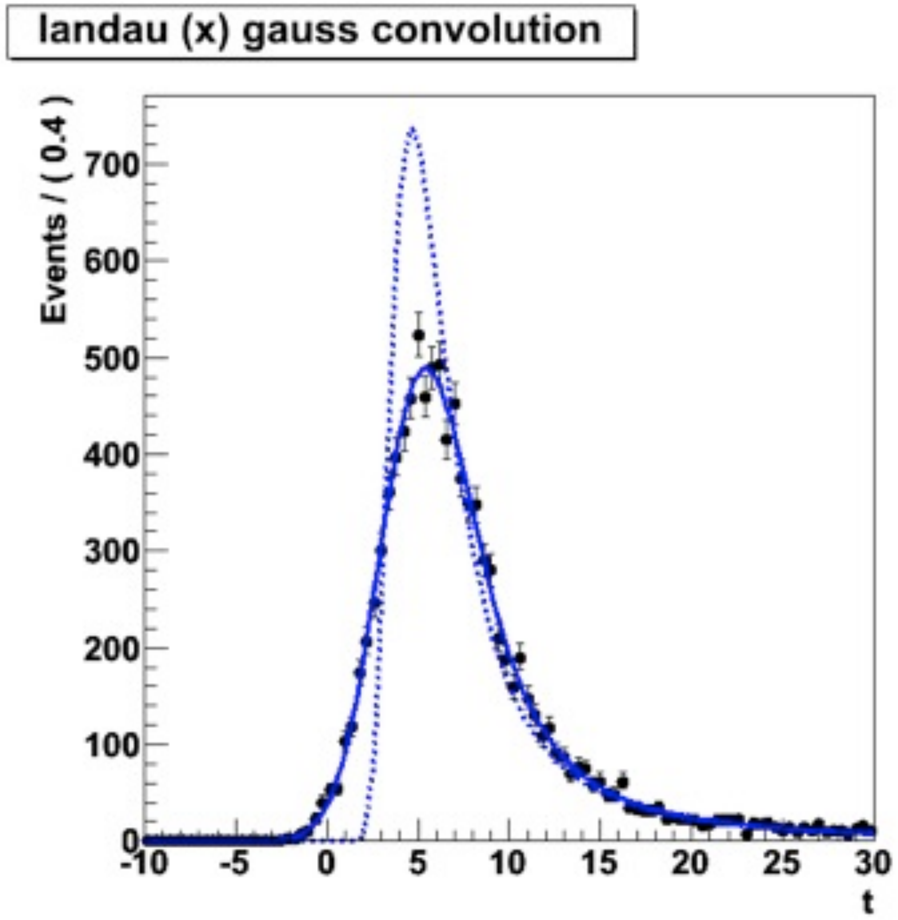
- But numeric calculation of convolution integral can be challenging. No one-size-fits-all solution, but 3 options available
  - Analytical convolution (BW⊗Gauss, various B physics decays)
  - Brute-force numeric calculation (slow)
  - FFT numeric convolution (fast, but some side effects)

Convolution

- Example

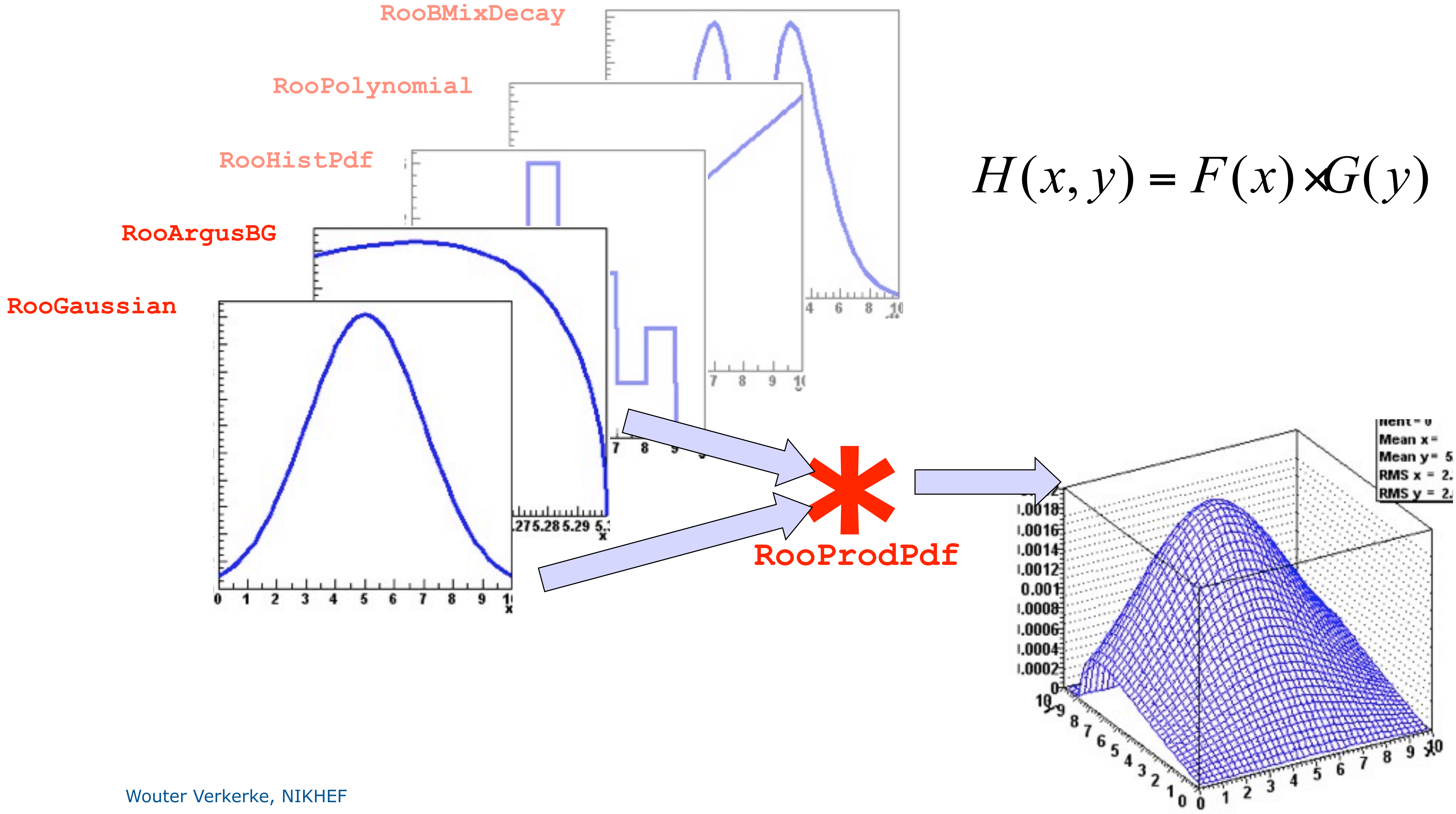
```
w.factory(“Landau::L(x[-10,30],5,1)”) :
w.factory(“Gaussian::G(x,0,2)”) ;

w::x.setBins(“cache”,10000) ; // FFT sampling density
w.factory(“FCONV::LGf(x,L,G)”) ; // FFT convolution

w.factory(“NCONV::LGb(x,L,G)”) ; // Numeric convolution
```

- FFT usually best
  - Fast: unbinned ML fit to 10K events take ~5 seconds
  - NB: Requires installation of FFTW package (free, but not default)
  - Beware of cyclical effects (some tools available to mitigate)



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

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


RooBMixDecay

RooPolynomial

RooHistPdf

RooArgusBG

RooGaussian

RooProdPdf

Uncorrelated products – Mathematics and constructors

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

2D **nD**

$$H(x,y) = F(x)\times G(y) \qquad H(x^{\{i\}}) = \prod_i F^{\{i\}}(x^{\{i\}})$$

  - No explicit normalization required → If input p.d.f.s are unit normalized, product is also unit normalized
  - (Partial) integration and toy MC generation automatically uses factorizing properties of product, e.g. $\int H(x,y)dx \equiv G(y)$ is deduced from structure.

- Corresponding factory operator is PROD

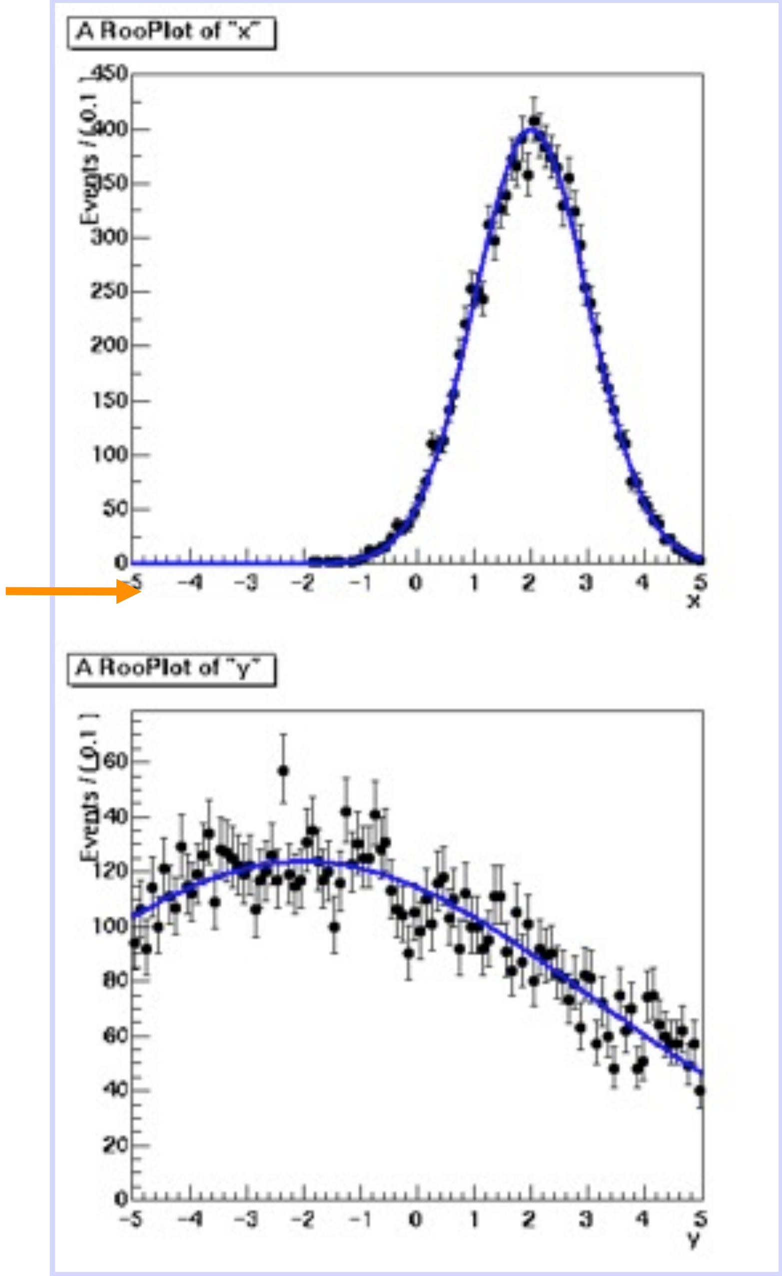
```
w.factory(“Gaussian::gx(x[-5,5],mx[2],sx[1])”) ;
w.factory(“Gaussian::gy(y[-5,5],my[-2],sy[3])”) ;

w.factory(“PROD::gxy(gx,gy)”) ;
```

Plotting multi-dimensional models

- N-D models usually projected on 1-D for visualization
  - Happens automatically.
    RooPlots tracks observables of plotted data,
    subsequent models automatically integrated

```
RooDataSet* dxy =
w::gxy.generate(RooArgSet(w::x,w::y,10000));

RooPlot* frame = w::x.frame() ;
dxy->plotOn(frame) ;
w::gxy.plotOn(frame) ;
```

$$P_{gxy}(x) = \int gxy(x,y)dy$$



  - Projection integrals analytically reduced
    whenever possible
    (e.g. in case of factorizing pdf)

- To make 2,3D histogram of pdf

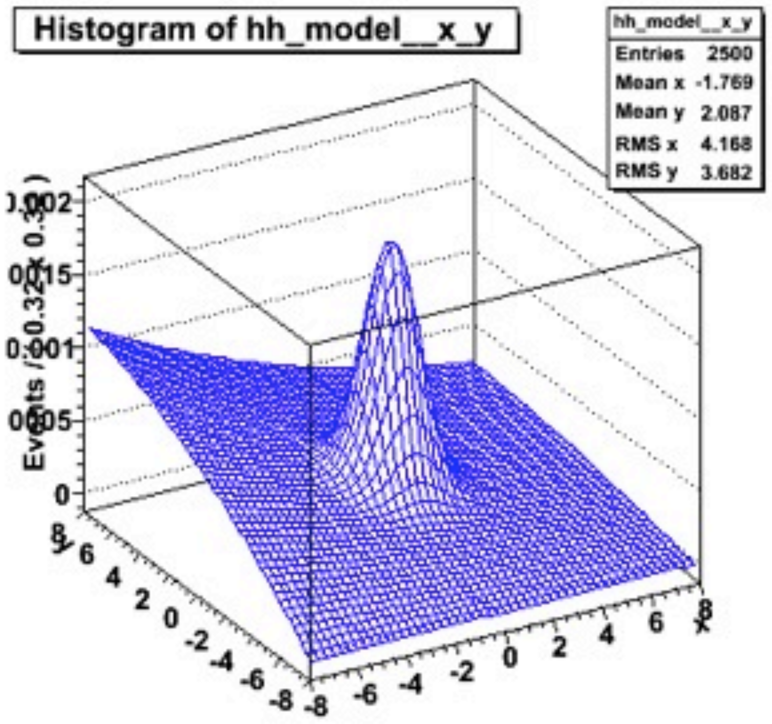
```
TH2* hh = w::gxy.createHistogram(“x,y”,50,50);
```

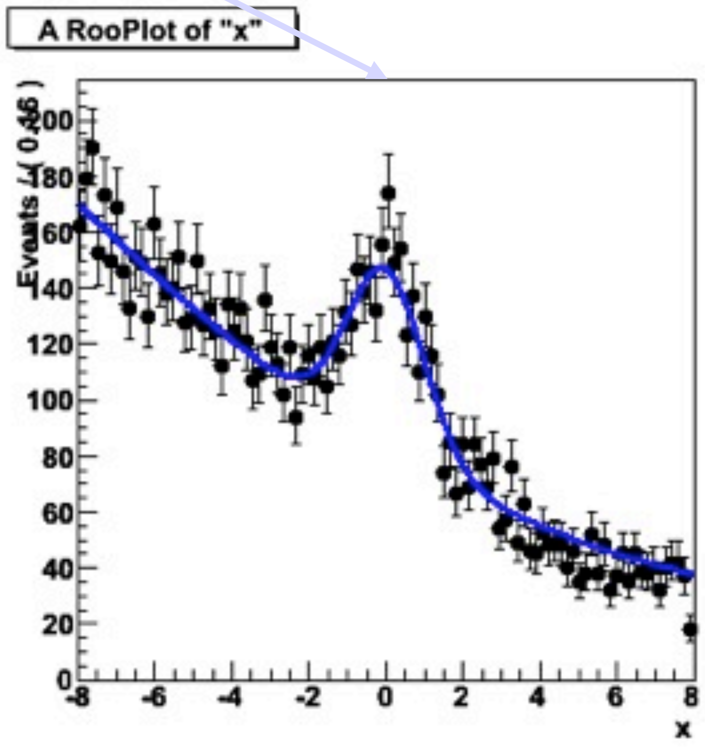
Can also project slices of a multi-dimensional pdf

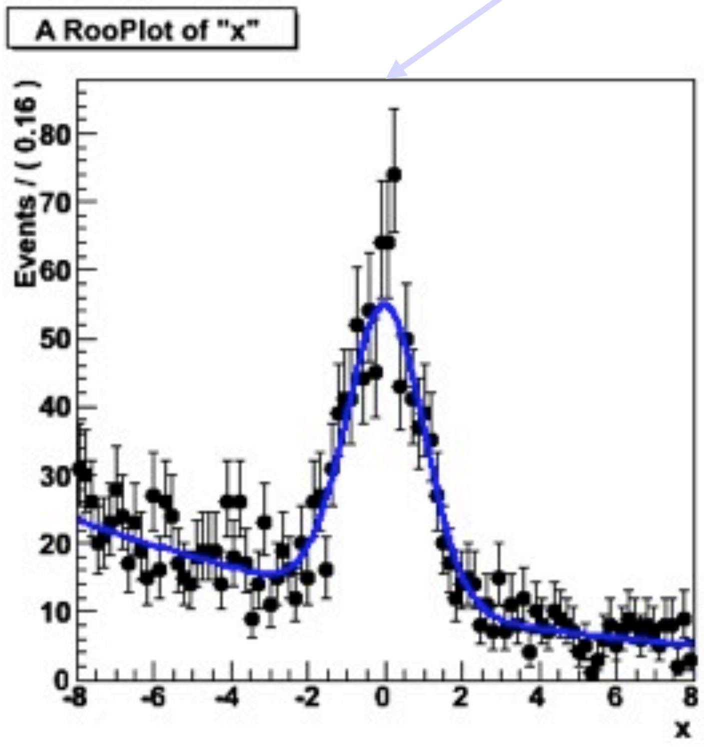
model(x,y) = gauss(x)*gauss(y) + poly(x)*poly(y)

```
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*

Introducing correlations through composition

- RooFit pdf building blocks do not require variables as input, just real-valued functions
  - Can substitute any variable with a function expression in parameters and/or observables

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

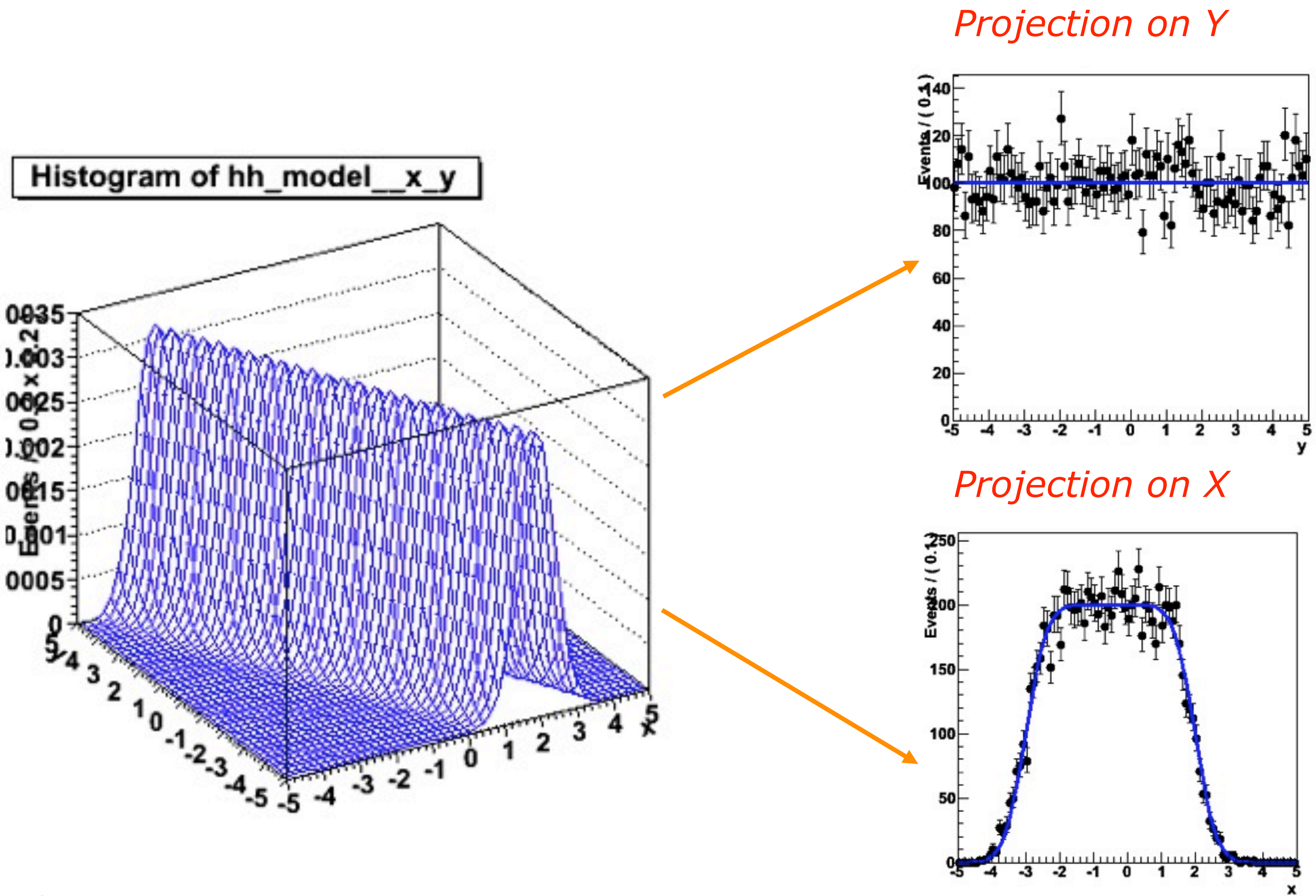
  - Example: Gaussian with shifting mean

```
w.factory(“expr::mean(‘a*y+b’,y[-10,10],a[0.7],b[0.3])”) ;
w.factory(“Gaussian::g(x[-10,10],mean,sigma[3])”) ;
```

  - No assumption made in function on a,b,x,y being observables or parameters, any combination will work

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

- Use example model with x,y as observables

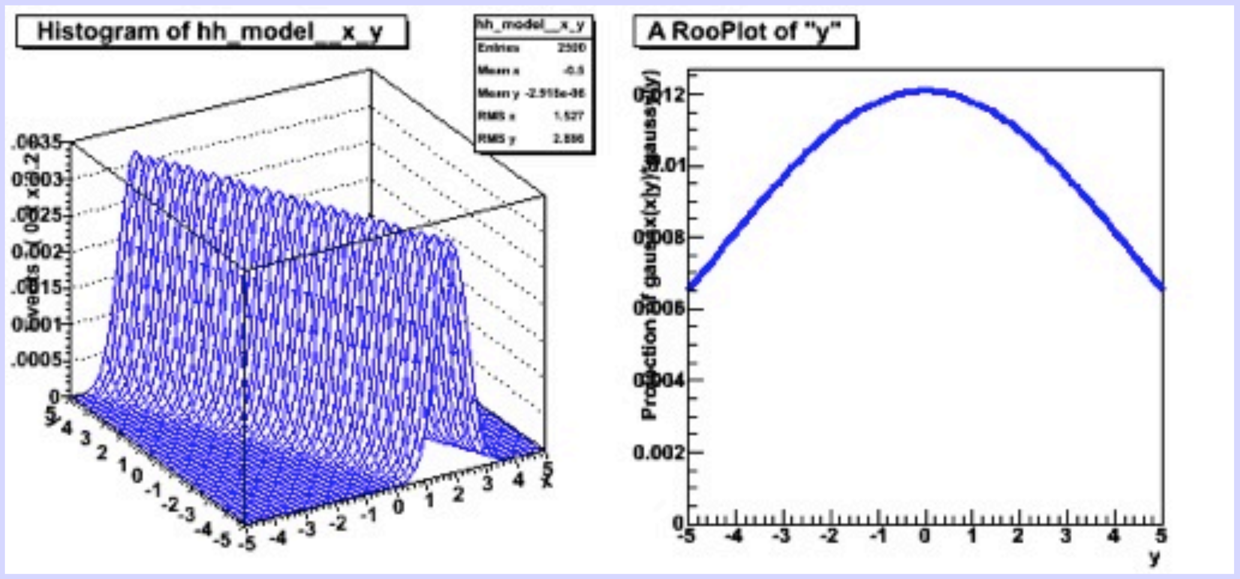


Projection on Y

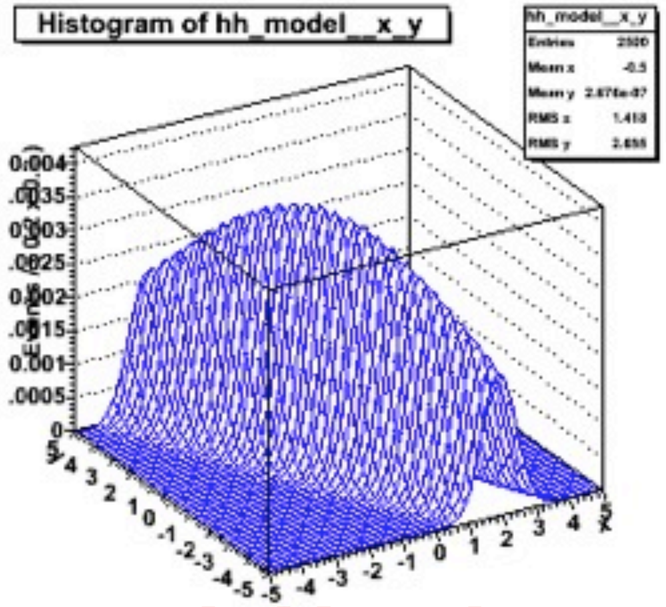
Projection on X

- Note flat distribution in y. Unlikely to describe data, solutions:
  1. Use as conditional p.d.f g(x|y,a,b)
  2. Use in conditional form multiplied by another pdf in y: g(x|y)*h(y)

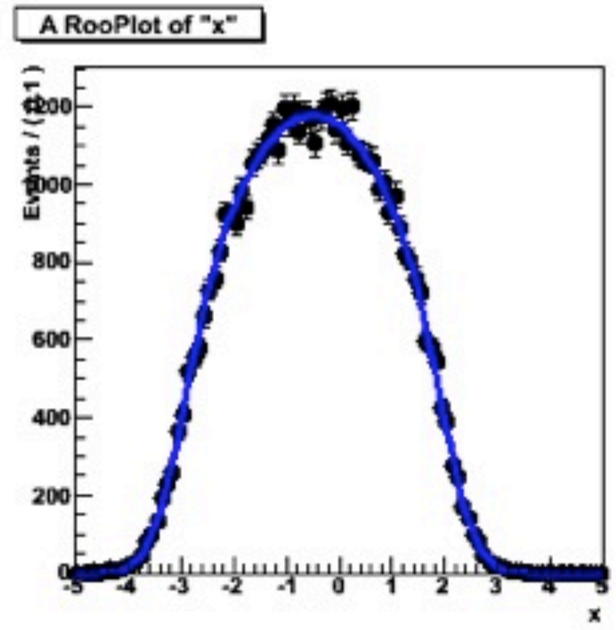
Example with product of conditional and plain p.d.f.



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



```
// I - Use g as conditional pdf g(x|y)
w::g.fitTo(data,ConditionalObservables(w::y)) ;

// II - Construct product with another pdf in y
w.factory(“Gaussian::h(y,0,2)”) ;
w.factory(“PROD::gxy(g|y,h)”) ;
```



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

Special pdfs – Kernel estimation model

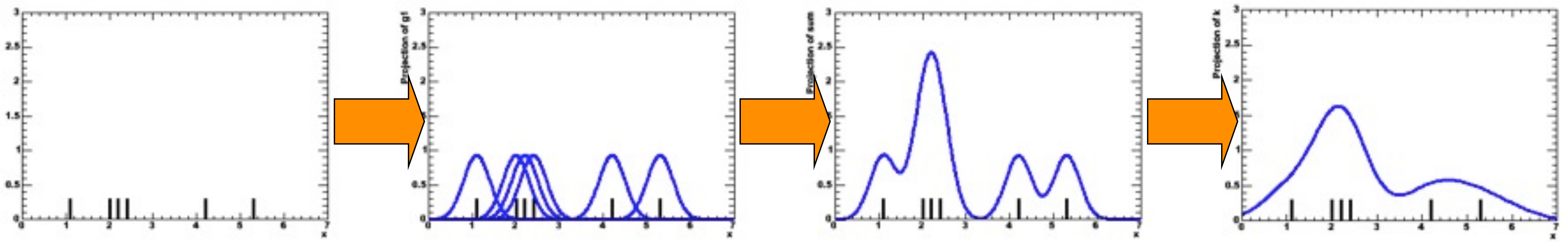
- Kernel estimation model
  - Construct smooth pdf from unbinned data, using kernel estimation technique

Sample of events

Gaussian pdf for each event

Summed pdf for all events

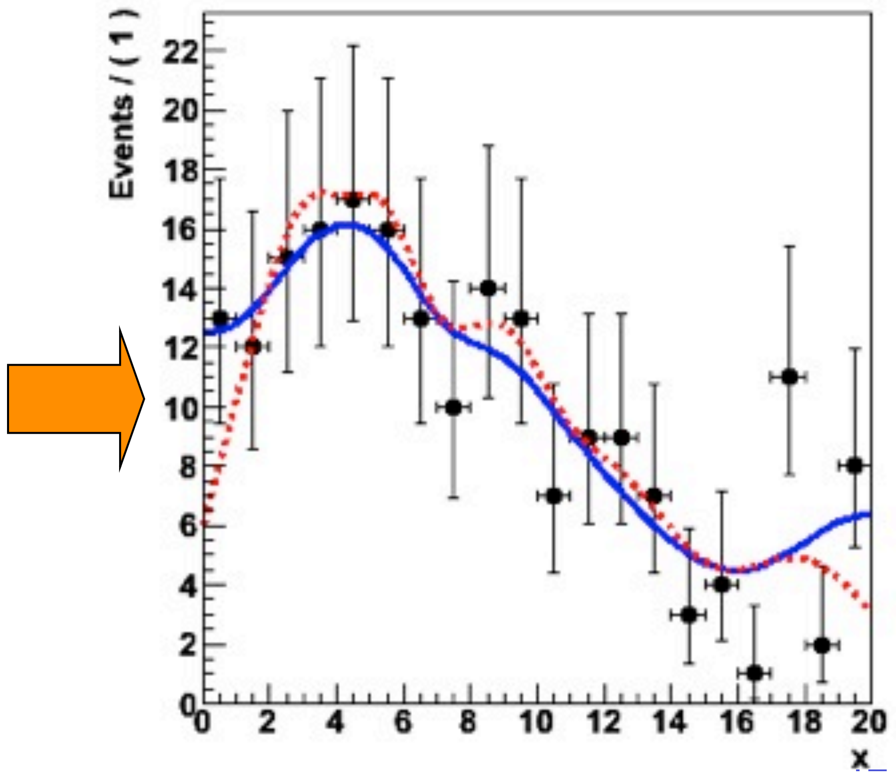
Adaptive Kernel: width of Gaussian depends on local event density



- Example

```
w.import(myData,Rename("myData")) ;
w.factory("KeysPdf::k(x,myData)") ;
```

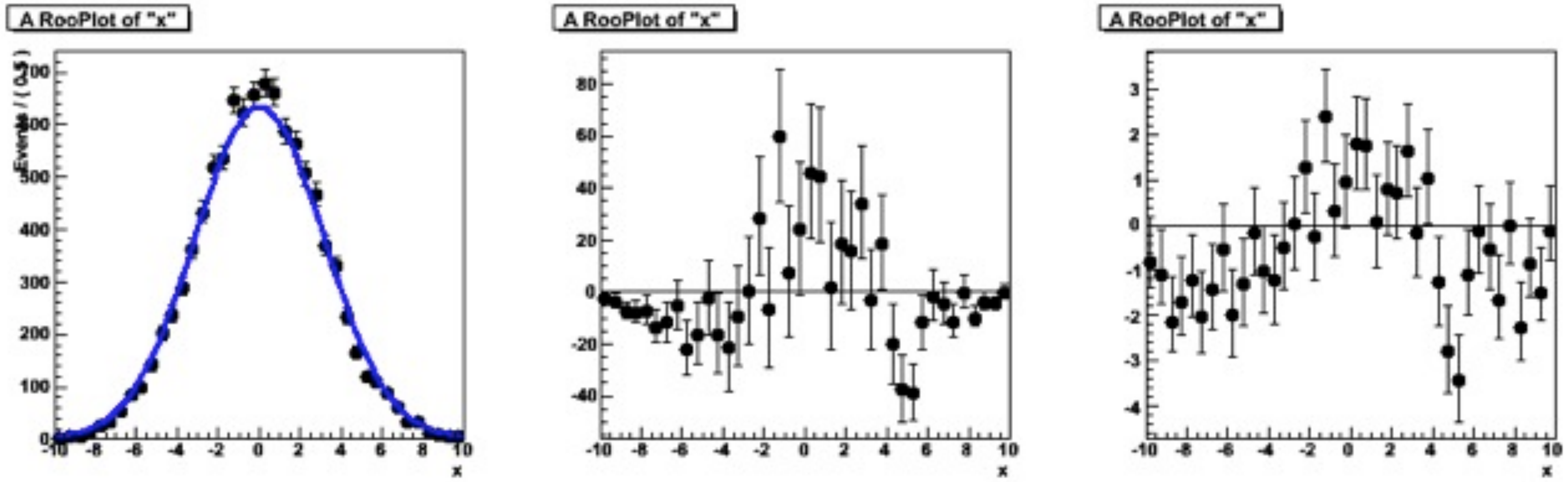
- Also available for n-D data



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

`d`

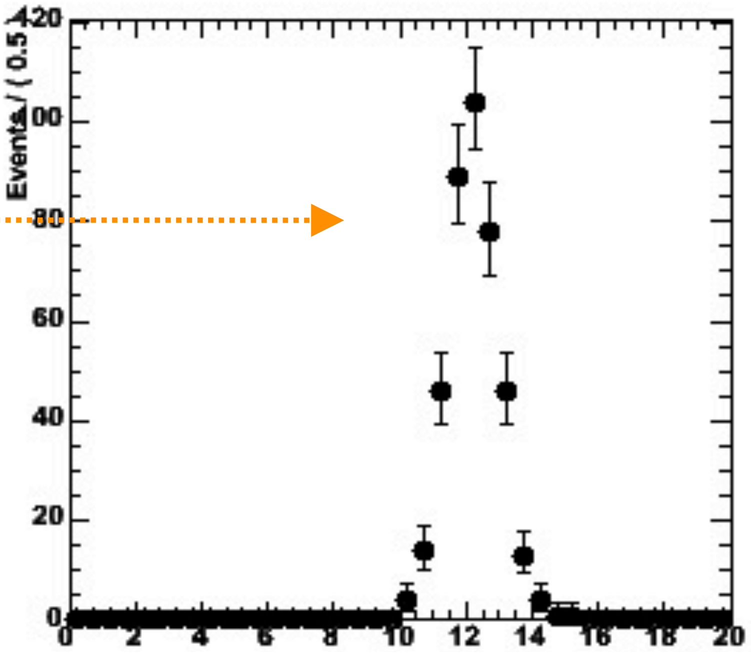


  - Also tools exists to plot residual and pull distributions from curve and histogram in a RooPlot

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

Fit Validation Study – The pull distribution

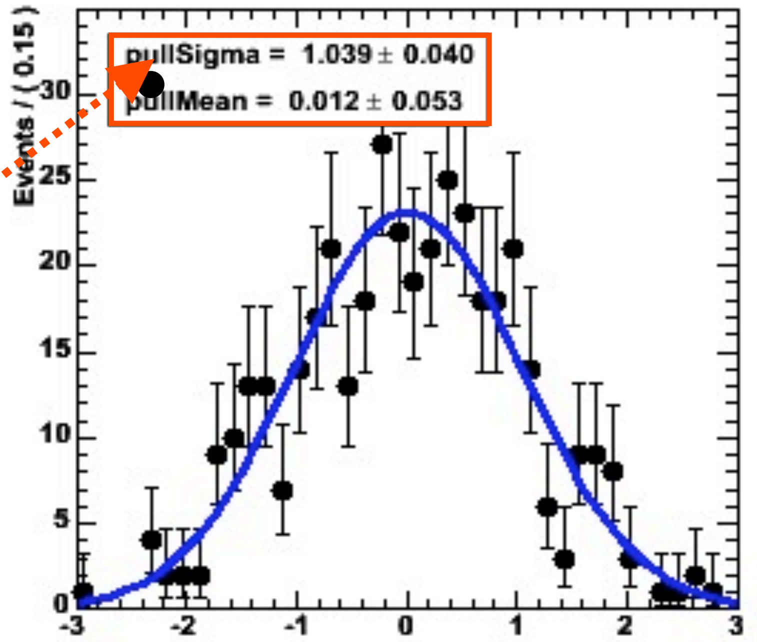
- What about the validity of the error?
  - Distribution of error from simulated experiments is difficult to interpret…
  - We don't have equivalent of N_{sig}(generated) for the error



- Solution: look at the ***pull distribution***
  - Definition:

$$\mathrm{pull}(\mathrm{N_{sig}}) = \frac{N_{sig}^{fit} - N_{sig}^{true}}{\sigma_N^{fit}}$$

  - 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



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

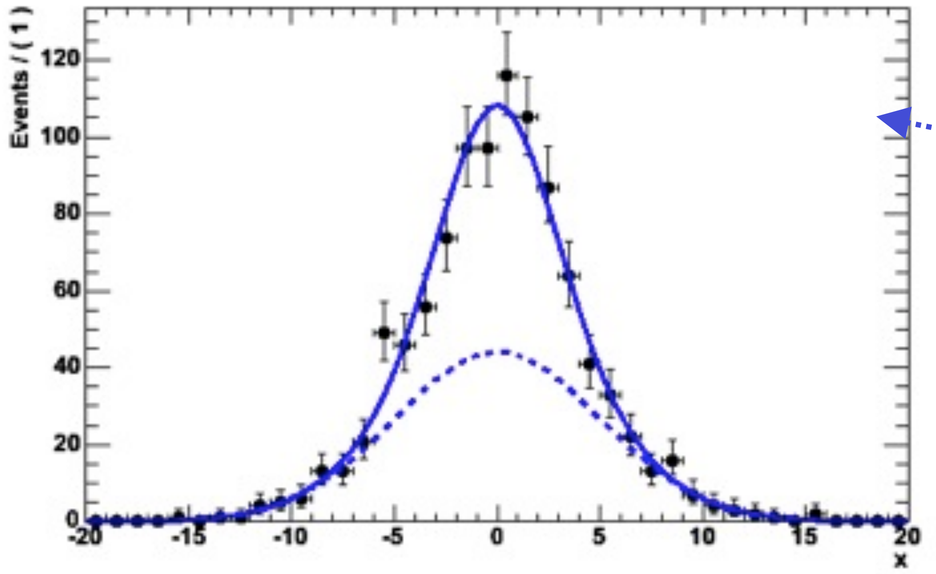
Signs of trouble...

  - 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
```

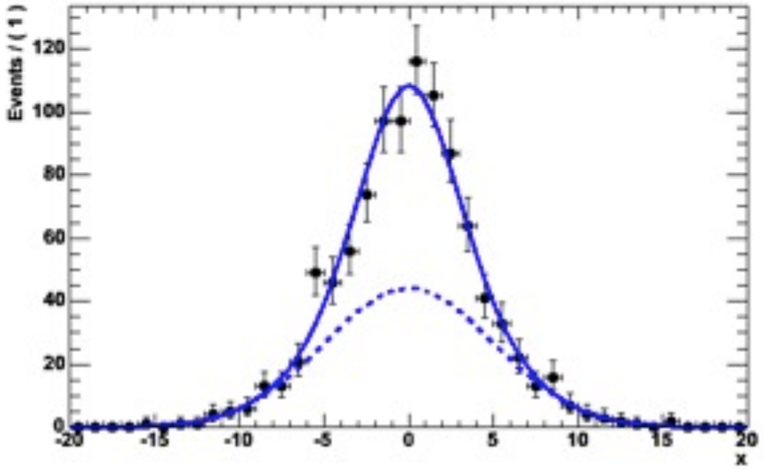
Widths s_1,s_2 strongly correlated fraction f

Mitigating fit stability problems

- Different parameterization:

$$fG_1(x;s_1,m_1)+(1-f)G_2(x;\underline{s_1\times s_2},m_2)$$

```
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
```



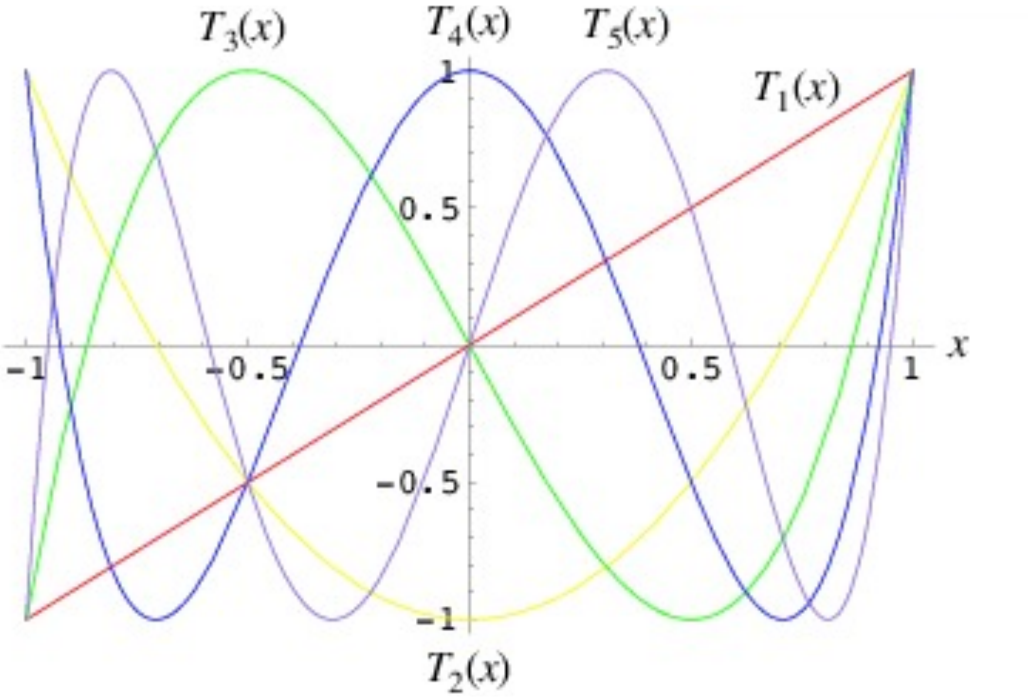
- Correlation of width s2 and fraction f reduced from 0.92 to 0.68
- Choice of parameterization matters!

- 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

Mitigating fit stability problems -- Polynomials

- **Warning:** Regular parameterization of polynomials $a_0+a_1x+a_2x^2+a_3x^3$ nearly always results in strong correlations between the coefficients a_i.
  - *Fit stability problems, inability to find right solution common at higher orders*
- **Solution:** Use existing parameterizations of polynomials that have (mostly) uncorrelated variables
  - ***Example: Chebychev polynomials***

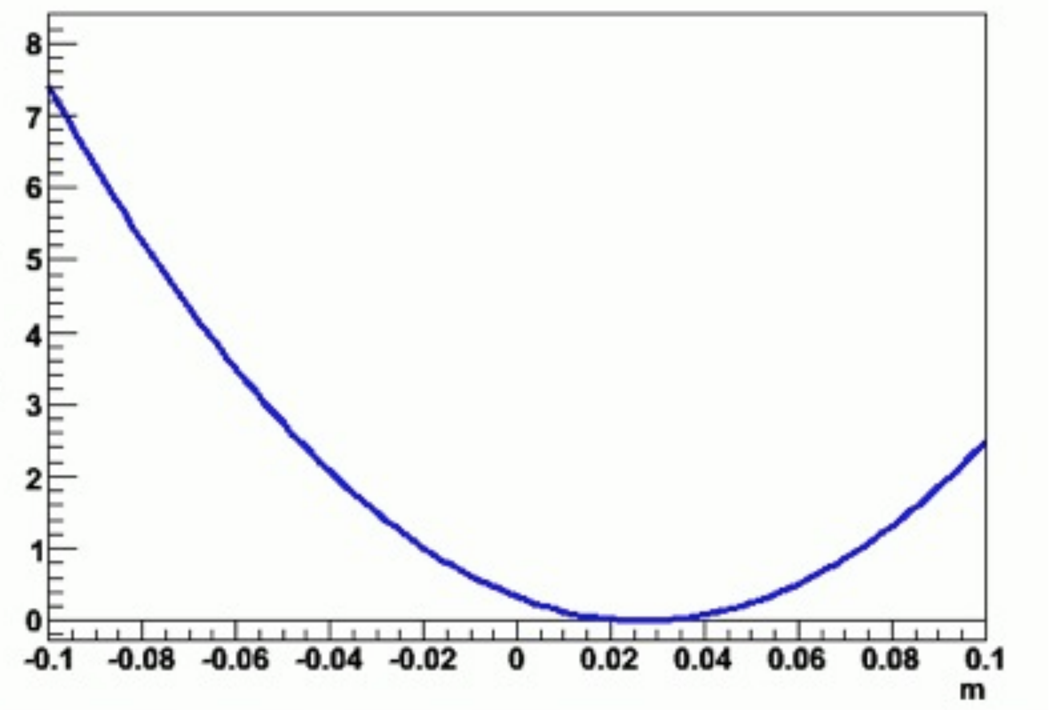
$$
\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 4x^3 - 3x \\
T_4(x) &= 8x^4 - 8x^2 + 1 \\
T_5(x) &= 16x^5 - 20x^3 + 5x \\
T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1.
\end{aligned}
$$



Constructing the likelihood

- So far focus on construction of pdfs, and basic use for fitting and toy event generation
- Can also explicitly construct the likelihood function of and pdf/data combination
  - Can use (plot, integrate) likelihood like any RooFit function object

```
RooAbsReal* nll = w::model.createNLL(data) ;

RooPlot* frame = w::param.frame() ;
nll->plotOn(frame,ShiftToZero()) ;
```



Constructing the likelihood

- Example – Manual MINUIT invocation
  - After each MINUIT command, result of operation are immediately propagated to RooFit variable objects (values and errors)

```
// Create likelihood (calculation parallelized on 8 cores)
RooAbsReal* nll = w::model.createNLL(data,NumCPU(8)) ;

RooMinuit m(*nll) ;   // Create MINUIT session
m.migrad() ;          // Call MIGRAD
m.hesse() ;           // Call HESSE
m.minos(w::param) ;   // Call MINOS for ‘param’

RooFitResult* r = m.save() ; // Save status (cov matrix etc)
```

  - Also other minimizers (Minuit2, GSL etc) supported

```
RooMinimizer m(*nll) ;              // create Minimizer class
m.minimize(“Minuit2”,”Migrad”);     // minimize using Minuit2
```

  - N.B. minimizer can also be used from RooAbsPdf::fitTo

```
//fit a pdf to a data set using Minuit2 as minimizer
pdf.fitTo(*data, RooFit::Minimizer(“Minuit2”,”Migrad”)) ;
```

Adding parameter pdfs to the likelihood

- Systematic/external uncertainties can be modeled with regular RooFit pdf objects.
- To incorporate in likelihood, simply multiply with orig pdf

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

w.factory(“PROD::gprime(f,Gaussian(mean,1.15,0.30))”) ;
```

$$-\log L(\mu,\sigma) = -\sum_{data} -\log(f(x_i;\mu,\sigma) - \log(Gauss(\mu,1.15,0.30))$$

- Any pdf can be supplied, e.g. a **RooMultiVarGaussian** from a **RooFitResult** (or one you construct yourself)

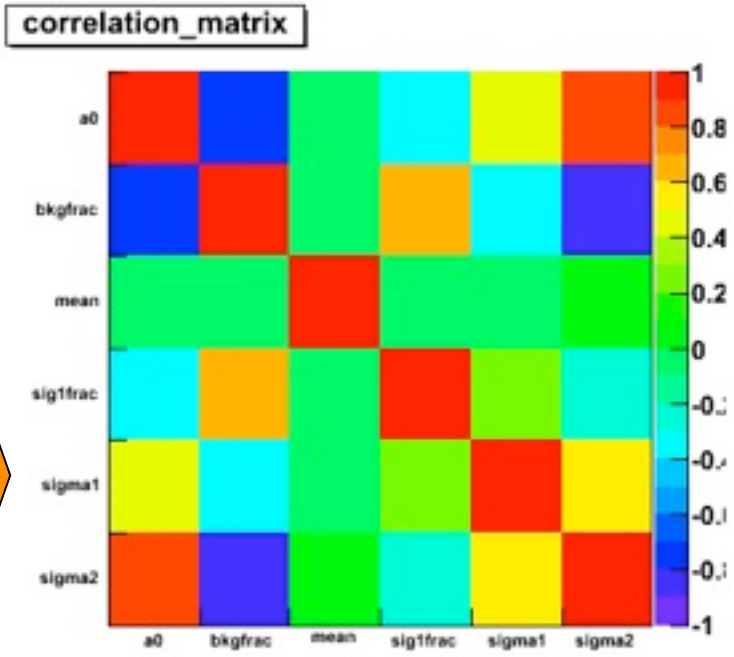
```
w.import(*fitRes->createHessePdf(w::mean,w::sigma),”parampdf”) ;
w.factory(“PROD::gprime(f,parampdf)”) ;
```

Using the fit result output

- The fit result class contains the full MINUIT output

- Easy visualization of correlation matrix

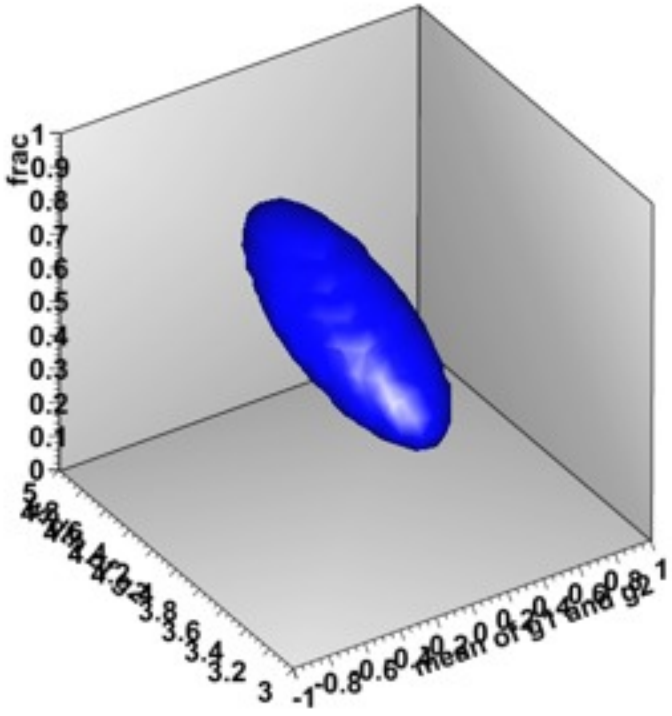
```
fitresult->correlationHist->Draw(“colz”) ;
```



- Construct multi-variate Gaussian pdf representing pdf on parameters

```
RooAbsPdf* paramPdf = fitRes->createHessePdf(RooArgSet(frac,mean,sigma));
```

  - Returned pdf represents HESSE parabolic approximation of fit

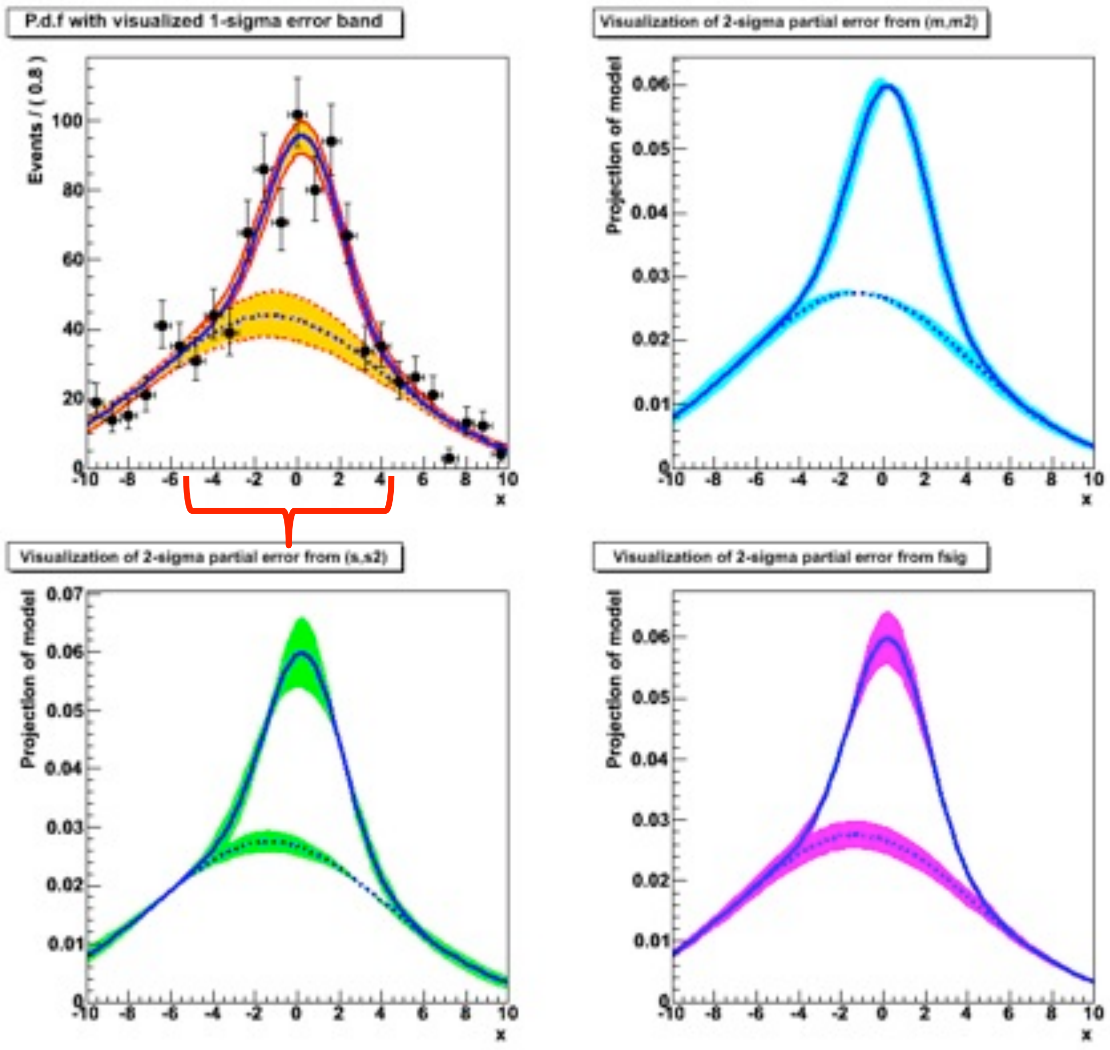


Using the fit result output – Error propagation

- Can (as visual aid) propagate errors in covariance matrix of a fit result to a pdf projection

```
w::model.plotOn(frame,VisualizeError(*fitresult)) ;
w::model.plotOn(frame,VisualizeError(*fitresult,fsig)) ;
```

  - Linear propagation on pdf projection $\Delta = \vec{E} V^{-1} \vec{E}$



- Propagated error can be calculated on arbitrary function
  - E.g fraction of events in signal range

```
RooAbsReal* fracSigRange =
     w::model.createIntegral(x,x,”sig”) ;

Double_t err =
    fracSigRange.getPropagatedError(*fitRes);
```

Working with profile likelihood

- A profile likelihood ratio

$$\lambda(p) = \frac{L(p,\hat{\hat{q}})}{L(\hat{p},\hat{q})}$$

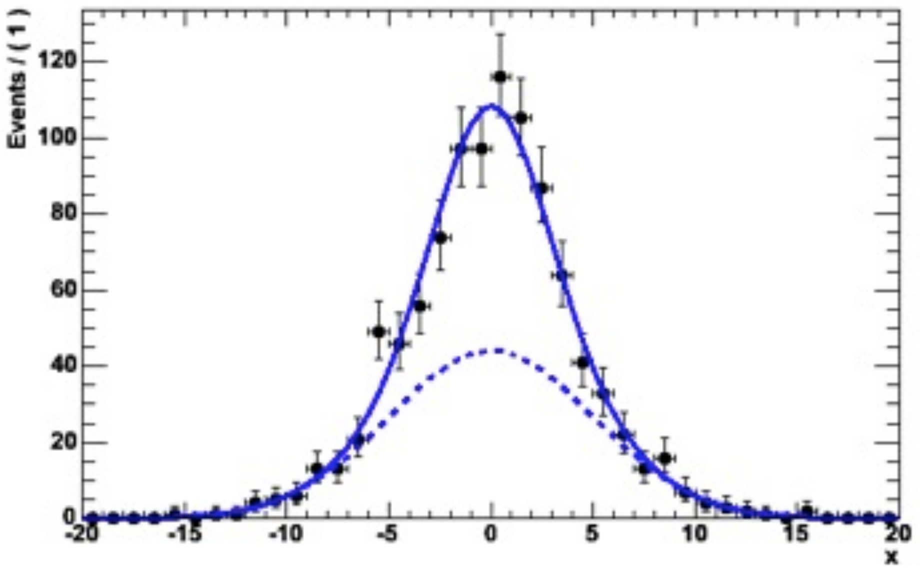
← **Best L for given p**

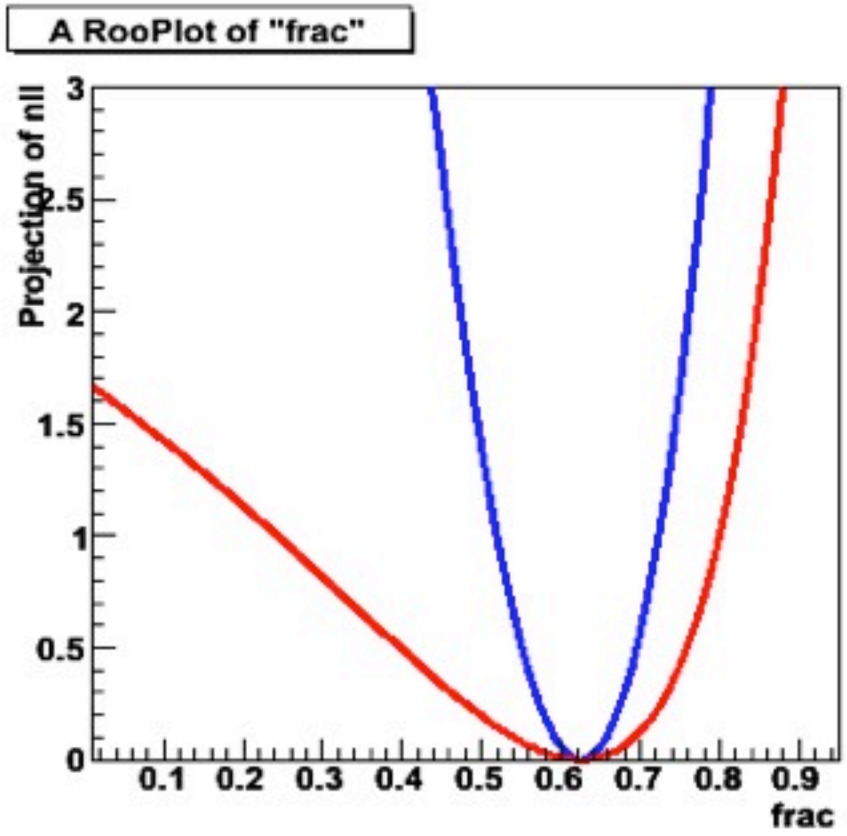
← **Best L**

can be represent by a regular RooFit function (albeit an expensive one to evaluate)

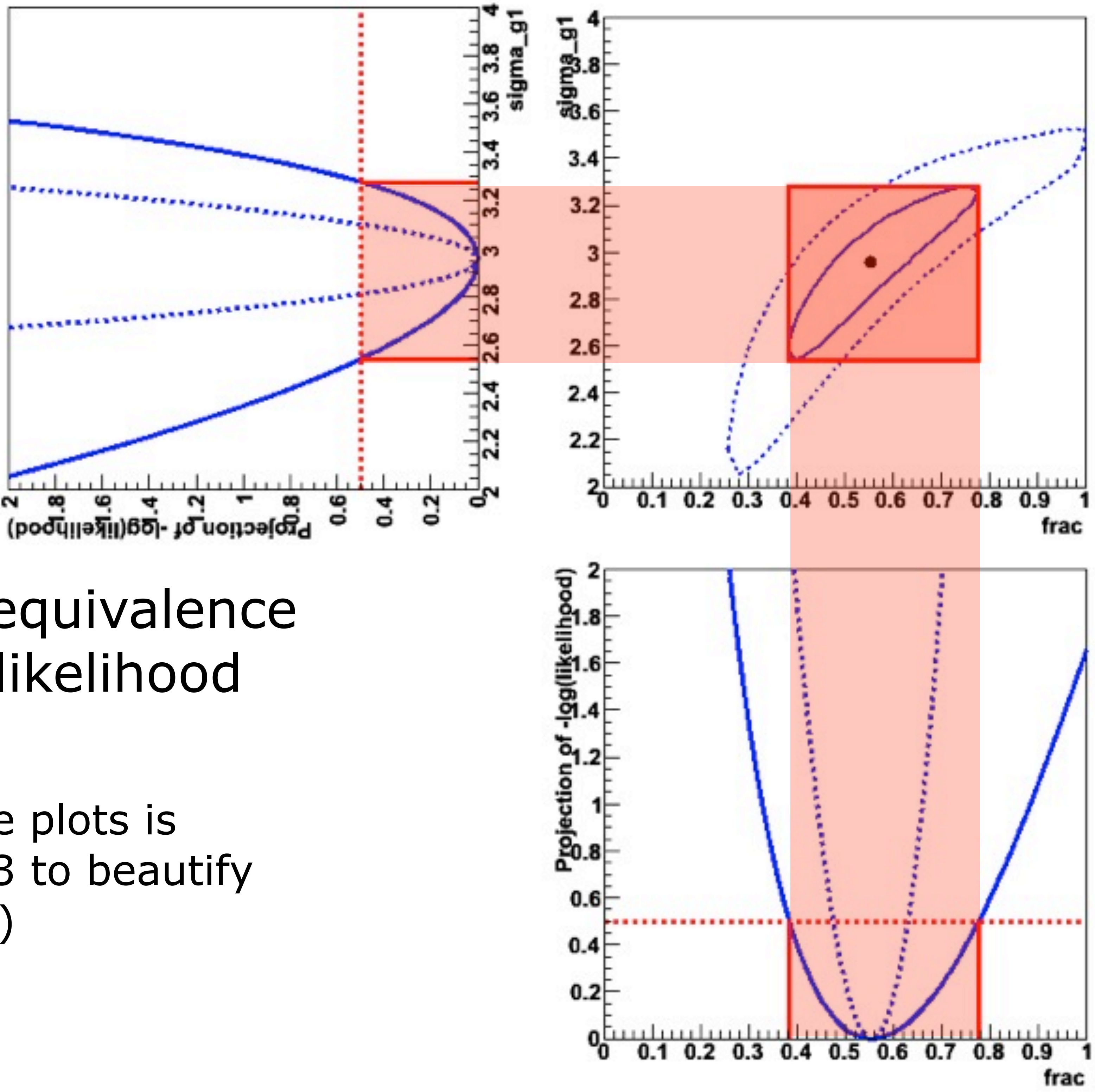
```
RooAbsReal* ll = model.createNLL(data,NumCPU(8)) ;
RooAbsReal* pll = ll->createProfile(params) ;
```

```
RooPlot* frame = w::frac.frame() ;
nll->plotOn(frame,ShiftToZero()) ;
pll->plotOn(frame,LineColor(kRed)) ;
```





On the equivalence of profile likelihood and MINOS



- Demonstration of equivalence of (RooFit) profile likelihood and MINOS errors
  - Macro to make above plots is 34 lines of code (+23 to beautify graphics appearance)