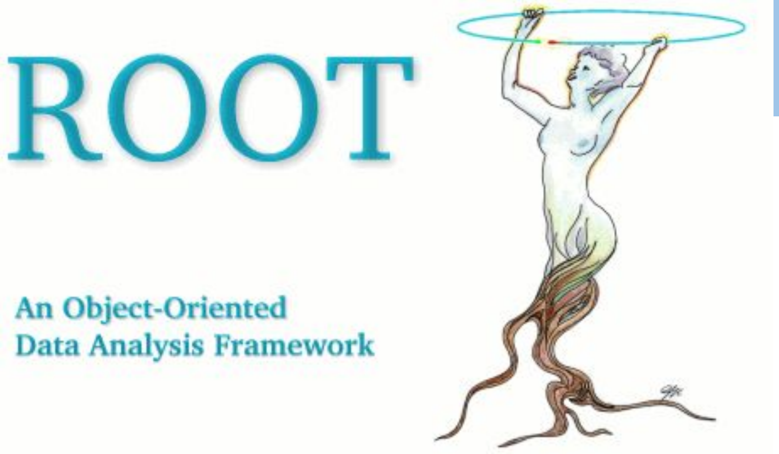
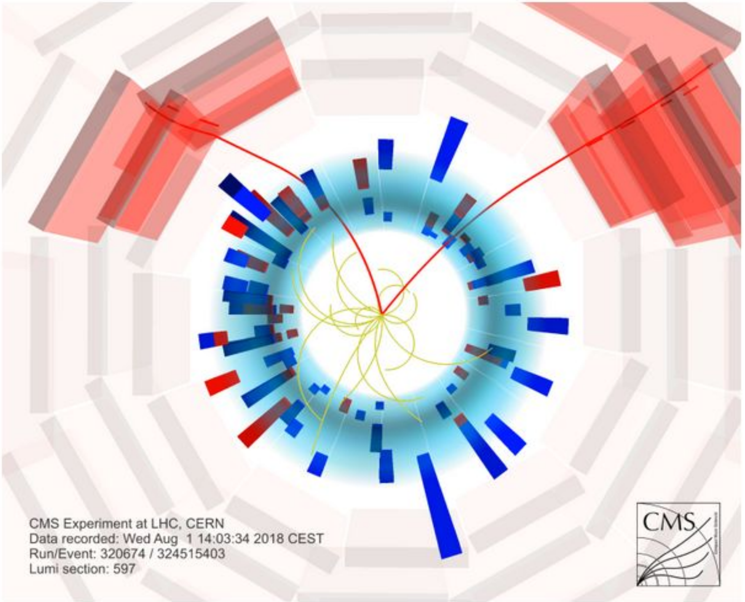
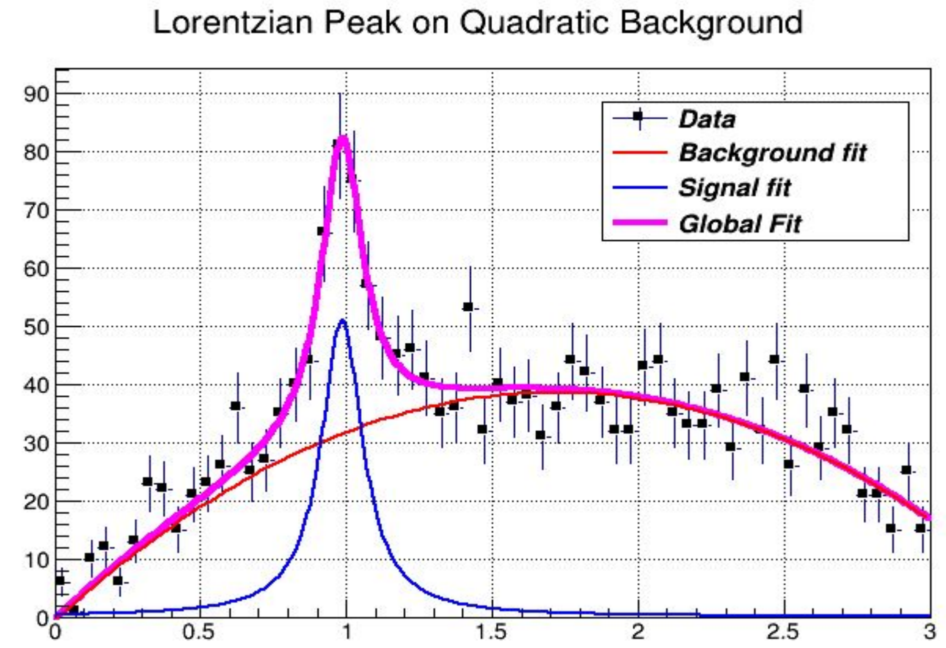






Big data en el CERN y otros contextos

Jhovanny Andres Mejia Guisao
UNIVERSIDAD DE ANTIOQUIA, COLOMBIA

Links interesantes

https://cms-analysis.github.io/HiggsAnalysis-CombinedLimit/
https://github.com/cms-analysis/HiggsAnalysis-CombinedLimit

Motivation

What we hope to discuss about scientific data analysis?

- Advanced graphical user interface
- Interpreter for the C++ programming language
- Persistency mechanism for C++ objects
- Used to write every year petabytes of data recorded by the Large Hadron Collider experiments

Input and plotting of data from measurements and fitting of analytical functions.

Credit

RooFit slides and example are extracted from material prepared by W. Verkerke (NIKHEF), author of RooFit

– more information and additional slides from W. Verkerke are available at

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

Roofit manual

https://root.cern/download/doc/RooFit_Users_Manual_2.91-33.pdf

Purpose

The RooFit library provides a toolkit for modeling the expected distribution of events in a physics analysis. Models can be used to perform unbinned maximum likelihood fits, produce plots, and generate "toy Monte Carlo" samples for various studies. RooFit was originally developed for the BaBar collaboration, a particle physics experiment at the Stanford Linear Accelerator Center. The software is primarily designed as a particle physics data analysis tool, but its general nature and open architecture make it useful for other types of data analysis also.

Mathematical model

The core functionality of RooFit is to enable the modeling of ‘event data’ distributions, where each event is a discrete occurrence in time, and has one or more measured observables associated with it. Experiments of this nature result in datasets obeying Poisson (or binomial) statistics. The natural modeling language for such distributions are probability density functions F(x;p) that describe the probability density the distribution of observables x in terms of function in parameter p.

The defining properties of probability density functions, unit normalization with respect to all observables and positive definiteness, also provide important benefits for the design of a structured modeling language: p.d.f.s are easily added with intuitive interpretation of fraction coefficients, they allow construction of higher dimensional p.d.f.s out of lower dimensional building block with an
intuitive language to introduce and describe correlations between observables, they allow the universal implementation of toy Monte Carlo sampling techniques, and are of course an prerequisite for the use of (unbinned) maximum likelihood parameter estimation technique.

Roofit manual

https://root.cern/download/doc/RooFit_Users_Manual_2.91-33.pdf

Design

RooFit introduces a granular structure in its mapping of mathematical data models components to C++ objects: rather than aiming at a monolithic entity that describes a data model, each math symbol is presented by a separate object. A feature of this design philosophy is that all RooFit models always consist of multiple objects. For example a Gaussian probability density function consists typically of four objects, three objects representing the observable, the mean and the sigma parameters, and one object representing a Gaussian probability density function. Similarly, model building operations such as addition, multiplication, integration are represented by separate operator objects and make the modeling language scale well to models of arbitrary complexity.

Math concept	*Math symbol*	*RooFit (base)class*
Variable	x	`RooRealVar`
Function	$f(x)$	`RooAbsReal`
P.D.F.	$F(x;p)$	`RooAbsPdf`
Integral	$\int_{x_{min}}^{x_{max}} f(x)dx$	`RooRealIntegral`
Space point	$\vec{x}$	`RooArgSet`
Addition	$fF(x)+(1-f)G(x)$	`RooAddPdf`
Convolution	$f(x)\otimes g(x)$	`RooFFTConvPdf`

Table 1 - Correspondence between selected math concepts and RooFit classes

Scope

RooFit is strictly a data modeling language: It implements classes that represent variables, (probability density) functions, and operators to compose higher level functions, such as a class to construct a likelihood out of a dataset and a probability density function. All classes are instrumented to be fully functional: fitting, plotting and toy event generation works the same way for every p.d.f., regardless of its complexity

RooFit

- ➔ is a library which provides a toolkit for data analysis
- ➔ is included in ROOT framework
- ➔ is used to model expected event distributions in physics analysis
- ➔ can perform (un)binned maximum likelihood fits, produce plots and study goodness-of-fit with toy Monte Carlo samples
- ➔ was originally developed for the BaBar collaboration @ Stanford Linear Accelerator Center

To use RooFit in ROOT CINT

Load library as:

- gSystem->Load("libRooFit") ;
  using namespace RooFit ;

OR

- Load prepared macro file
  .x path-to-file

Introduction – Purpose

Model the distribution of observables $\vec{x}$ in terms of

- Physical parameters of interest $\vec{p}$
- Other parameters $\vec{q}$ to describe detector effects (resolution,efficiency,...)

RooFit

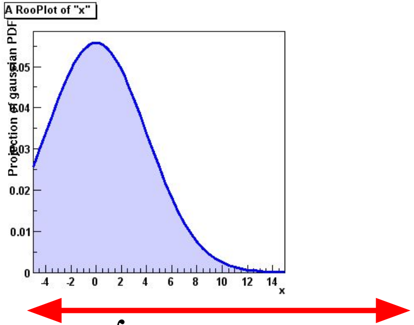
Probability density function $\mathbf{F(\vec{x};\vec{p},\vec{q})}$

- normalized over allowed range of the observables **x** w.r.t the parameters **p** and **q**

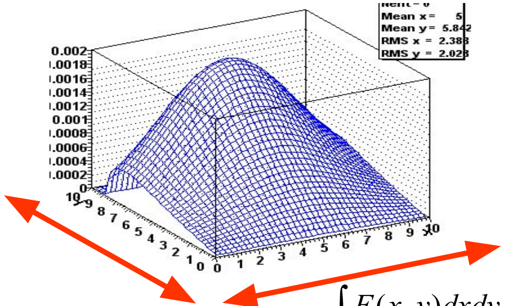
Mathematic – Probability density functions

- 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

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



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



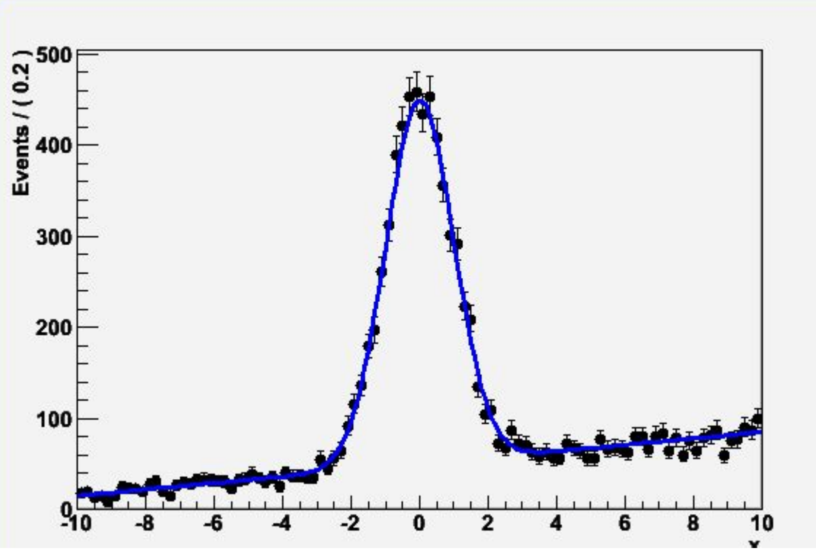
$$\int F(x,y)\,dx\,dy \equiv 1$$

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

Coding a probability density function

How do you formulate your p.d.f. in ROOT

For ‘simple’ problems (gauss, polynomial), ROOT built-in models well sufficient

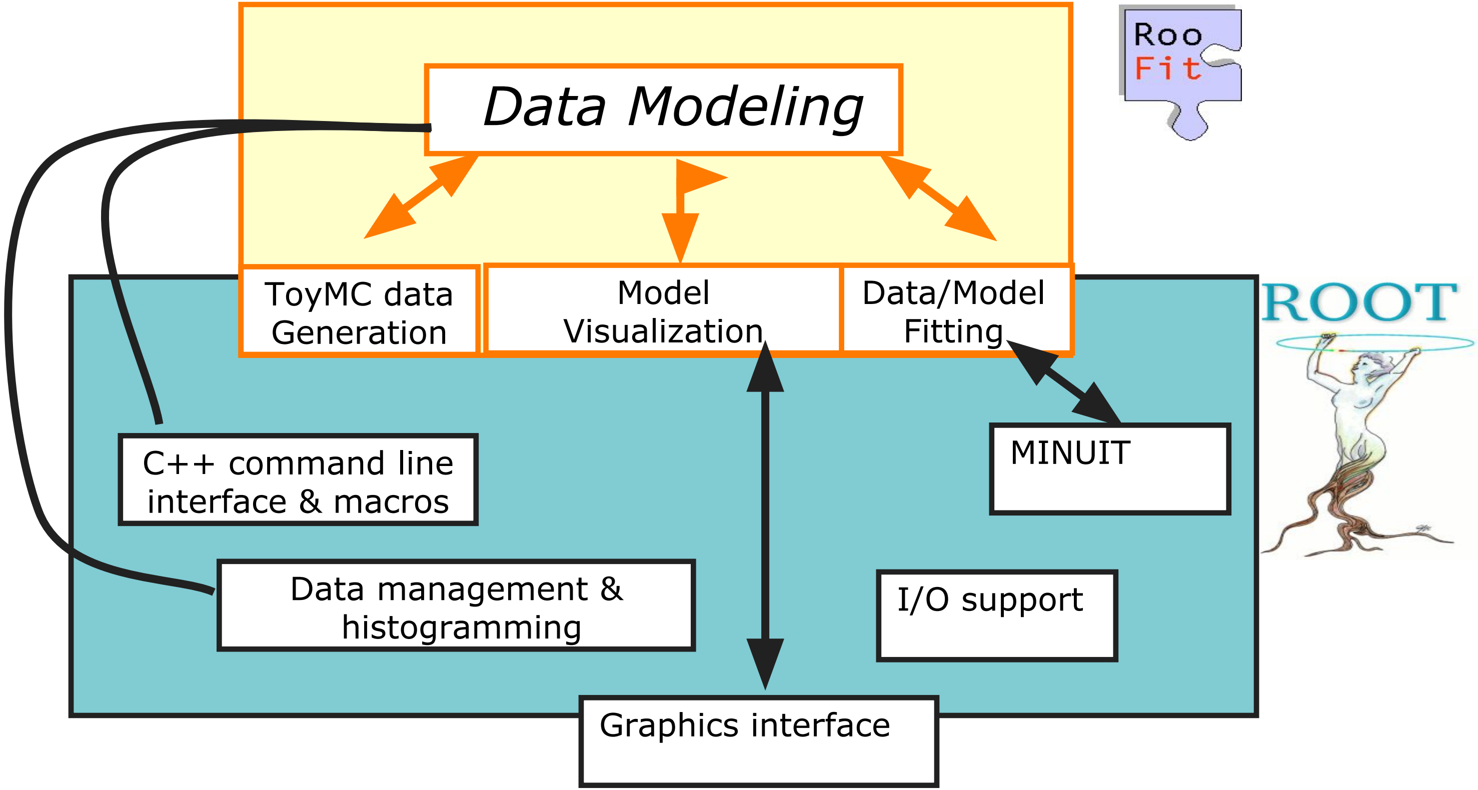


But if we want to do complex likelihood fits using non-trivial functions and composing several p.d.f., or to work with multidimensional functions it is difficult to do it in ROOT

- we need some tools to help us !

Introduction – Relation to ROOT

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



Introduction – Why RooFit was developed

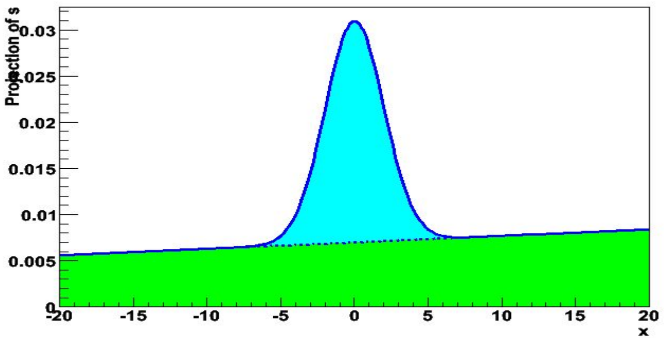
- BaBar experiment at SLAC: Extract sin(2β) from time dependent CP violation of B decay: e^+e^- □ Y(4s) □ BB
  - Reconstruct both Bs, measure decay time difference
  - Physics of interest is in decay time dependent oscillation

$$f_{sig} \cdot \left[\mathrm{SigSel}(m; \bar{p}_{sig}) \cdot \left(\mathrm{SigDecay}(t; \vec{q}_{sig}, \sin(2\beta)) \otimes \mathrm{SigResol}(t \mid dt; \vec{r}_{sig}) \right) \right] +$$
$$(1 - f_{sig}) \left[\mathrm{BkgSel}(m; \bar{p}_{bkg}) \cdot \left(\mathrm{BkgDecay}(t; \vec{q}_{bkg}) \otimes \mathrm{BkgResol}(t \mid dt; \vec{r}_{bkg}) \right) \right]$$

- 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

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

RooFit Modeling

- 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`

Simple example of complete maximum likelihood fit

```
RooRealVar x("x", "x", -10, 10);
RooRealVar mean("mean", "mean of gaussian", 1, -10, 10);
RooRealVar sigma("sigma", "width of gaussian", 1, 0.1, 10);

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

RooDataSet *data = gauss.generate(x, 10000);

gauss.fitTo(*data);

RooPlot *xframe2 = x.frame();
data->plotOn(xframe2);
gauss.plotOn(xframe2);
xframe2->Draw();
```

1. define 3 variables:
- observable x
- free parameters mean, sigma

2. create PDF model with these variables

3. generate 10^4 toy events

4. fit PDF and all floating parameters to data

5. plot data and PDF

Example1_myBasic

[rf101_basics.C]

1. Defining variables

- variables are defined as:

  RooRealVar(“name”, “title”, value, minValue, maxValue, “unit”)

  construct with either a fixed value / or a range / or starting value + range

- observables (i.e. x, y, energy, time) and parameters of a PDF (i.e. mean, sigma, slope) are both variables
  - the data set “tells” a PDF what it's observable is
  - all other variables must be parameters
- when fitting a PDF model to data: all free floating (= not fixed) parameters are fitted
- you can later on define and exclude a parameter from being fitted by the method

  RooRealVar.setValue(value) and RooRealVar.setConstant()

- construct flexible variable:

  RooFormulaVar mean_shifted(“mean_shifted”,”@0+@1”,RooArgList(mean,shift))

  ROOT TFormula expression RooRealVar's

2. About PDFs

➔ **construction of PDF is one of the most important steps**

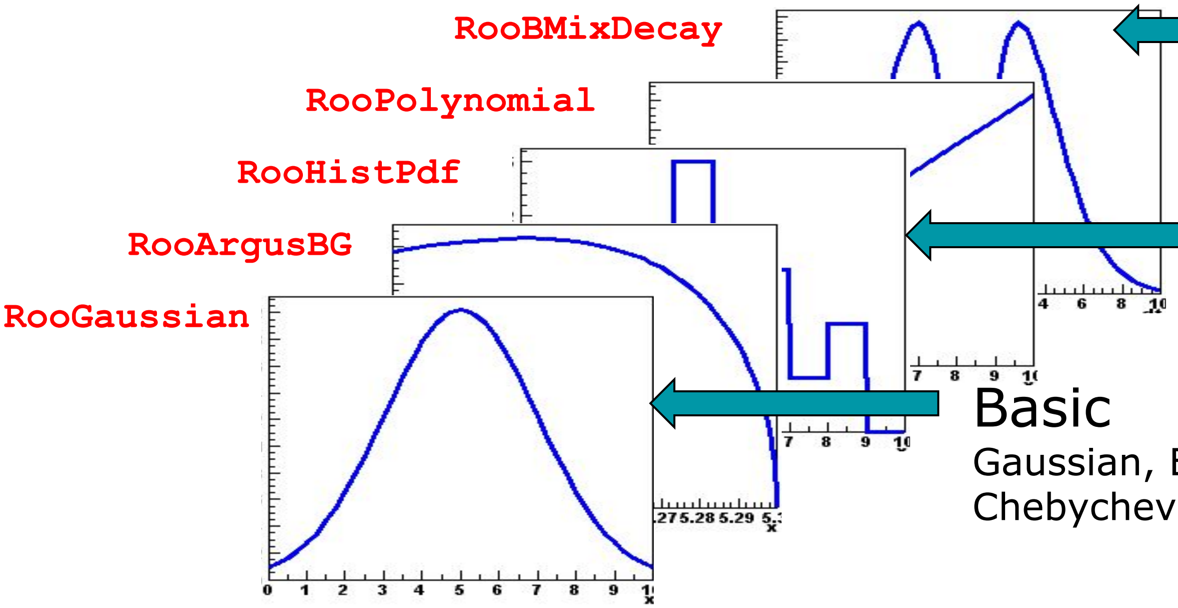
➔ **bad PDF → bad fit**

➔ **the PDF contains the parameters which are fitted:**
this can either be parameters defining the shape of a PDF (like decay constant, Gaussian width, …) or often fractions of different PDF components (i.e. signal vs. background component)

➔ **PDFs are automatically normalized within RooFit**

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

Build in PDFs

~20 predefined PDFs to build models from

Basic functions:

- RooGaussian: normal Gaussian
- RooBifurGauss: different width on low and high side of mean
- RooExponential: standard exponential decay
- RooPolynomial: standard polynoms
- RooChebychev: Chebychev polynomials (recommended because of higher fit stability due to little correlation)
- RooPoisson: Poisson distribution

Physics inspired functions:

- Landau (RooLandau), Breit-Wigner, Crystal Ball, …

Specialized functions for B physics:

- Decay distributions with mixing, CP violation, …

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,y,a,b in files `MyPdf.h`, `MyPdf.cxx`

```
RooClassFactory::makePdf("MyPdf","x,y,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

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));
```

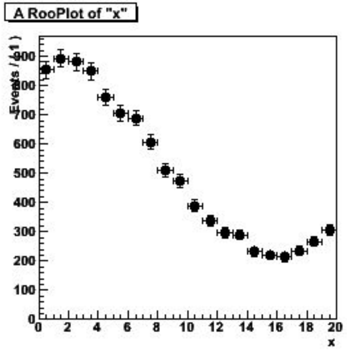
Interpreted code

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

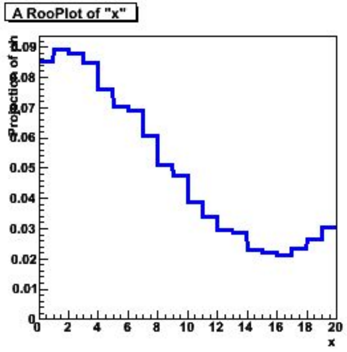
Highlight of non-parametric shapes - histograms

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

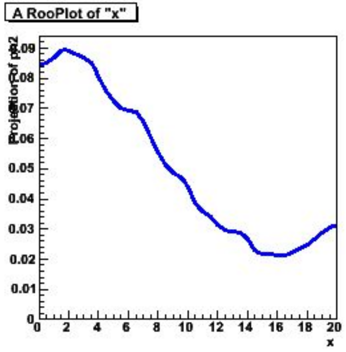
dataHist



RooHistPdf(N=0)



RooHistPdf(N=4)




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

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

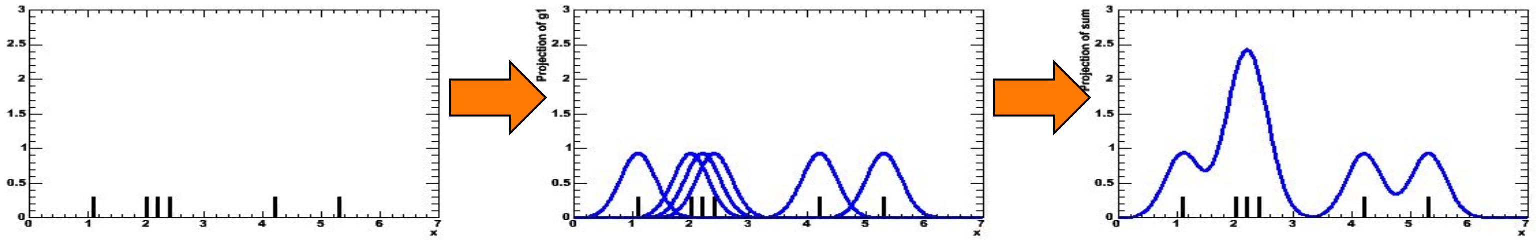
Highlight of non-parametric shapes – kernel estimation

- 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

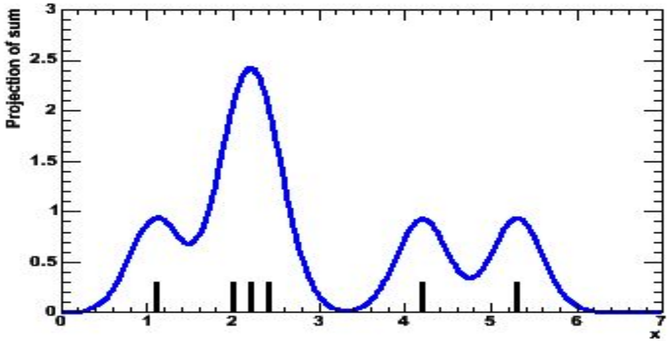
Summed probability distribution for all events in sample



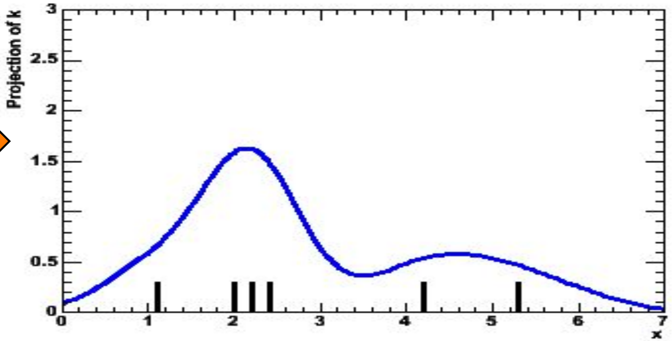
Highlight of non-parametric shapes – kernel estimation

- 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)

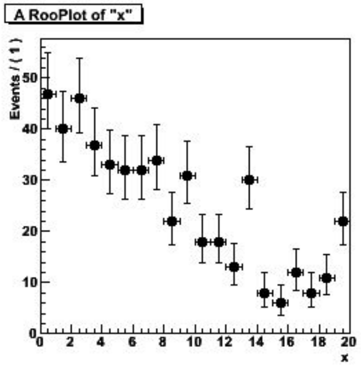


Highlight of non-parametric shapes – kernel estimation

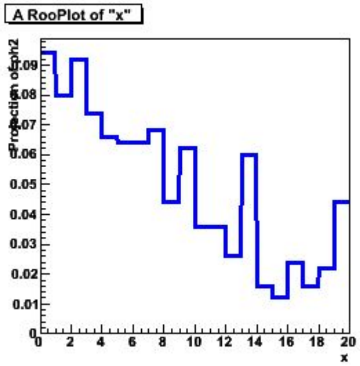
- Example with comparison to histogram based p.d.f
  - Superior performance at low statistics
  - Can mirror input data over boundaries to reduce 'edge leakage'
  - Works also in >1 dimensions (class `RooNDKeysPdf`)

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

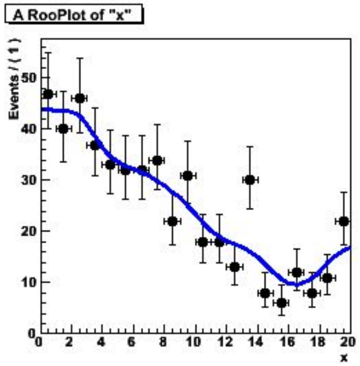
Data(N=500)



RooHistPdf(data)

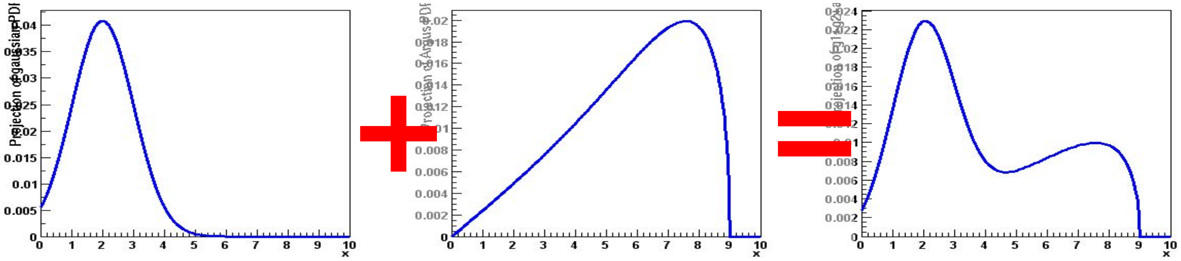


RooKeysPdf(data)

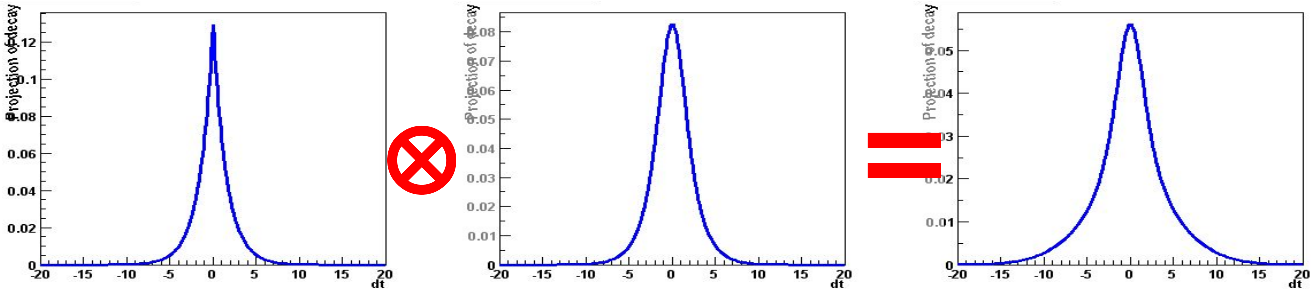


PDF addition and convolution: Building realistic models

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

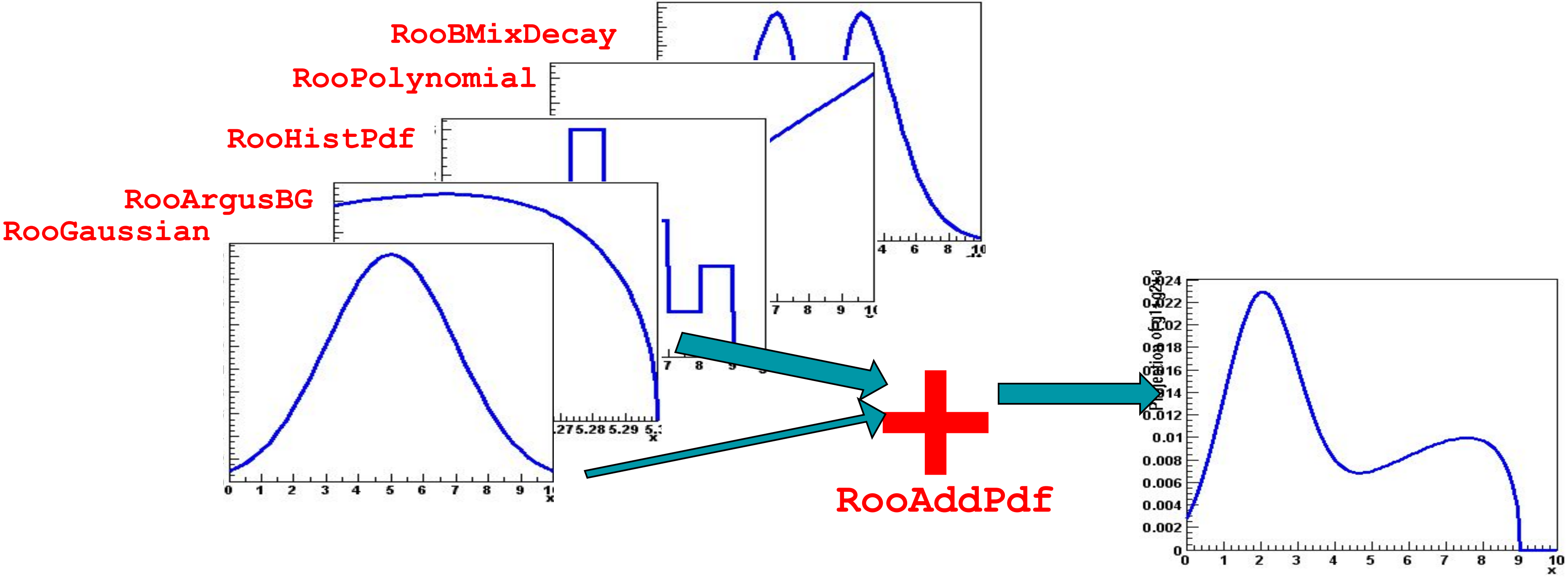


  - Convolution



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 oefficients:

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

Example2_addPDF

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

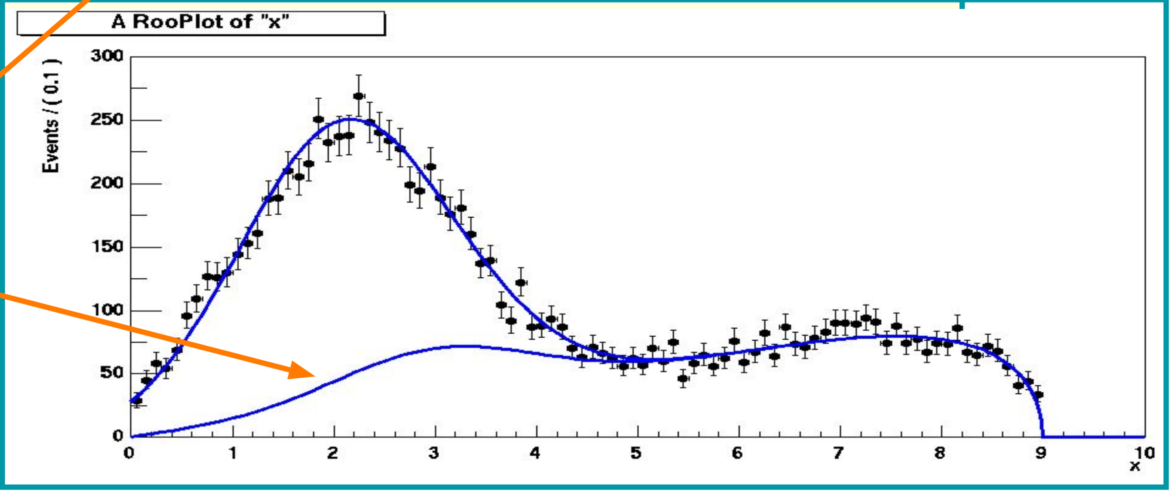
Plotting a sum of p.d.f.s, and its components

```
// 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() ;
```

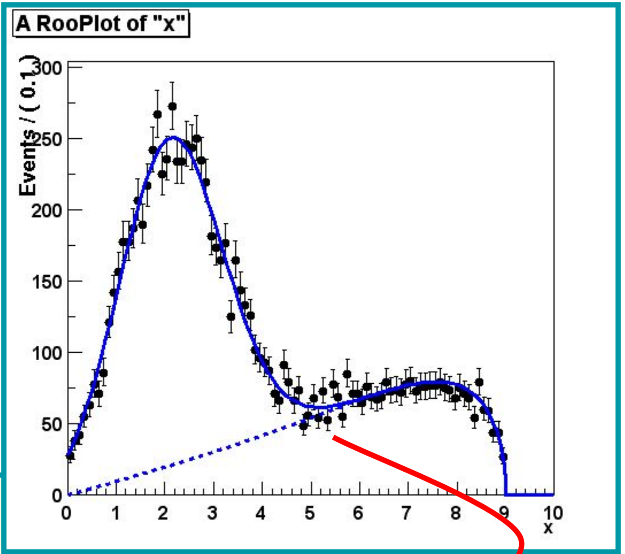
Plot selected components of a **RooAddPdf**



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")) ;
```

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

$$S_2(x) = f_1 P_1(x) + (1 - f_1) P_2(x)$$

$$S_3(x) = f_1 P_1(x) + (1 - f_1)\left(f_2 P_2(x) + (1 - f_2) P_3(x)\right)$$

$$S_4(x) = f_1 P_1(x) + (1 - f_1)\left(f_2 P_2(x) + (1 - f_2)\left(f_3 P_3(x) + (1 - f_3) P_4(x)\right)\right)$$

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

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 $-\log(\text{Poisson}(N_{obs}, N_{exp}))$ 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) ;
```

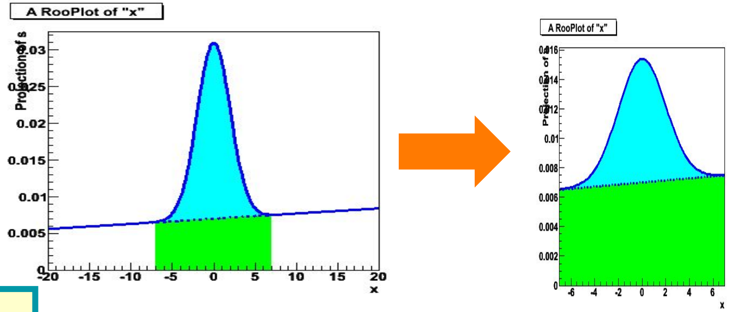
How it works – Normalization of `RooAddPdfs`

- 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

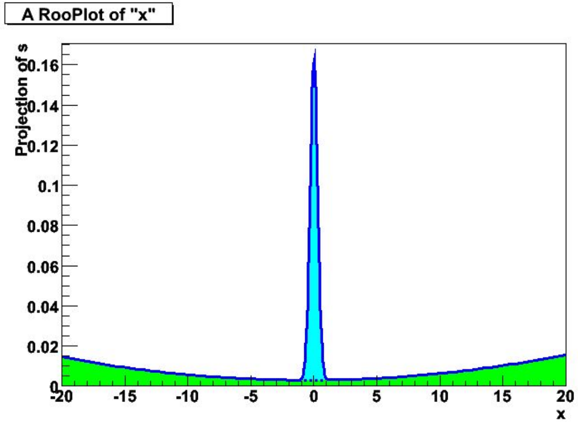
- Interpretation of fraction depends on range of observables (and number of observables for >1D)
  - If range of observable is changed and fraction parameter is same, the shape effectively different
  - Can mitigate this by specifying a fixed reference range for fraction interpretation




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

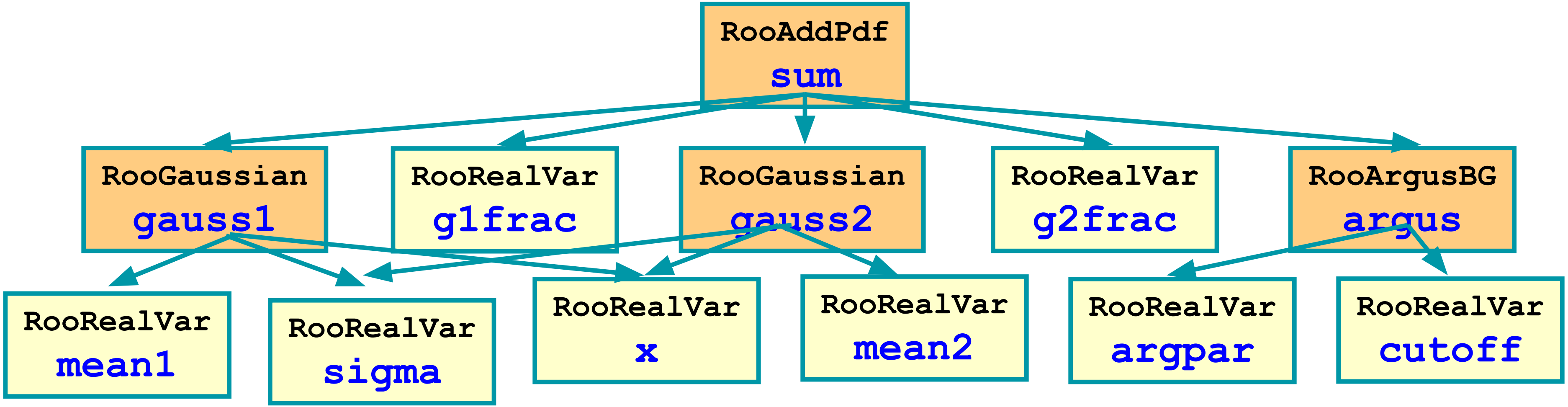
How it works – event generation of RooAddPdf

- Composite event generation algoritm 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

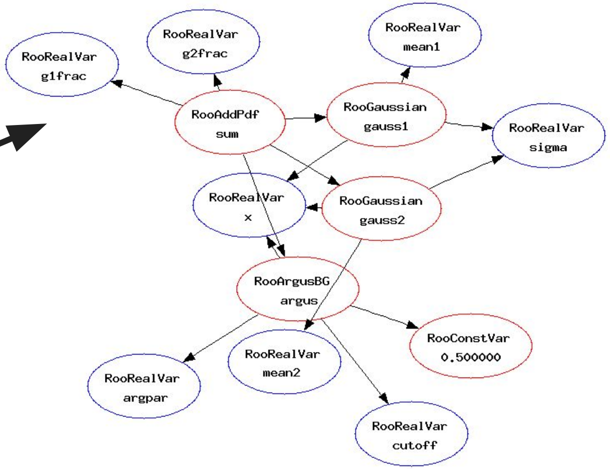
- Special tools exist to visualize the tree structure of composite objects
  - On the command line

```
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
```

Visualization tools for composite objects

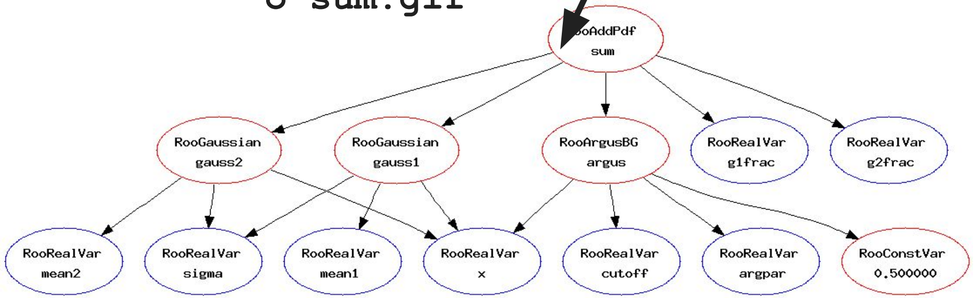
- Interface to GraphViz graph visualization tool

'fdp -Tgif sum.dot -o sum.gif'




```
Root> sum.graphVizTree("sum.dot")
```

'dot -Tgif sum.dot -o sum.gif'



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

Example3_Extended

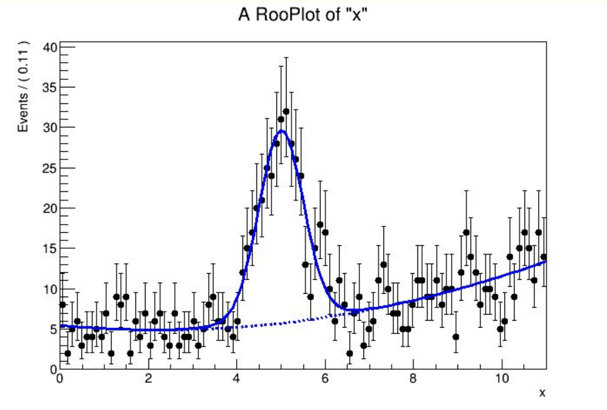
```
// Declare observable x
RooRealVar x("x","x",0,10) ;
// Creation of 'sig', 'bkg' component p.d.f.s omitted for clarity

// 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() ;
```



3. Datasets

- ➔ class RooDataSet is an N-dimension collection of points with continuous RooRealVar or discrete RooCategory observables and optional weights
- ➔ for all testing purposes: method **generate(observable,#events)** works on all PDFs (including composite, product, convoluted, ...)
- ➔ internally stored as unbinned or binned data in a ROOT TTree object
- ➔ importing unbinned data
  - ◆ from ASCII files (values in tab seperated columns)

    ```
    RooRealVar x("x","x",-10,10) ;
    RooRealVar c("c","c",0,30) ;
    RooDataSet::read("ascii.txt",RooArgList(x,c)) ;
    ```
  - ◆ from ROOT TTrees

    ```
    RooDataSet data("data","data",inputTree,RooArgSet(x,c));
    ```
- ➔ importing binned data from ROOT THx histograms

  ```
  RooDataHist bdata2("bdata","bdata",RooArgList(x,y),histo2d);
  ```
- ➔ manual filling with **dataset.add(RooArgSet(x,c))**

only values which are in observable range are imported

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

*(Internal ROOT **`TTree`**)*

x	Y	wgt
1.0	6.6	1
3.5	11.1	1
2.7	2.2	1
5.2	1.1	1

Current coordinate return by **`RooAbsData::get()`**
Current weight returned by **`RooAbsData::weight()`**

`RooArgSet`

`RooRealVar` x | **`RooRealVar`** y

`Double_t` wgt

Move current row with **`RooAbsData::get(index)`**

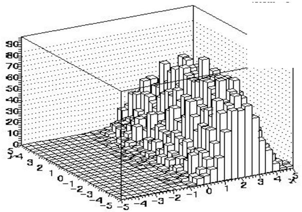
Binned data, or unbinned data (with optional weights)

- 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



RooDataSet

RooDataHist

RooAbsData

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

A ROOT TTree with the bin coordinates

- 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() ;
```

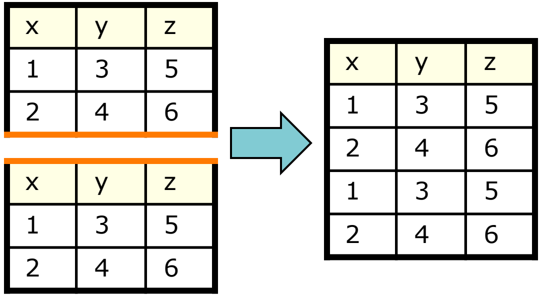
Example4_Createddataset
Example4_ImporOurtree

Extending and reducing **unbinned** datasets

- Appending

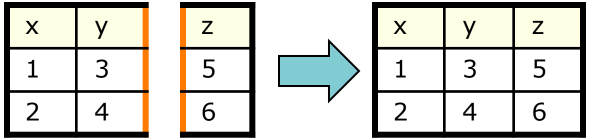
```
RooDataSet d1("d1","d1",RooArgSet(x,y,z));
RooDataSet d2("d2","d2",RooArgSet(x,y,z));

d1.append(d2) ;
```



- Merging

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

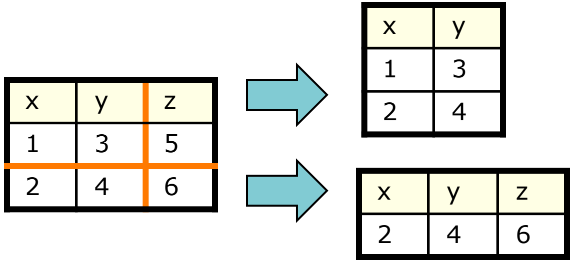


- Reducing

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

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

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



4. Fitting and accessing of results

- ➔ 2 different ways of fitting a PDF model to data
  - ◆ automatic mode on a given pdf

    pdf.fitTo(*data)

  - ◆ manual mode:

    // Construct function object representing –log(L)

    RooNLLVar nll(“nll”,”nll”,pdf,data) ;

  - ◆ //Minimize nll w.r.t its parameters

    RooMinuit m(nll) ;
    m.migrad() ; // find min NLL
    m.hesse() ; // symmetric errors assuming parabola
    m.minos() ; // asymmetric errors from min NLL +0.5

- ➔ both methods accept fit-options (Extended-mode, # of CPU-Cores, fit range,etc)
- ➔ fitting is performed via interface with ROOT MINUIT package
- ➔ Ioption “r” saves result in RooFitResults object
- ➔ further possibilities:
  - ◆ profile likelihood with class RooProfileLL
  - ◆ exporting likelihood function + PDF + data in Workspace object

Exemplary fit output

fit values and errors

progress information

```
**********
**    6 **MIGRAD        2500           1
**********
FIRST CALL TO USER FUNCTION AT NEW START POINT, WITH IFLAG=4.
START MIGRAD MINIMIZATION.  STRATEGY  1.  CONVERGENCE WHEN EDM .LT. 1.00e-03
FCN=-3661.65 FROM MIGRAD    STATUS=INITIATE        16 CALLS          17 TOTAL
                    EDM= unknown      STRATEGY= 1      NO ERROR MATRIX
 EXT PARAMETER               CURRENT GUESS       STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  a0           5.00000e-01   1.00000e-01   2.01358e-01  -2.65254e+00
  2  a1           2.00000e-01   1.00000e-01   2.57889e-01  -6.14686e+00
  3  nbkg         7.00000e+02   2.00000e+02   2.11716e-01  -7.60710e+00
  4  nsig         3.00000e+02   1.50000e+02   2.16811e-01   1.32870e+01
  5  sig1frac     8.00000e-01   1.00000e-01   2.57889e-01  -2.29354e+00
                               ERR DEF= 0.5
MIGRAD MINIMIZATION HAS CONVERGED.
MIGRAD WILL VERIFY CONVERGENCE AND ERROR MATRIX.
COVARIANCE MATRIX CALCULATED SUCCESSFULLY
FCN=-3662.44 FROM MIGRAD    STATUS=CONVERGED      129 CALLS         130 TOTAL
                    EDM=4.92835e-07    STRATEGY= 1      ERROR MATRIX ACCURATE
```

status, distance to minimum (EDM)

min NLL

```
EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  5    ERR DEF=0.5
 3.862e-03  1.512e-03 -5.916e-01  5.917e-01 -3.443e-03
 1.512e-03  8.521e-03 -2.345e+00  2.346e+00 -1.273e-02
-5.916e-01 -2.345e+00  1.704e+03 -9.851e+02  5.545e+00
 5.917e-01  2.346e+00 -9.851e+02  1.266e+03 -5.545e+00
-3.443e-03 -1.273e-02  5.545e+00 -5.545e+00  4.076e-02
PARAMETER  CORRELATION COEFFICIENTS
      NO.  GLOBAL      1      2      3      4      5
       1  0.29868   1.000  0.263 -0.231  0.268 -0.274
       2  0.75559   0.263  1.000 -0.615  0.714 -0.683
       3  0.72212  -0.231 -0.615  1.000 -0.671  0.665
       4  0.82576   0.268  0.714 -0.671  1.000 -0.772
       5  0.81060  -0.274 -0.683  0.665 -0.772  1.000
```

error & correlation matrix

5. Plotting

- ➔ first: create empty RooPlot frame for an observable (i.e. “x”)
- ➔ an unbinned dataset is automatically shown as binned histogram when drawn on the frame with data->plotOn()
  - ◆ customizeable with Binning(int nbins, double xlo, double xhi)
  - ◆ Markerstyle/color/width etc can of course be changed too
- ➔ PDF drawn with pdf.plotOn()
  - ◆ gets automatically normalized to data set
  - ◆ lgets automatically projected over all other observables if necessary
- ➔ RooPlot-frames can hold any other ROOT drawable objects (arrows, text boxes, …): i.e. xframe.addObject(TArrow)
- ➔ useful information about PDF an data:
  - ◆ pdf.paramOn(xframe,data) ;
  - ◆ data.statOn(xframe) ;
- ➔ further possibilities: plot small slice or larger range of a data set and a PDF
- ➔ for >1D PDFs & data: createHistogram() method gives a ROOT TH2/TH3

rf106_plotdecoration.C
rf107_plotstyles.C

goodness-of-fit test

How do you know if your fit was good?

rf109_chi2residpull.C

- for 1-D fit:
  - calculate χ^2/d.o.f. of a curve w.r.t. data:
    frame->chiSquare()
  - make pull and residual histogram:
    frame->makePullHist() ;
    frame->makeResidHist() ;

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

Example5_pull

2. Testing the Goodness-of-fit for > 1-D: toy Monte Carlo study using class RooMCstudy
rf801_mcstudy.C