Introduction to RooFit

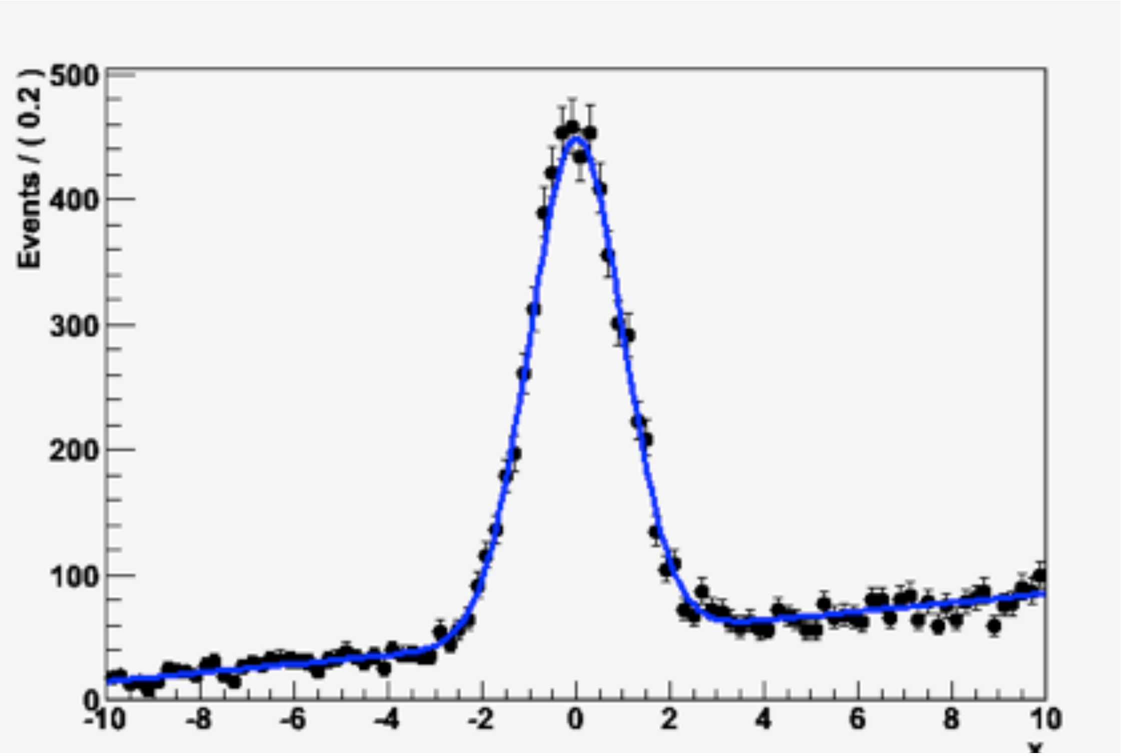
1. Introduction and overview
2. Creation and basic use of models
3. Composing models
4. Working with (profile) likelihood
5. Simultaneous fits and combined models

W. Verkerke (NIKHEF)

1 Introduction & Overview

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) this is easy



  - But if you want to do unbinned ML fits, use non-trivial functions, or work with multidimensional functions you quickly find that you need some tools to help you

Introduction – Why RooFit was developed

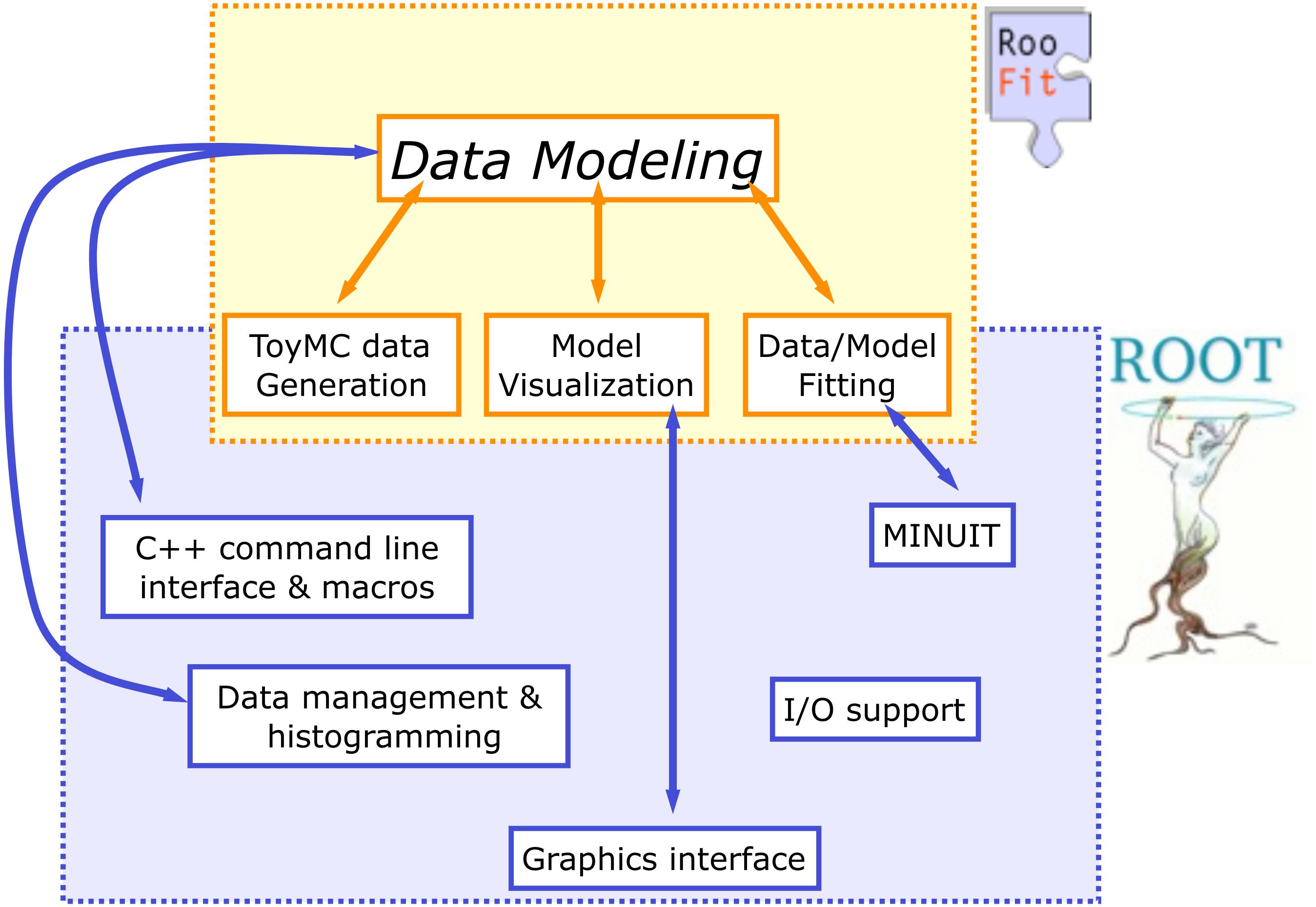
- BaBar experiment at SLAC: Extract sin(2β) from time_ dependent CP violation of B decay: e^+e^- → Y(4s) → BB
  - Reconstruct both Bs, measure decay time difference
  - Physics of interest is in decay time dependent oscillation

$$
\begin{aligned}
& f_{sig} \times \Big[\mathrm{SigSel}(m;\bar{p}_{sig}) \times \Big(\mathrm{SigDecay}(t;\vec{q}_{sig},\sin(2\beta)) \otimes \mathrm{SigResol}(t\,|\,dt;\vec{r}_{sig})\Big)\Big] + \\
& (1-f_{sig})\Big[\mathrm{BkgSel}(m;\bar{p}_{bkg}) \times \Big(\mathrm{BkgDecay}(t;\vec{q}_{bkg}) \otimes \mathrm{BkgResol}(t\,|\,dt;\vec{r}_{bkg})\Big)\Big]
\end{aligned}
$$

- Many issues arise
  - Standard ROOT function framework clearly insufficient to handle such complicated functions → must develop new framework
  - Normalization of p.d.f. not always trivial to calculate → may need numeric integration techniques
  - Unbinned fit, >2 dimensions, many events → computation performance important → must try optimize code for acceptable performance
  - Simultaneous fit to control samples to account for detector performance

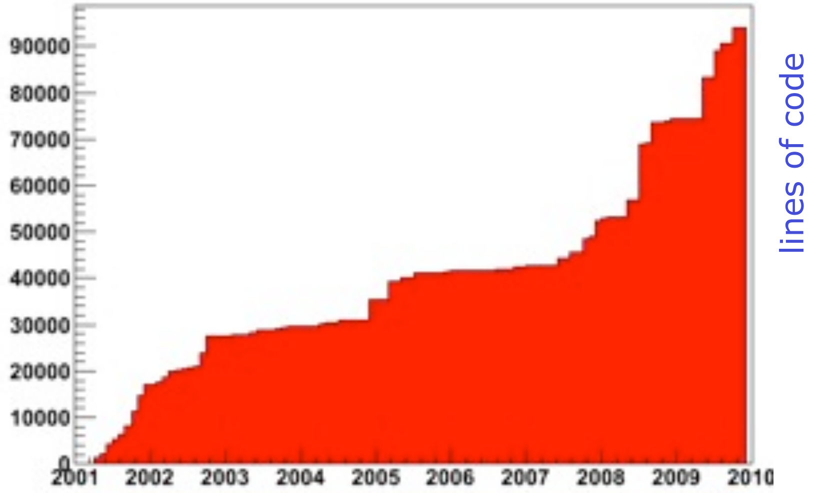
Introduction – Relation to ROOT

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



Project timeline

- 1999 : Project started
  - First application: ‘sin2b’ measurement of BaBar (model with 5 observables, 37 floating parameters, simultaneous fit to multiple CP and control channels)
- 2000 : Complete overhaul of design based on experience with sin2b fit
  - Very useful exercise: new design is still current design
- 2003 : Public release of RooFit with ROOT
- 2004 : Over 50 BaBar physics publications using RooFit
- 2007 : Integration of RooFit in ROOT CVS source
- 2008 : Upgrade in functionality as part of RooStats project
  - Improved analytical and numeric integration handling, improved toy MC generation, addition of workspace
- 2009 : Now ~100K lines of code
  - (For comparison RooStats proper is ~5000 lines of code)



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**

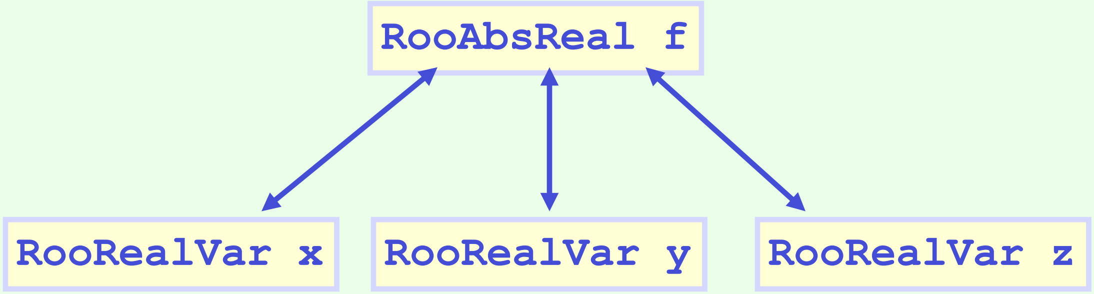
RooFit core design philosophy

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

Math

$f(x,y,z)$

RooFit diagram



RooAbsReal f

RooRealVar x

RooRealVar y

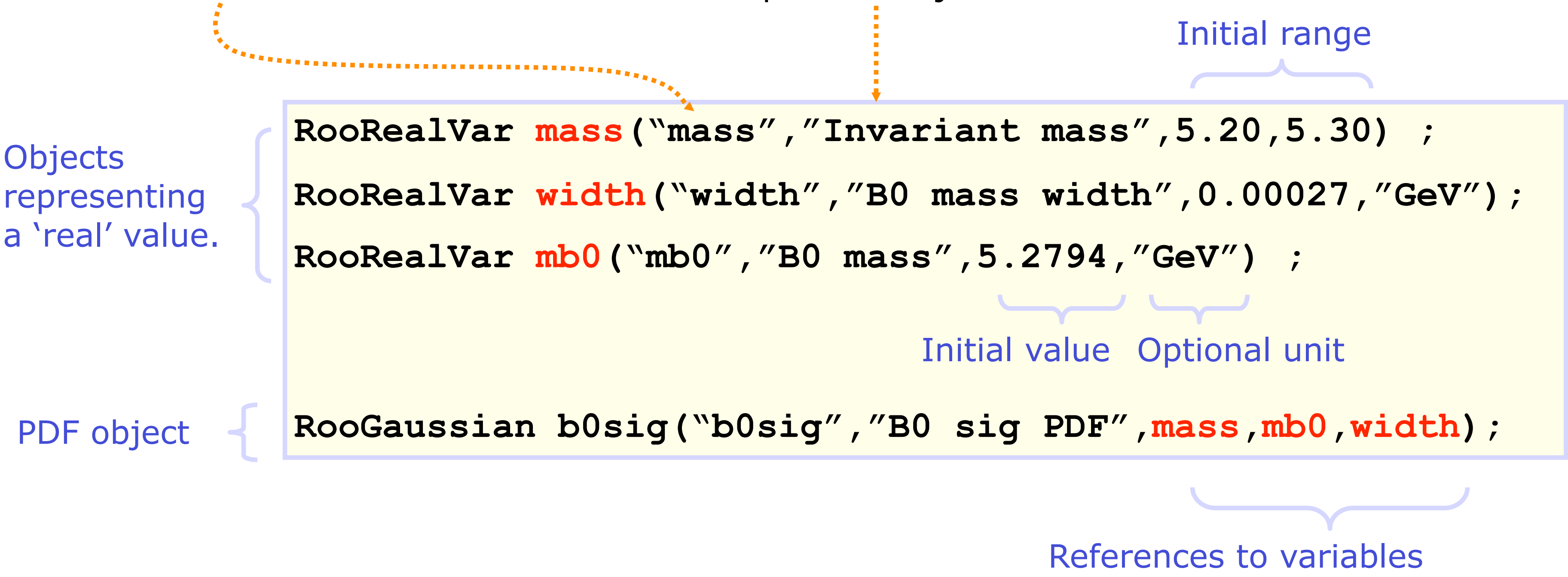
RooRealVar z

RooFit code

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

Object-oriented data modeling

- All objects are **self documenting**
  - **Name** - Unique identifier of object
  - **Title** – More elaborate description of object

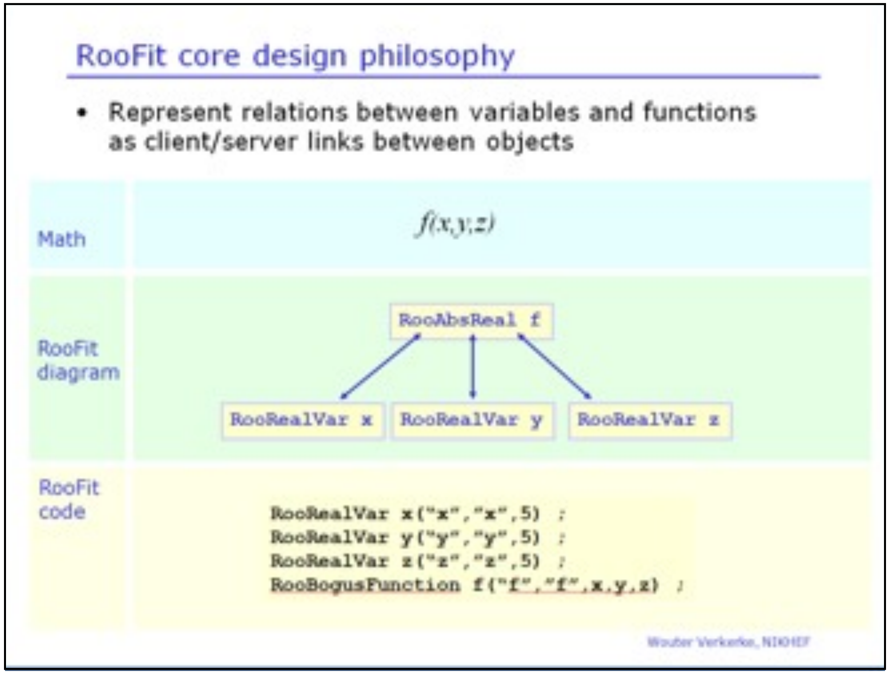


```
RooRealVar mass(“mass”,”Invariant mass”,5.20,5.30) ;
RooRealVar width(“width”,”B0 mass width”,0.00027,”GeV”);
RooRealVar mb0(“mb0”,”B0 mass”,5.2794,”GeV”) ;

RooGaussian b0sig(“b0sig”,”B0 sig PDF”,mass,mb0,width);
```

2 Basic use

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



RooFit core design philosophy

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

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

Wouter Verkerke, NIKHEF

```
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 and Workspace

- This is *not* the same as reinventing Mathematica!
  String **constructs** an expression in terms of C++ objects, rather than being the expression
  - Objects can be tailored after construction through object pointers
  - For example: tune parameters and algorithms of numeric integration to be used with a given object
- Implementation in RooFit:

  Factory makes objects, **Workspace** owns them

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

w.Print(“t”) ;
variables
---------
(mean,sigma,x)

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

Accessing the workspace contents

- Contents can be accessing in two ways

- Through C++ namespace corresponding through w’space
  - Super easy
  - But works in ROOT interpreted macros only

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

w::g.Print() ;
```

- Through accessor methods
  - A bit more clutter, but 100% ISO compliant C++ (and compilable)

```
RooAbsPdf* g = w.pdf(“g”) ;
RooRealVar* x = w.var(“x”) ;
```

Factory language

- The factory language has a 1-to-1 mapping to the constructor syntax of RooFit classes
  - With a few handy shortcuts for variables

- Creating variables

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

- Creating pdfs (and functions)

```
Gaussian::g(x,mean,sigma) → RooGaussian(“g”,”g”,x,mean,sigma)
Polynomial::p(x,{a0,a1}) → RooPolynomial(“p”,”p”,x”,RooArgList(a0,a1));
```

  - Can always omit leading ‘Roo’
  - Curly brackets translate to set or list argument (depending on context)

Factory language

- Composite expression are created by nesting statements
  - No limit to recursive nesting

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

- You can also use numeric constants whenever an unnamed constant is needed

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

- Names of nested function objects are optional
    - SUM syntax explained later

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

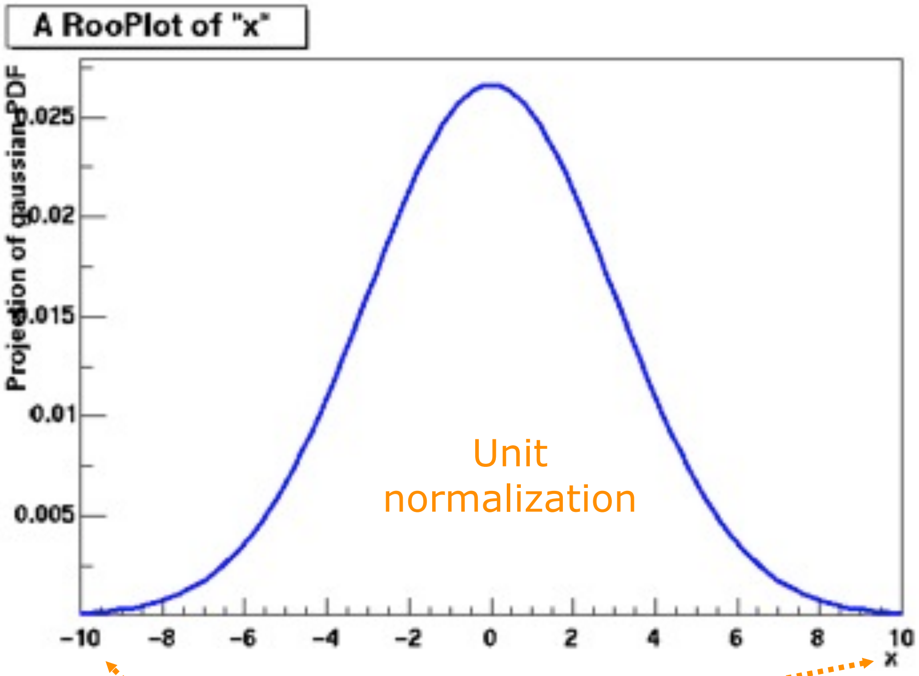
Basics – Creating and plotting a Gaussian p.d.f

Setup gaussian PDF and plot

```
// Build Gaussian PDF
w.factory(“Gaussian::gauss(x[-10,10],mean[-10,10],sigma[3,1,10]”)

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

Axis label from `gauss` title

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



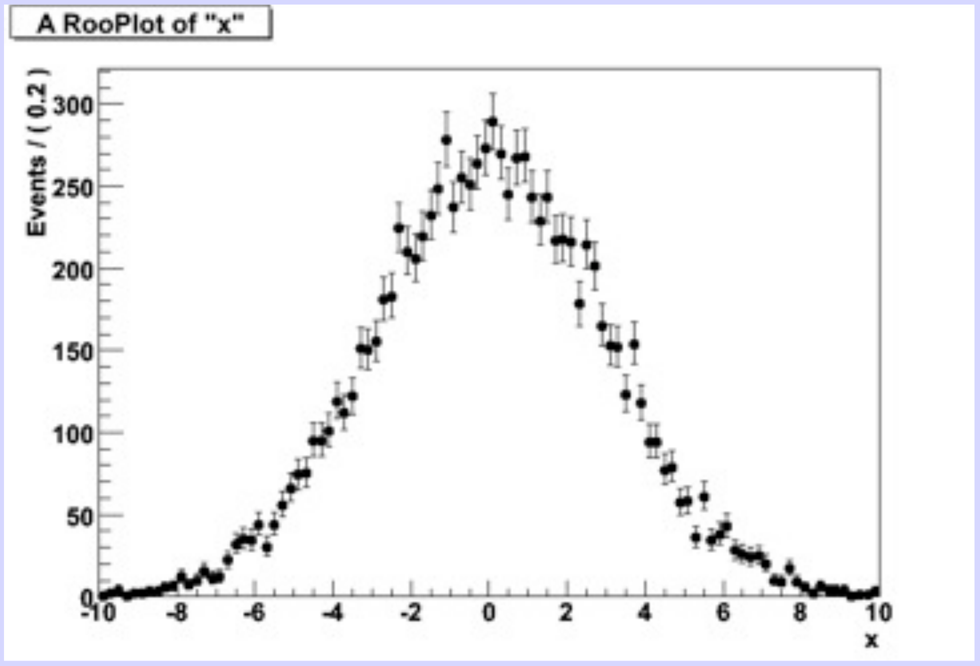
Unit normalization

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 = w::gauss.generate(w::x,10000) ;

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

// Plot PDF
RooPlot* xframe = w::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”,w::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

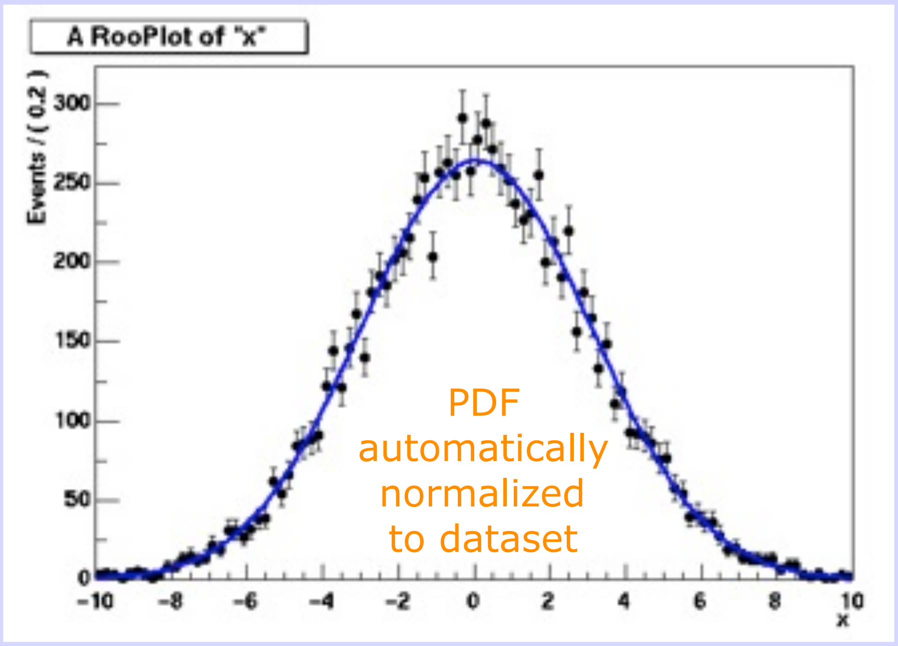
- Binned data can be imported from ROOT **THx** histograms

```
// Import unbinned data
RooDataHist data(“data”,”data”,w::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
w::gauss.fitTo(*data) ;
(MINUIT printout omitted)

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

// Plot fitted PDF and toy data overlaid
RooPlot* xframe = w::x.frame() ;
data->plotOn(xframe) ;
w::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 = w::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
```

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) must be passed to **gauss.getVal()** when called directly

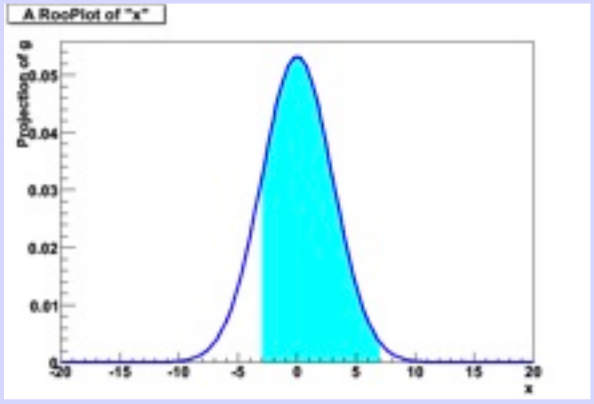
```
gauss.getVal() ;  // Not normalized (i.e. this is _not_ a pdf)
gauss.getVal(x) ; // Guarantees Int[xmin,xmax] Gauss(x,m,s)dx==1
gauss.getVal(s) ; // Guarantees Int[smin,smax] Gauss(x,m,s)ds==1
```

  - This flexibility is useful in e.g. Bayesian context

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

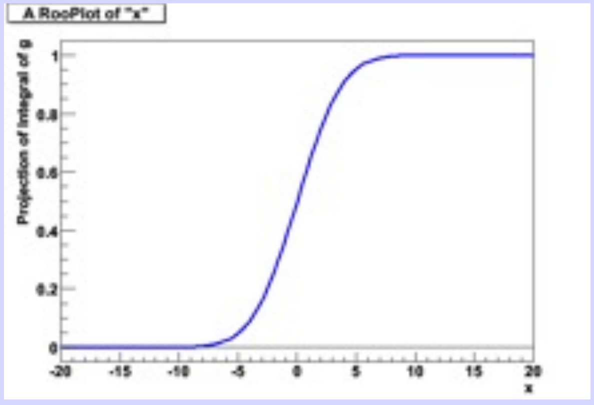
```
w::x.setRange(“sig”,-3,7) ;
RooAbsReal* ig = w::g.createIntegral(x,NormSet(x),Range(“sig”)) ;
cout << ig.getVal() ;
0.832519
mean=-1 ;
cout << ig.getVal() ;
0.743677
```



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

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

```
RooAbsReal* cdf = gauss.createCdf(x) ;
```



Model building – (Re)using standard components

- RooFit provides a collection of compiled standard PDF classes

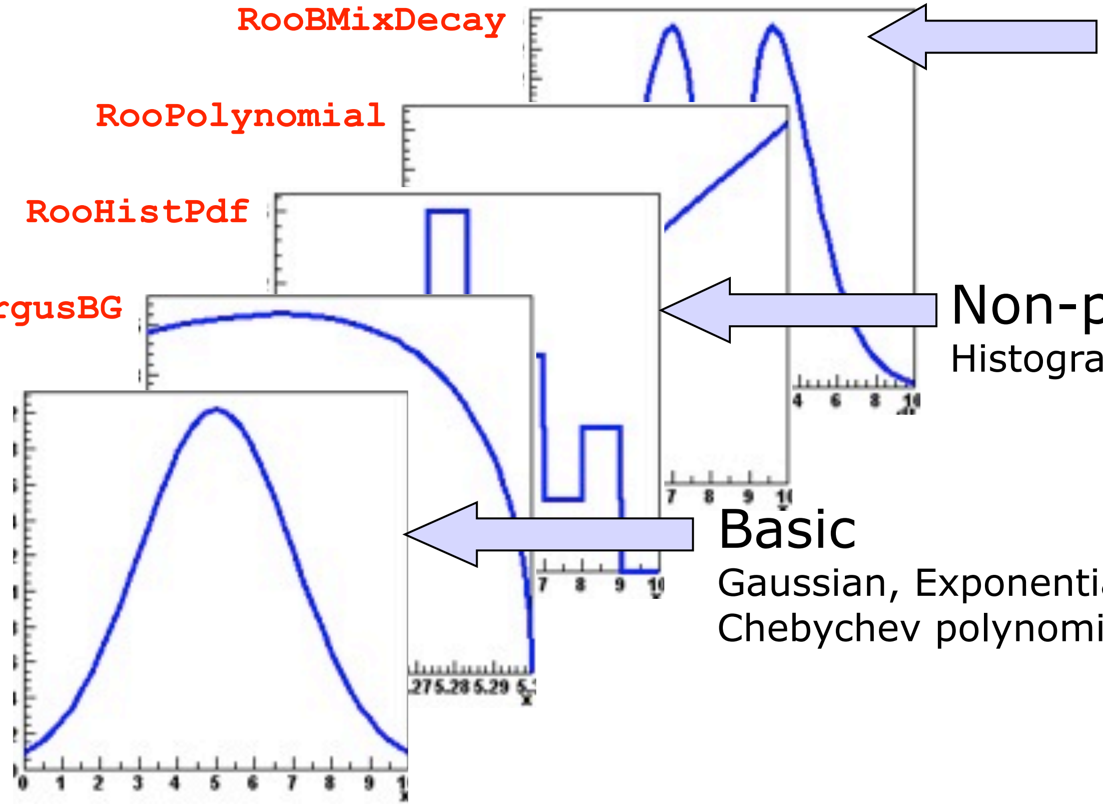
RooBMixDecay

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	`Exponental::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

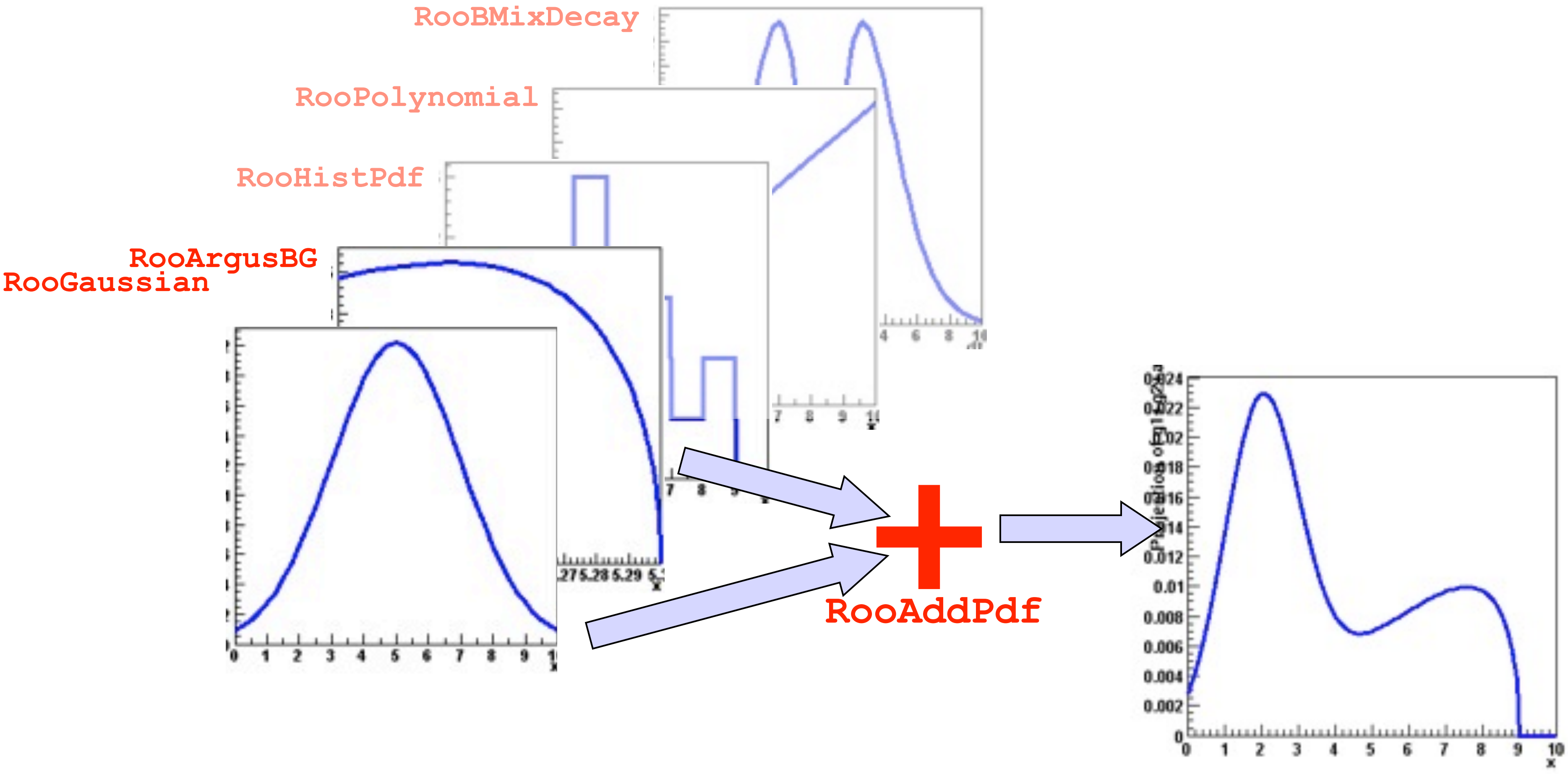
```
w.factory(“CEXPR::mypdf(‘sqrt(a*x)+b’,x,a,b)”) ;
```

- 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

3 Composite models

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 – 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

$$S(x) = c_0P_0(x) + c_1P_1(x) + \ldots + c_{n-1}P_{n-1}(x) + \left(1 - \sum_{i=0,n-1} c_i\right)P_n(x)$$

- 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

Adding p.d.f.s – Factory syntax

- Additions created through a SUM expression

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

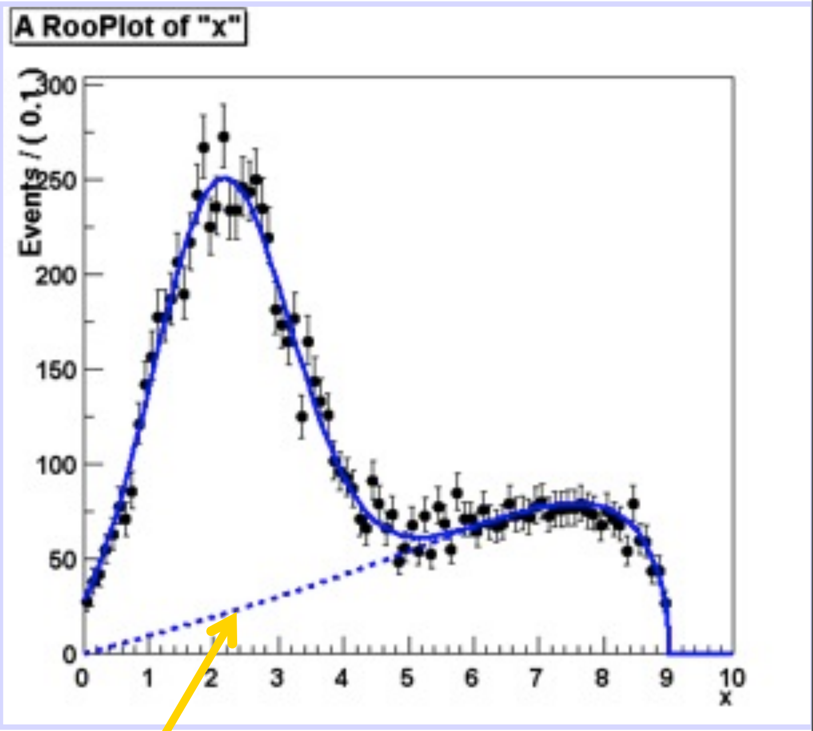
  - Note that last PDF does not have an associated fraction

- 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

Poisson(N_{obs},N_{exp})

- This is most useful in combination with a composite pdf

shape *normalization*

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

$$f, N \Rightarrow N_S, N_B$$

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

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

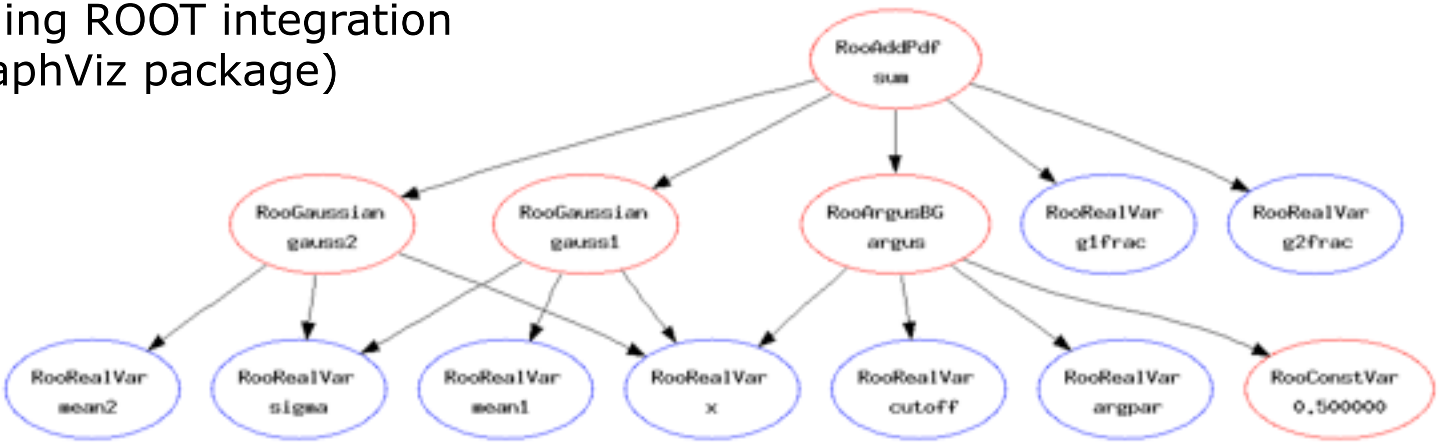
```
SUM::name(Nsig*S,Nbkg*B)
```

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
```

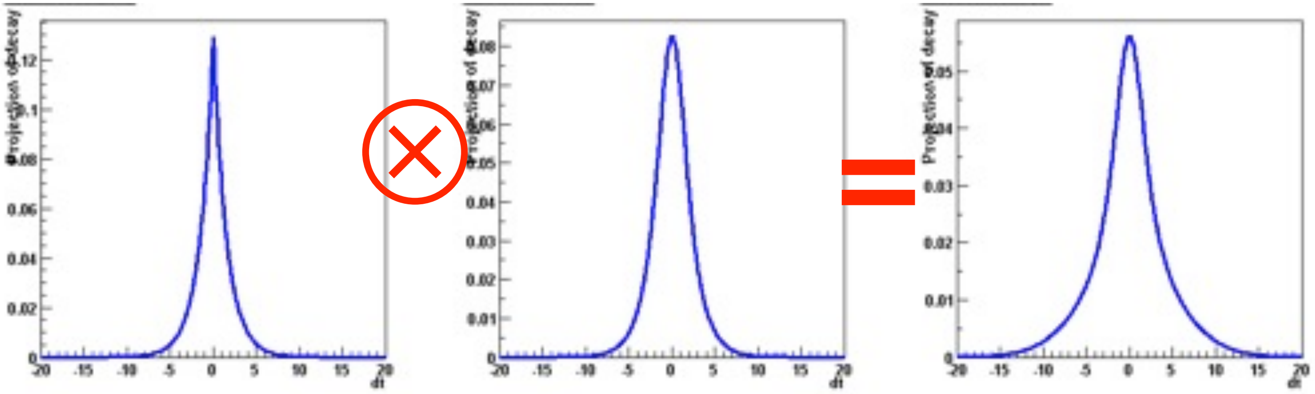
  - Can also make input files for GraphViz visualization (**w::sum.graphVizTree("myfile.dot")**)
  - Graph output on ROOT Canvas in near future (pending ROOT integration of GraphViz package)



Convolution

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

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



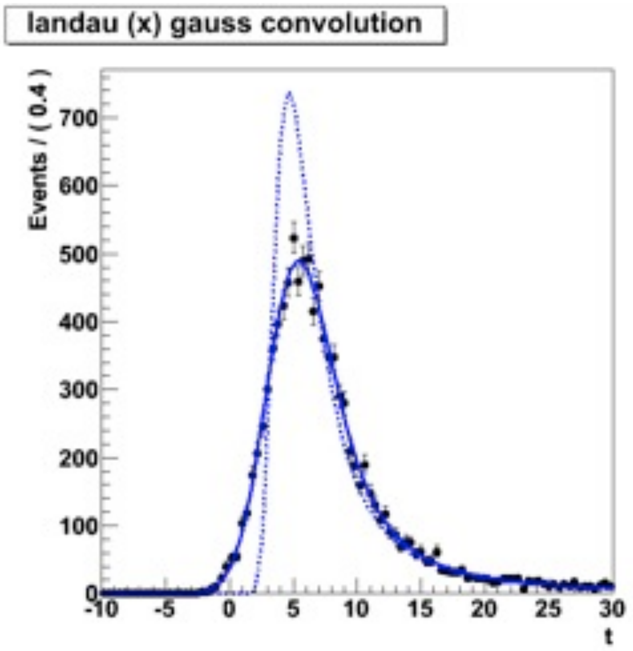
- But numeric calculation of convolution integral can be challenging. No one-size-fits-all solution, but 3 options available
  - Analytical convolution (BW⊗Gauss, various B physics decays)
  - Brute-force numeric calculation (slow)

Convolution

- Example

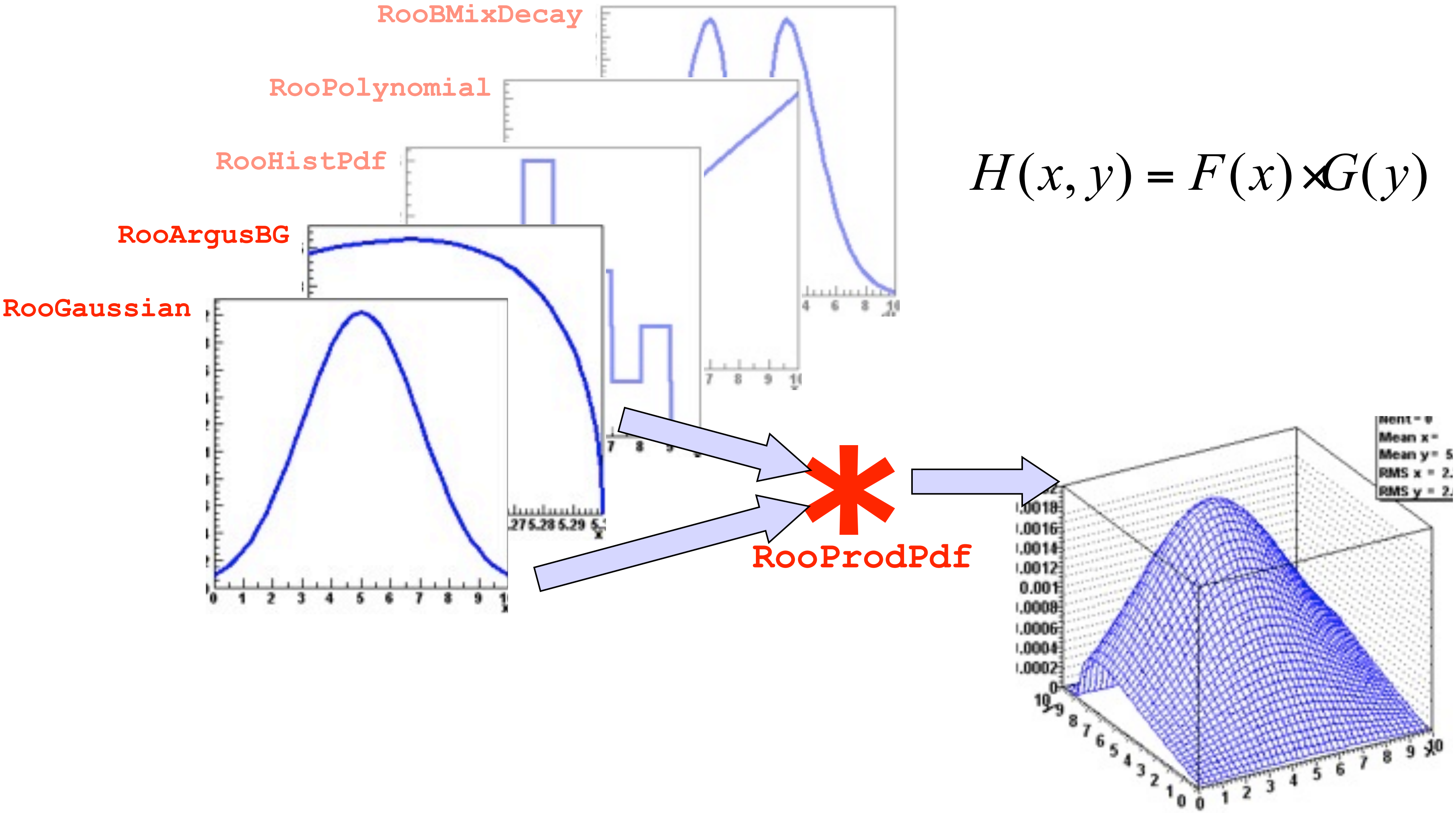
```
  w.factory(“Landau::L(x[-10,30],5,1)”) :
  w.factory(“Gaussian::G(x,0,2)”) ;

  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

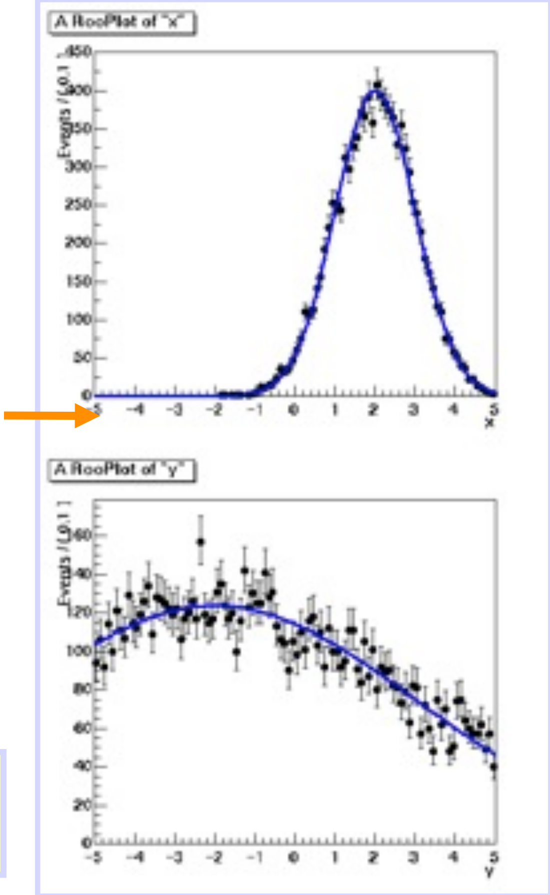
```
w.factory(“Gaussian::gx(x[-5,5],mx[2],sx[1])”) ;
w.factory(“Gaussian::gy(y[-5,5],my[-2],sy[3])”) ;

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

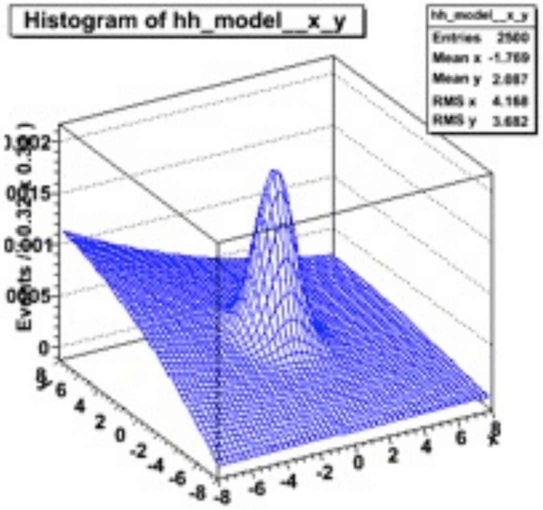
```
TH2* hh = w::gxy.createHistogram(“x,y”,50,50);
```

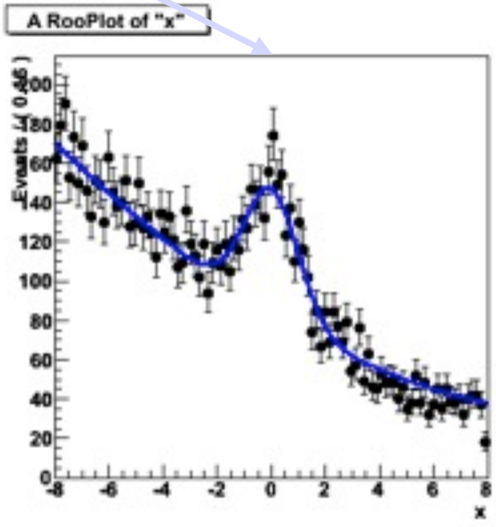
Can also project slices of a multi-dimensional pdf

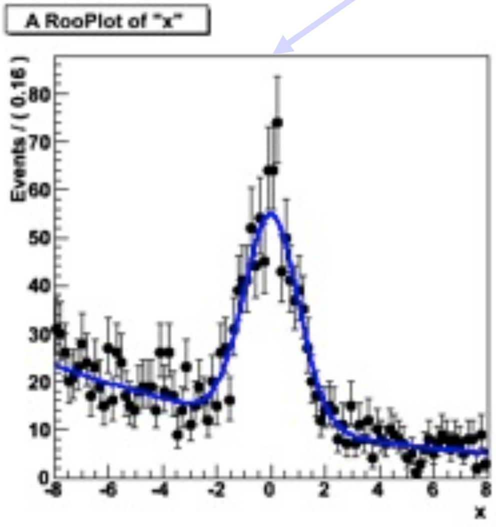
model(x,y) = gauss(x)*gauss(y) + poly(x)*poly(y)

```
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

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

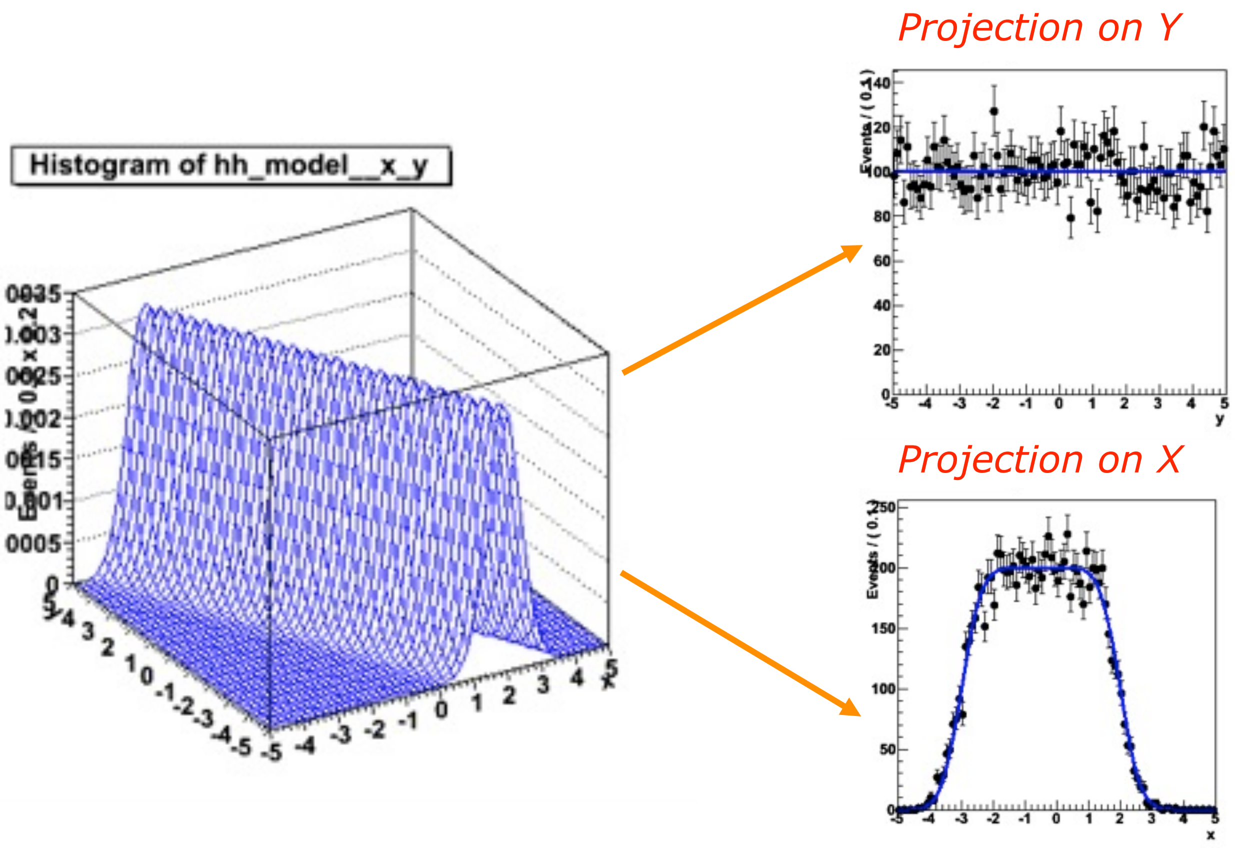
  - Example: Gaussian with shifting mean

```
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

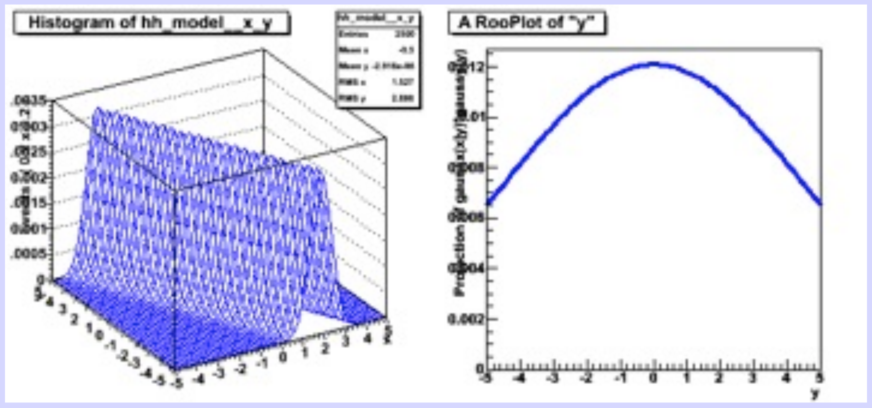


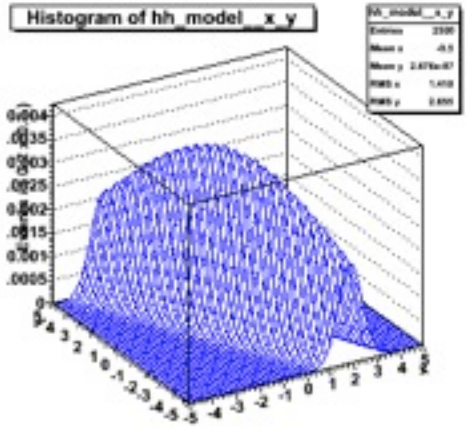
Projection on Y

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)

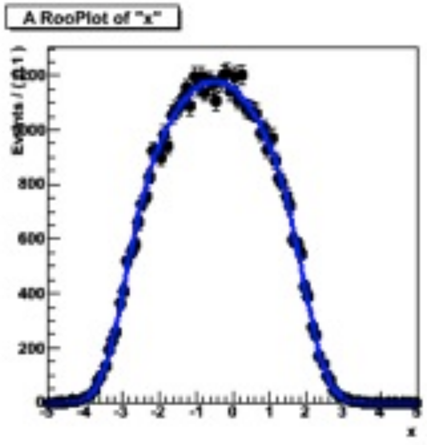
Example with product of conditional and plain p.d.f.





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

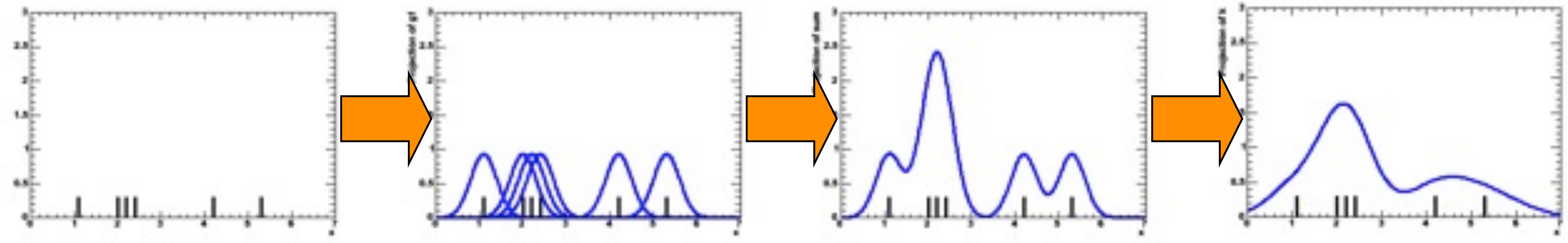
```
// I - Use g as conditional pdf g(x|y)
w::g.fitTo(data,ConditionalObservables(w::y)) ;

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



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

Special pdfs – Kernel estimation model

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



Sample of events

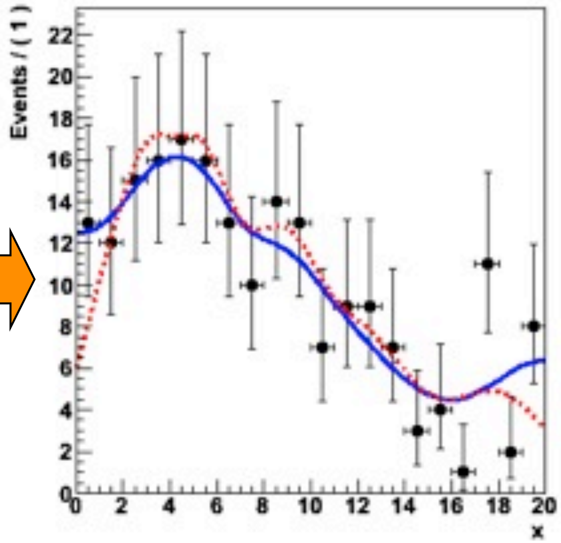
Gaussian pdf for each event

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)”) ;
```

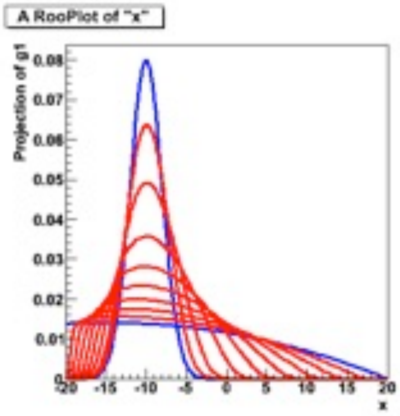


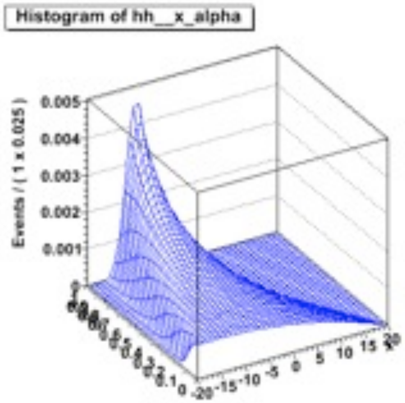
- Also available for n-D data

Special pdfs – Morphing interpolation

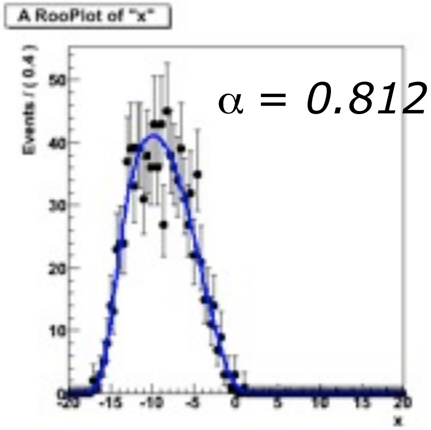
- Special operator pdfs can interpolate existing pdf shapes
  - Ex: interpolation between Gaussian and Polynomial

```
w.factory(“Gaussian::g(x[-20,20],-10,2)”) ;
w.factory(“Polynomial::p(x,{-0.03,-0.001})”) ;
w.factory(“IntegralMorph::gp(g,p,x,alpha[0,1])”) ;
```





Fit to data



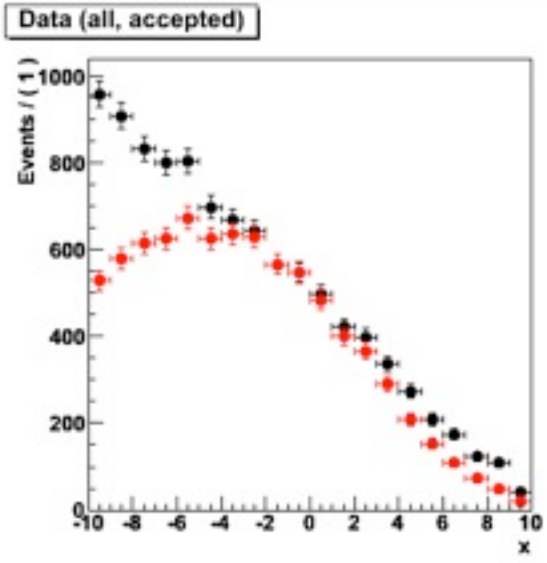
α = *0.812 ± 0.008*

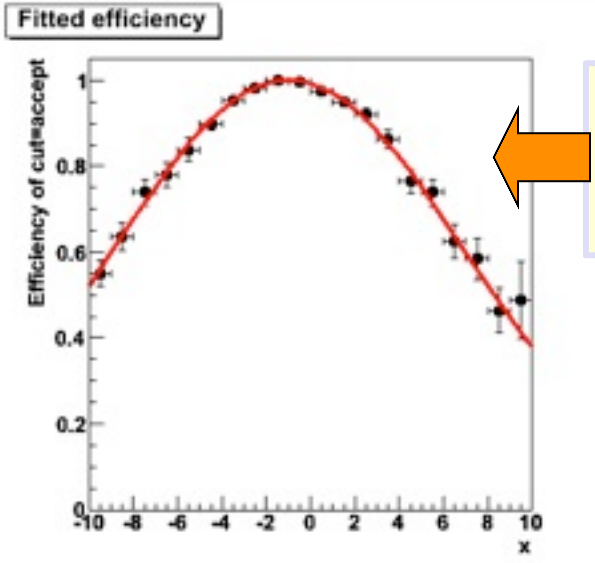
- Two morphing algorithms available
  - IntegralMorph (Alex Read algorithm).
    CPU intensive, but good with discontinuities
  - MomentMorph (Max Baak).
    Fast, can handling multiple observables (and soon multiple interpolation

Special pdfs – Unbinned ML fit for efficiency function

- Binomial pdf
  - Constructs pdf that can estimate *efficiency function e(x)* in from dataset D(x,c) where ‘c’ distinguishes accepted and rejected events

```
w.factory(“expr::e(‘(1-a)+a*cos((x-c)/b)’,x,a,b,c);
w.factory(“Efficiency::model(e,cut[acc,rej],"acc")”) ;

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


Data (all, accepted)


Fitted efficiency

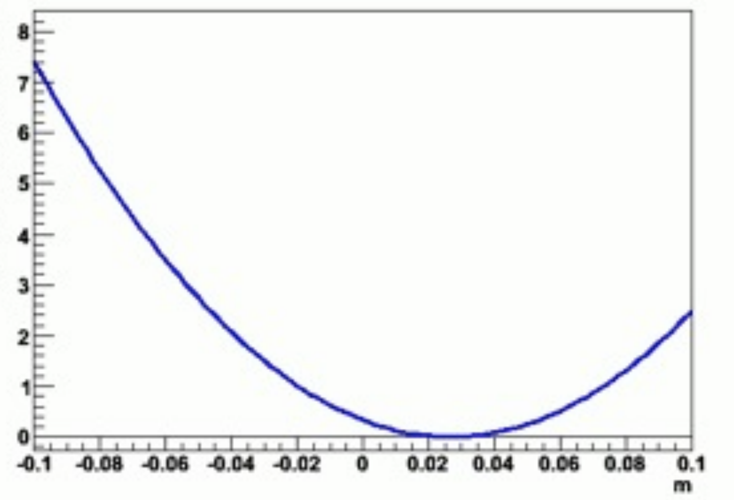
```
RooPlot* frame = w::x.frame() ;
data->plotOn(frame,
             Efficiency(cut)) ;
e.plotOn(frame) ;
```

4 Likelihood & Profile Likelihood

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)
  - NB: Also other minimizers (Minuit2, GSL etc) supported since 5.24

```
// 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)
```

- Can also create χ^2 functions objects

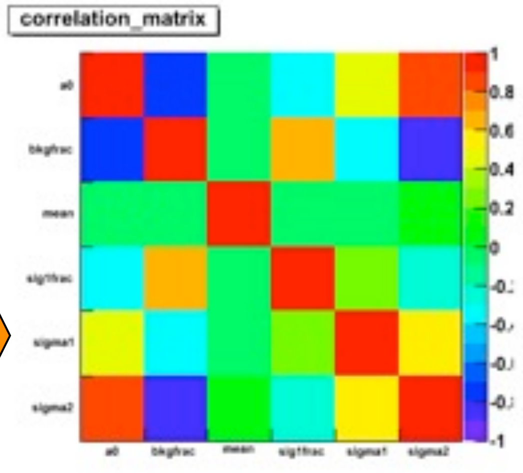
```
RooAbsReal* chi2 = w::model.createChi2(binnedData) ;
RooAbsReal* chi2 = w::model.createXYChi2(xyData) ;
```

Using the fit result output

- The fit result class contains the full MINUIT output

- Easy visualization of correlation matrix

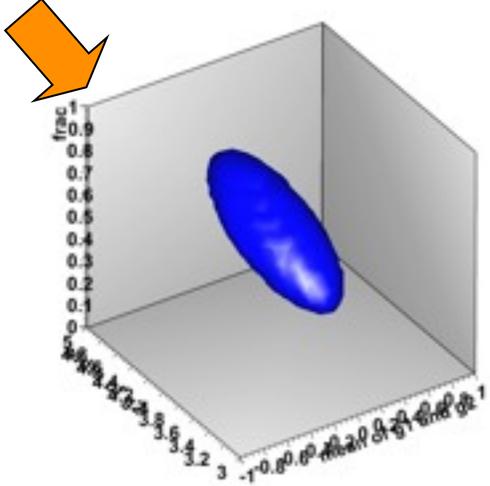
```
fitresult->correlationHist->Draw(“colz”) ;
```



- Construct multi-variate Gaussian pdf representing pdf on parameters

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

  - Returned pdf represents HESSE parabolic approximation of fit



- Extract correlation, covariance matrix

```
TMatrixDSym cov = fr->covarianceMatrix() ;
TMatrixDSym cov = fr->covarianceMatrix(a,b) ;
```

  - Can also retrieve partial matrix (Schur compl.)

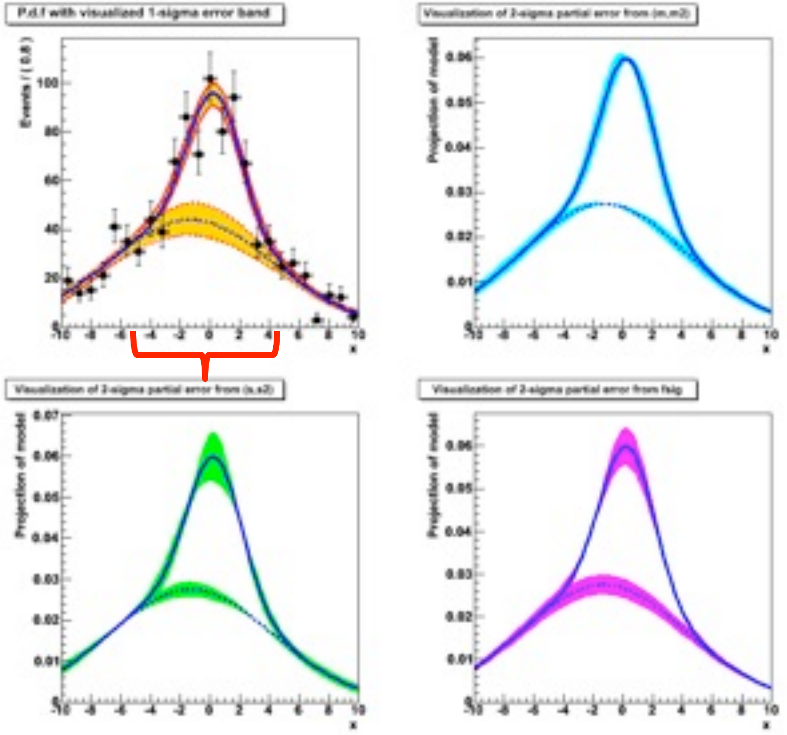
Using the fit result output – Error propagation

- 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(*fr);
```

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

Working with profile likelihood

- A profile likelihood ratio $$\lambda(p) = \frac{L(p,\hat{\hat{q}})}{L(\hat{p},\hat{q})}$$

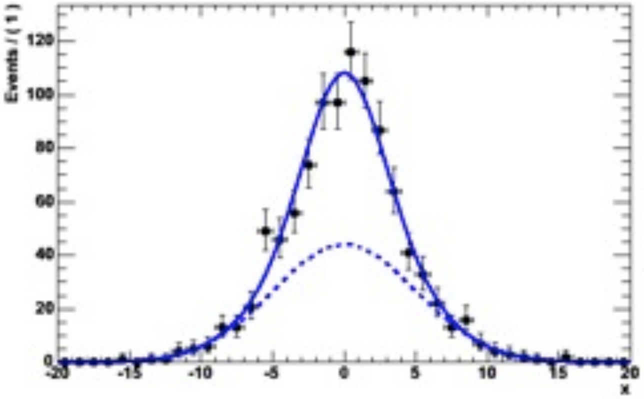
  ← **Best L for given p**

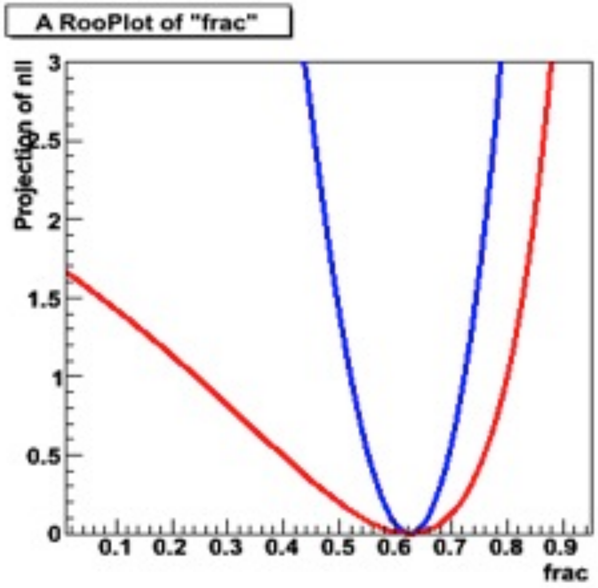
  ← **Best L**

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

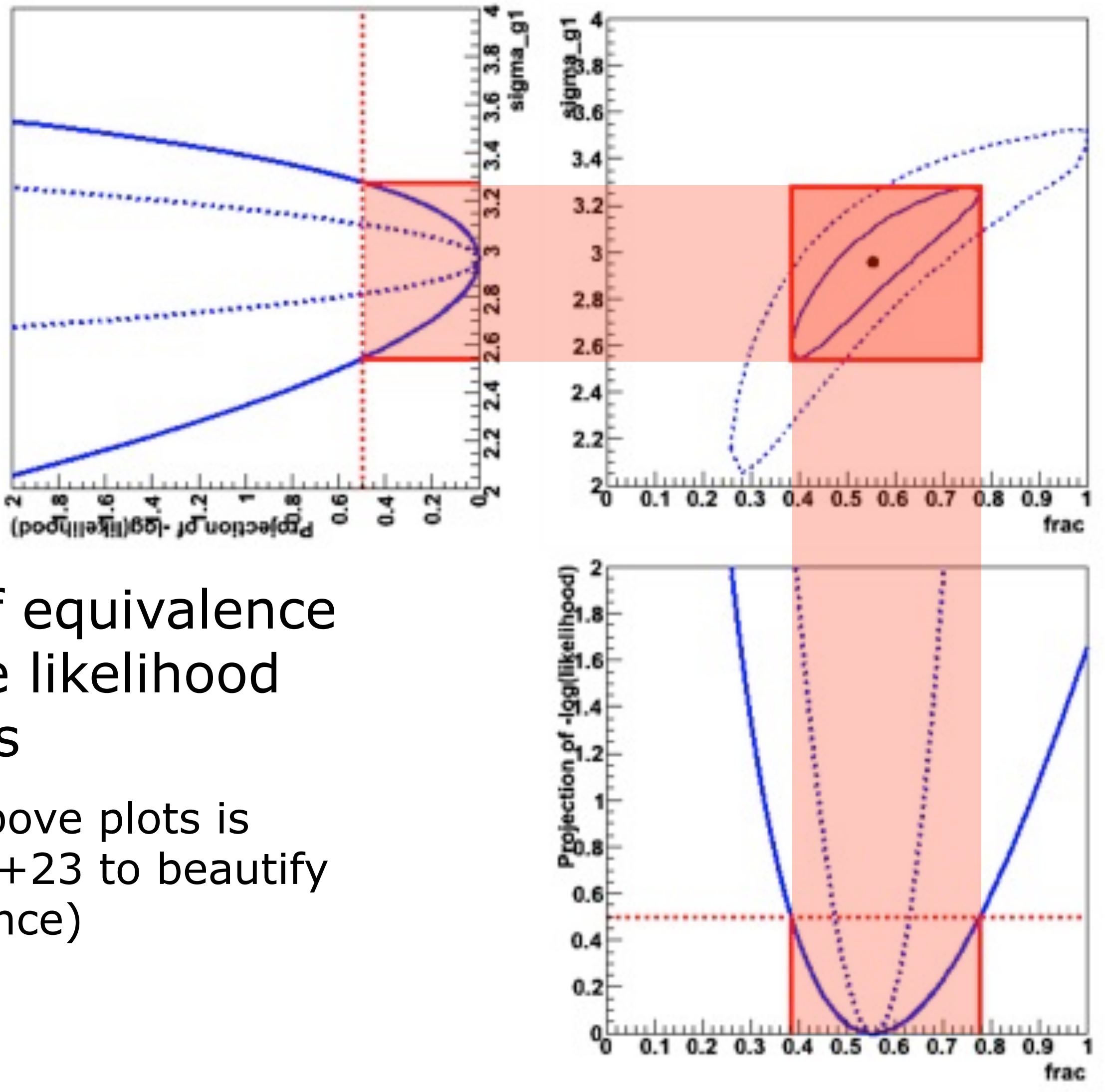
```
RooAbsReal* ll = model.createNLL(data,NumCPU(8)) ;
RooAbsReal* pll = ll->createProfile(params) ;
```

```
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)

5 Simultaneous fits and combinations

Constructing joint pdfs

- Operator class SIMUL to construct joint models at the pdf level

```
// Pdfs for channels ‘A’ and ‘B’
w.factory(“Gaussian::pdfA(x[-10,10],mean[-10,10],sigma[3])”) ;
w.factory(“Uniform::pdfB(x)”) ;

// Create discrete observable to label channels
w.factory(“index[A,B]”) ;

// Create joint pdf
w.factory(“SIMUL::joint(index,A=pdfA,B=pdfB)”) ;
```

- Can also construct joint datasets

```
RooDataSet *dataA, *dataB ;
RooDataSet dataAB(“dataAB”,”dataAB”,Index(w::index),
                  Import(“A”,*dataA),Import(“B”,*dataB)) ;
```

Constructing joint likelihood

- Can then construct the joint likelihood as usual

```
RooAbsReal* nllJoint = w::joint.createNLL(dataAB) ;
```

- Also possible to make likelihood first and then join

```
RooAbsReal* nllA = w::A.createNLL(*dataA) ; w.import(nllA) ;
RooAbsReal* nllB = w::B.createNLL(*dataB) ; w.import(nllB) ;
w.factory(sum::nllJoint(nllA,nllB)) ;
```

  - But then there is no definition of joint pdf
    and cannot execute frequentist techniques on joint models...

Using joint models

- When constructing joint models and likelihoods:

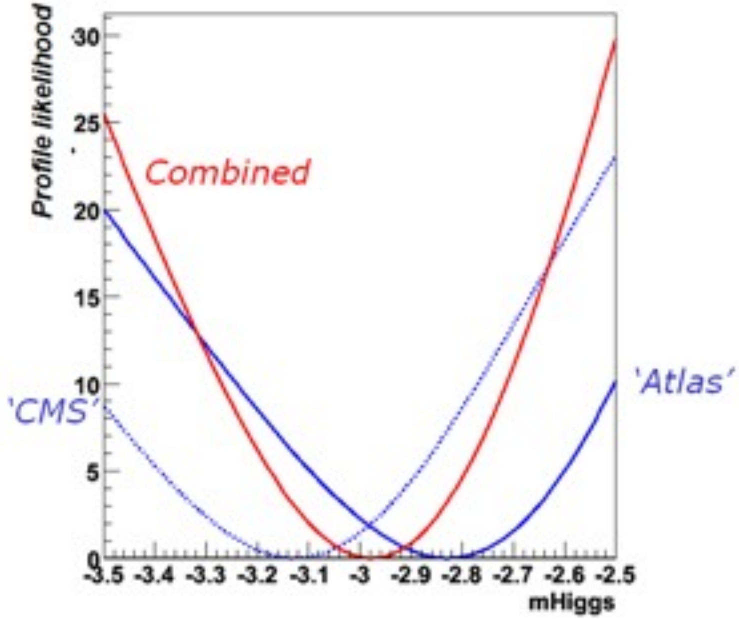
  parameters with the same name = same parameter

- If intentional, you are done at this point.

```
RooAbsReal* pllJoint = nllJoint->createProfile(paramOfInterest) ;
```

- Takes all parameter correlations fully into account
- To add additional correlations, simply multiply joint pdf with appropriate **`RooMultiVarGaussian`** pdf in parameters of choice

```
w.factory("MultiVarGaussian::corr
                ({a,b},{0,0},COV)");
w.factory("PROD::jointc(joint,corr)") ;
```



Tools to aid logistics of building a joint model

- Multiple experiments / analysis groups are unlikely to be organized to an extent where parameter naming schemes match exactly
  - The workspace has tools to manage this
  - These tools are the basis for (future) high level combination tools that will be part of the RooStats project
- Import model from another workspace
  - Example:: rename all variables of import model to unique names by appending a suffix **`_aHZZ`**, and rename mHiggs to **`MH`**

```
w.import(atlasHiggsZZ,
         RenameAllVariablesExcept(“mHiggs”,”aHZZ”),
         RenameVariable(“mHiggs”,”MH”) ;
```

  - Can also import straight from file using **`fileName:wspaceName:objName`** syntax

```
w.importFromFile(“ahzz.root:w:atlasHiggsZZ”,…) ;
```

Summary

- Brief overview of RooFit functionality, tailored to serve as introductory to RooStats
  - Many features were not mentioned here
  - No discussion of how this work internally (optimization, analytical deduction abilities)
  - About 90% of the details were omitted
- Documentation
  - Starting point: http://root.cern.ch/drupal/content /roofit
  - Users manual (134 pages ~ 1 year old)
  - Quick Start Guide (20 pages, recent)
  - Link to 84 tutorial macros (also in $ROOTSYS/tutorials/roofit)
- Support
  - Post your question on 'Stat & Math Forum' of ROOT (root.cern.ch → Forum → Stat & Math tools)
  - I aim for <24h response (but I don't manage every day!)

6 Hands-on exercises

Getting started – ROOT setup

- Start a ROOT 5.25/02 session
  - On lxplus (SLC4) or lx64slc5 (SLC5) choose appropriate line below

```
lxplus> source ~verkerke/public/setup_slc4.csh
lxplus> source ~verkerke/public/setup_slc4.sh
lxplus> source ~verkerke/public/setup_slc5.csh
lxplus> source ~verkerke/public/setup_slc5.sh
```

- Now move to your personal working area
- Load the roofit & roostats libraries

```
root> gSystem->Load(“libRooStats”) ;
```

- If you see a message that RooFit v3.10 is loaded you are (almost) ready to go.
- Import the namespace RooFit in CINT

```
root> using namespace RooFit ;
```

- Recommendation: put the last two lines in your ROOT login script to automate the loading
  - At least for the duration of the tutorial

Getting started – Online reference material

- RooFit class documentation (from code)
  - http://root.cern.ch/root/html/ROOFIT_ROOFITCORE_Index.html
  - http://root.cern.ch/root/html/ROOFIT_ROOFIT_Index.html
- RooFit home page at ROOT web site
  - http://root.cern.ch/drupal/content/roofit
  - Has links to manual and tutorial macros

Exercise 1 – A simple fit

- Copy ~verkerke/public/ex1.C and run it.
  - This macro uses the ‘w::’ shortcut syntax only available in CINT
  - Look at ex1var.C to see the solution written in pure C++
- This macro does the following for you:
  - Creates a workspace “w”, and uses the factory to fill it with a Gaussian g(x,mean,sigma)
  - Generates an unbinned dataset in x with 10K events from the pdf
  - Performs an unbinned ML fit of the pdf to the data
  - Makes a plot of the data with the pdf overlaid
  - Calls the Print() function on the parameter to see that the parameter estimate and its error have been propagated to the variable
- Modify the macro to generate a binned dataset instead of an unbinned dataset and run again
  - Use generateBinned() instead of generate()

Exercise 2 – Making a composite model

- Rename ex1.C to ex2.C

- Add a 2rd order Chebychev pdf to the workspace with coefficients a1=0 and a2=0.1 (each with range [-1,1])
  - See page 21 of presentation for help

- Using the SUM operator create a new pdf model that adds the Gaussian and the Chebychev.
  - Give each component a coefficient (e.g. Nsig and Nbkg) with a range [0,10000]
  - See page 27 of presentation for the syntax of SUM.
  - You can create Nsig and Nbkg in the same command as the SUM constructions following the logic explained on page 12 of the presentation

- Call the Print("t") method on the workspace to see the new contents

- Generate a dataset with 1000 events from model, fit it, and plot the data, model, as well as the background component of model
  - Use the Components() method to specify the background component.
  - If you like you can add LineStyle(kDashed) option
  - *If you get ROOT error messages that 'Components()' is not defined, you have forgotten your 'using namespace RooFit'*

Exercise 2 – Making a composite model (cont’d)

- This part is optional – do it only when you feel you are progressing quickly, otherwise do it when you have completed the other exercises

- Redo the fit, adding a Save() argument to fitTo() and save the returned RooFitResult* pointer
  - See page 17 of presentation for help

- Visualize the correlation matrix from the fit result
  - gStyle->SetPalette(1) ;
  - myFitResult->correlationHist()->Draw(“colz”) ;

- Plot the fitted pdf with the error band defined by the fit result
  - Add a VisualizeError(*myFR) option in RooAbsPdf::plotOn().
  - Do the same for the background component plot
  - NB: You can change the color of the band using e.g. FillColor(kYellow), and have the band placed at the bottom of the draw stack with the additional MoveToBack() command

Demo 1 – FFT convolution of arbitrary pdfs

- Copy ~verkerke/public/fftdemo.C and run it
- This macro demonstrates how the FCONV fourier convolution operator is used to convolute a Landau pdf with a Gaussian resolution model
- A binned likelihood fit of the numerically convoluted pdf with three floating parameters takes ~1 second

Exercise 3 – Persisting your model

- Copy ex2.C to ex3a.C
- At the end of the macro, import the toy data you generated into the workspace as follows
  - w.import(data,Rename("data")) ;
- Write your workspace to file
  - using the method w.writeToFile("model.root").
- Now quit your ROOT session
- Copy ~verkerke/public/ex3b.C.
  - This macro will read in your model.root file and plot the pdf and dataset contained in it
- Look at the macro and run it

Demo 2 – simultaneous fitting

- Copy ~verkerke/public/simfitdemo.C and run it
- This macro demonstrates techniques to make simultaneous fits to a ‘signal’ and ‘control’ samples in multiple ways

1. Plain fit to signal sample with sigPdf+BkgPdf
2. Plain fit to control sample with sigPdf+BkgPdfCtrl
3. Simultaneous fit to signal and control samples
4. Construct a pdf on sigPdf *parameters* from fit 2), multiply with pdf for signal sample.
   - Equivalent to 3) in the approximation of a parabolic likelihood for the control sample

Exercise 4 – Working with the likelihood

- Copy ex3b.C to ex4.C
- Remove the plotting code and add a line to create a function object that represents the –log(likelihood)
  - Use method RooAbsPdf::createNLL(RooAbsData&), the returned object is of type RooAbsReal*
  - See page 41 in the presentation for help
- Minimize the likelihood function 'by hand' by passing it to a RooMinuit object and calling its methods migrad() and hesse()
  - See page page 42 in the presentation for help (also for below)
  - Now call the minos() function only for parameter Nsig.
  - Call w::Nsig.Print() afterwards to see that the asymmetric error has been propagated
  - Fix the width of the Gaussian and run minos again and observe the effect. (use w::sigma.setConstant(kTRUE))
- Make a plot of –log(L) vs Nsig
  - First create a plot frame in the parameter using RooPlot* frame = w::Nsig.frame() ;
  - Now plot the likelihood function on the frame, using plotOn() as usual
  - If you like you can add a ShiftToZero() argument to the plotOn() call and see what that does
  - You can adjust the virtual range of the plot frame with SetMinimum() and

Demo 3 – n-Dim models and likelihood ratio plot

- Copy ~verkerke/public/llrplot.C and run it
- This macro builds a 3-dimensional model
  - Flat background in (x,y,z)
  - Gaussian signal in (x,y,z) with correlations
- It plots three 2D projections (x,y), (x,z) and (y,z)
- Then it makes three varieties of 1D plots of model and data
  - Plain projection on x (shows lots of background)
  - Projection on x in a 'signal box' in (y,z)
  - Projection on x with a cut on the LR(y,z)>68%, where LR(y,z) is defined as

$$LR(y,z) = \frac{\int f_{sig} \times S(x,y,z)dx}{\int S(x,y,z) + B(x,y,z)dx}$$

(i.e. the signal probability according to the model using the (y,z) observables only)

Exercise 5 – Profile likelihood

- Copy ~verkerke/public/ex4.C (standard solution to ex4) to ex5.C
- Adjust the *horizontal* plot range of the likelihood plot so that it just covers the interval ΔLL=+25 units
  - Make a new plot frame that zooms in on that range and plot the likelihood again (you can use myparam.frame(pmin,pmax) to control the plot range)
- Create the *profile* likelihood function in Nsig
  - Call createProfile(w::Nsig) on the likelihood and save the returned pointer to the profile likelihood function (of type RooAbsReal*)
  - Plot the profile likelihood ratio on the Nsig frame too (make it red by adding a LineColor(kRed))
- Find the profile likelihood ratio interval of Nsig : find the points at which the PLR rises by +0.5 units
  - Compare the interval to that of the MINOS error of exercise Ex 4.

Exercise 6 – Parallelizing the likelihood calculation

- Check the number of CPU cores available on the current host (‘cat /proc/cpuinfo’)
- Modify the createNLL() call of ex5 to take an extra NumCPU(N) argument
  - The likelihood calculation will now be parallelized over N cores
- Rerun ex5 and observe the difference in wall-time execution speed.
  - The speedup is best demonstrated on an empty worker node (your best is lx64slc5)