0 安装和设置roofit

0.1 安装root和roofit

roofit库是标准root发行版的一部分，并且预编译在二进制发行版中，可以通过 `root.cern.ch` 获取。如果您从源代码发行版编译 ROOT，则必须在运行 configure 时使用标志 --enable-roofit

0.2 设置交互式root环境

只要在命令行引用了 RooFit 类之一，ROOT 就会自动加载 RooFit 库 libRooFitCore 和 libRooFit。方便起见，建议你ROOT登录脚本中添加下面的命令 `using namespace RooFit;` ，这样你就能在命令行中使用RooFit命名空间中的辅助函数了。这个命令也会自动加载RooFit库。这篇指导的所有例子都是建立在你已经加载RooFit命名空间的基础上的。

> *" using namespace RooFit; "*

0.3 用ROOT与RooFit设置已编译的应用

要用ROOT与RooFit设置一个独立的已编译应用程序，请使用标准ROOT推荐做法，但要在链接命令行中加入RooFit、RooFitCore 和 Minuit 库

> *"export CFLAGS= `root-config —cflags`*
> *export LDFLAGS=`root-config —ldflags –glibs` -lRooFit –lRooFitCore -lMinuit*
> *g++ ${CFLAGS} -c MyApp.cxx*
> *g++ -o MyApp MyApp.o ${LDFLAGS} "*

1 动手做

这部分会带你做简单的模型去拟合数据。目的是让你熟悉一些基本概念，并且能快速地做一些有用的事情。在接下来的小节中我们会更仔细地探索RooFit几个方面。

1.1 构建模型

RooFit中一个关键概念是以面向对象的方式构建模型的。每个RooFit的类都有一个对应的数学对象： `RooRealVar` 表示一个变量， `RooAbsReal` 表示一个函数， `RooAbsPdf` 表示一个概率密度函数（probability density function），等等。就算是最简单的数学函数也包含了很多对象——函数本身与其变量——RooFit模型也因此由许多对象组成。
下面这个例子可以说明：

```
1 RooRealVar x("x","x",-10,10) ;
2 RooRealVar mean("mean","Mean of Gaussian",0,-10,10);
3 RooRealVar sigma("sigma","Width of Gaussian",3,-10,10);
4 RooGaussian gauss("gauss","gauss(x,mean,sigma)",x,mean,sigma);
```

example - 构建一个高斯分布的概率密度函数

Note:对于一个函数来说，需要自变量x和参数，对于自变量来说你需要设定一个范围，也就是函数的定义域，而对于参数来说，因为你需要用这个函数去拟合一些数据，那么就不可能一开始就确定这个函数，而是给他一个大致的范围让他去拟合。

在 `gauss` 用到的每个变量都用几个属性进行初始化：name，title，范围和可选的初始值。用 `RooRealVar` 描述的变量有更多的属性在这个例子中没有展现出来，例如与变量关联的对称误差以及指定变量是常量还是浮动的标志。本质上，类 `RooRealVar` 收集所有与变量有关的属性。

代码的最后一行创建了一个高斯分布的概率密度函数，由 `RooGaussian` 实现。
类 `RooGaussian` 描述了所有概率密度函数的共同属性。PDF 高斯有一个名称和一个标题，就像变量对象一样，并通过构造函数中传递的引用关联到变量 x、mean 和 sigma。

1.2 模型可视化

通常我们最先想到的是看到这个模型。RooFit采取了比普通ROOT稍微正式一点的可视化方法。首先你需要定义一个“视图”，本质上是一个空的框架，包含 RooRealVar 中变量之一x作为横坐标轴，然后你要把你的模型gauss放进这个空白框架中，最后你再把这个视图画在ROOT TCanvas上：

```
1 RooPlot* xframe = x.frame() ;  //创建一个以x这个变量为坐标的空坐标图，
  这个会直接获取你之前设置的x变量的范围定义你的坐标的范围，这里就是 -10～10
2 gauss.plotOn(xframe) ;       //将你之前创建的gauss函数画到你的坐标图中
3 xframe->Draw();              //将你这个图通过TCanvas“打印”出来
```


figure 1 高斯pdf

结果看图1。需要注意的是，在定义或创建一个视图的时候并不需要特意去设置范围，会根据 RooRealVar 中定义的变量的范围自动设置，当然你也可以重新设置范围。另外，将gauss画在框架中时，也不用说明gauss是x的函数，可以在框架中找到这个信息，就很智能。

一个框架可以可视化多个对象（曲线，直方图），我们可以画两个不同sigma值的高斯曲线

```
1 RooPlot* xframe = x.frame() ;
2 gauss.plotOn(xframe) ;
3 sigma = 2 ;
4 gauss.plotOn(xframe,LineColor(kRed)) ;
5 xframe->Draw();
```

这个例子中，在画完第一个gauss图像之后，我用赋值运算符改变了 RooRealVar sigma的值。第二条曲线的颜色通过添加 LineColor(kRed) 参数传递给plotOn()变为红色。LineColor 是“命名参数”的一个示例。命名参数贯穿整个 RooFit 的使用，它提供一种方便且可读的方式来修改一些默认值。命名参数将在后面的部分

中更详细地介绍。第二个代码片段的输出如图 2 所示。

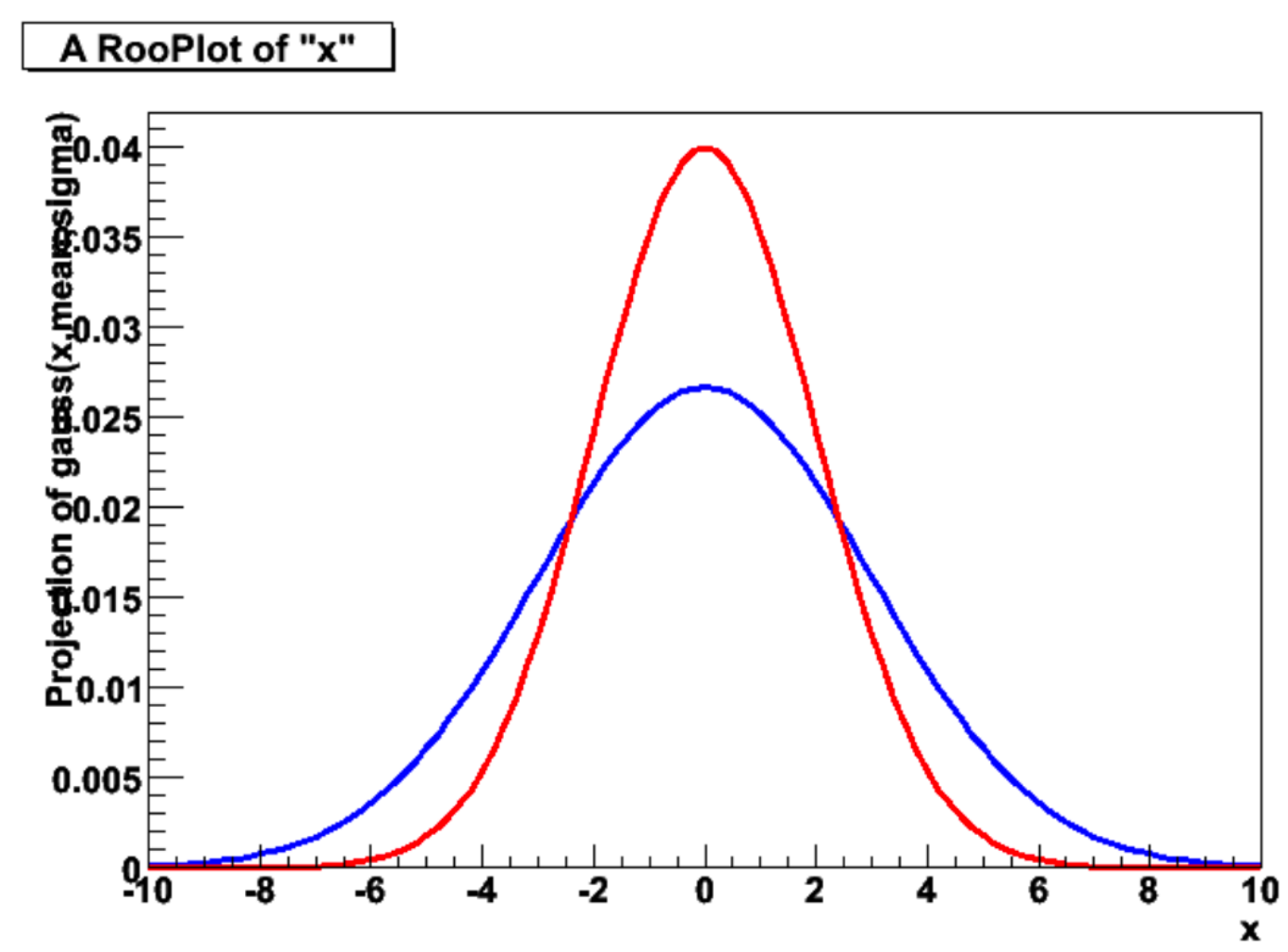


figure 2 不同宽度的高斯pdf

这个例子同时也展示了plotOn()方法会对PDF进行一个“冻结”的快照，如果一个pdf在画进去之后改变了形状，之前已经画进去的曲线不会发生改变。图2还展示了无论数值如何， `RooGaussian` 总是归一化的。

1.3 输入数据

通常来说，数据有两类：未分bin的，用ROOT中的TTree类表示，和分bin的，用ROOT中的TH1，TH2，TH3表示。RooFit可以处理这两种数据。

1.3.1 分bin的数据（直方图）

在RooFit中，分bin的数据由 RooDataHist 类表示，你可以将任何ROOT直方图的内容导入到 RooDataHist 对象中

```
TH1* hh = (TH1*) gDirectory->Get("ahisto");
RooRealVar x("x","x",-10,10);
RooDataHist data("data","dataset with x",x,hh);
```

example 2 - 从TTree中导入数据并将其画在TCanvas上

note：代码从当前目录 (gDirectory) 中获取名为 "ahisto" 的直方图，并将其强制转换为 TH1* 类型的指针。在ROOT中，TH1 是表示一维直方图的类。

创建了一个名为 "data" 的 RooDataHist 对象。它表示一个数据集，可以在RooFit中用于拟合或其他统计分析。
数据集用名称 "data" 和标题 "dataset with x" 构建。
此数据集根据变量 "x"（我们之前定义的变量）进行分箱，并使用先前获得的直方图 hh 填充。这意味着 hh 中的值将用于填充数据集的箱子。

当你导入ROOT直方图的时候，原直方图的bin是如何定义的同样也被导入了。 RooDataHist 比普通直方图更进一步的是，它将直方图的内容和一个或多个 RooRealVar 类型的RooFit变量关联起来。通过这种方式，它总知道直方图中存储了什么样的数据。

RooDataHist 可以和gauss函数一样被plotOn在画布上。

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

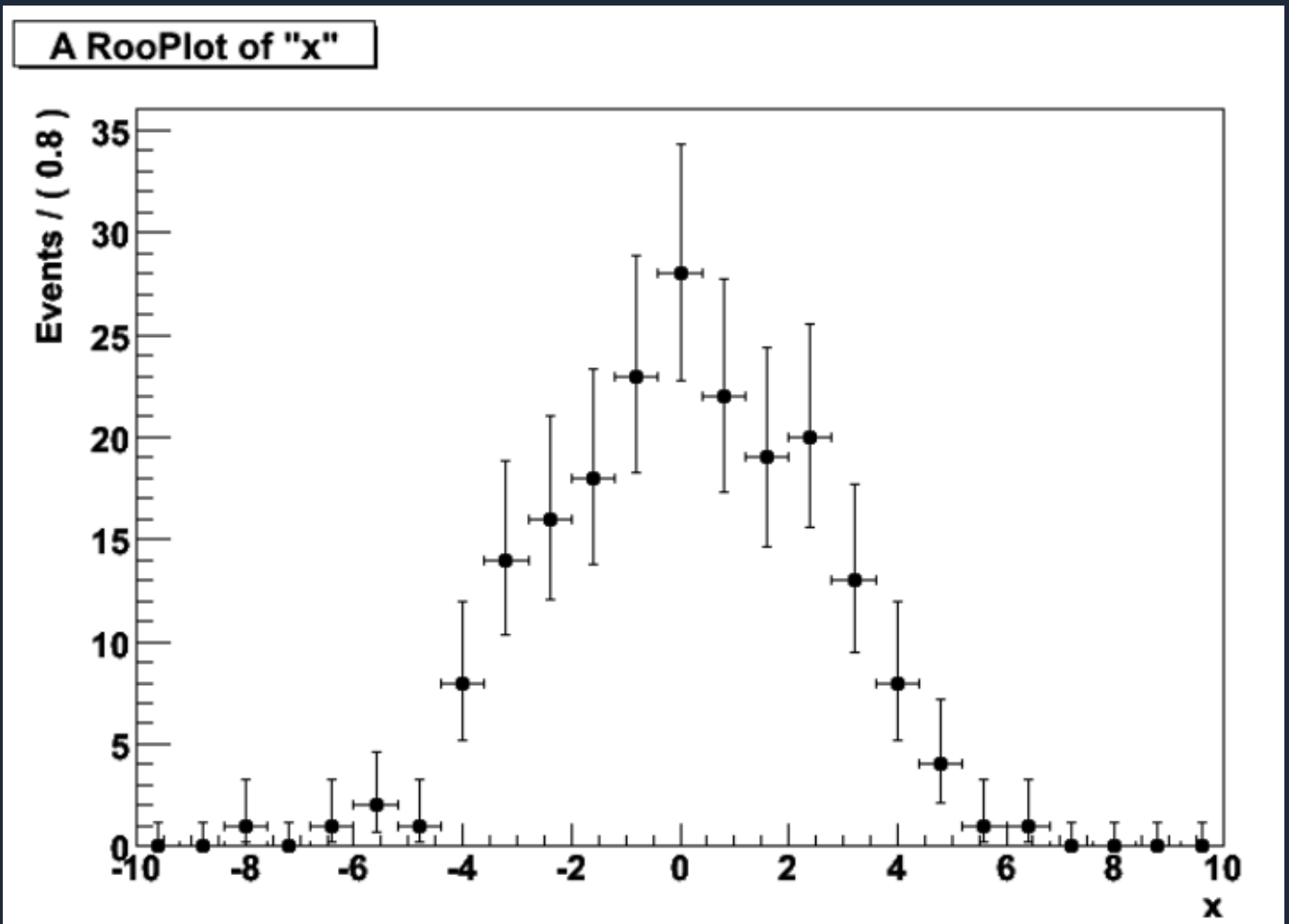


Figure 3 – Histogram visualized in RooFit

如果你仔细看图3你会发现，在低统计下的误差棒不是对称的。RooFit 默认显示泊松统计的 68% 置信区间，通常是不对称的，尤其是在低统计量下，如果直方图内容来自泊松过程，则可以更准确地反映每个箱中的统计不确定性。你可以选择将 `DataError(RooAbsData::SumW2)` 添加到 `data.plotOn()` 行来显示$\sqrt{N}$误差。这个选项只会影响数据集的可视化。

1.3.2 未分bin的数据（树trees）

未分bin的数据可以以同样的方式导入到RooFit，并存储到 `RooDataSet` 的类中

```
TTree* tree = (TTree*) gDirectory->Get("atree");
RooRealVar x("x","x",-10,10);
RooDataSet data("data","dataset with x",tree,x);
```

在这个例子中，假设 tree 有一个名为“x”的分支，因为 RooDataSet 构造函数将从与作为参数传递的 RooRealVar 同名的树分支中导入数据。RooDataSet 可以从类型为 Double_t、Float_t、Int_t、UInt_t 和 Bool_t 的分支导入数据，用于一个 RooRealVar 可观测量。如果分支不是 Double_t 类型，则数据将转换为 Double_t，因为这是 RooRealVar 的内部表示形式。无法从数组分支（如 Double_t[10]）导入数据。可以将整数类型的数据作为离散值观测量导入到 RooFit 中，这在第8章中会更详细地解释。

绘制未分 bin 的数据与绘制分 bin 的数据类似，不同之处在于现在您可以以任何您喜欢的 binning 来显示它，默认情况下是100个bins，下面设置了25个bins的情况。

```
RooPlot* xframe = x.frame();
data.plotOn(frame,Binning(25));
frame->Draw();
```

1.3.3 working with *data*

在一般情况下，在RooFit中处理分组数据和未分组数据非常相似，因为RooDataSet（用于未分组数据）和RooDataHist（用于分组数据）都继承自一个共同的基类RooAbsData，该基类定义了一个通用的抽象数据样本接口。除少数例外，所有RooFit方法都接受抽象数据集作为输入参数，使得分组和未分组数据可以互换使用。本节中的示例始终处理一维数据集。然而，RooDataSet和RooDataHist都可以处理任意维数的数据。在接下来的章节中，我们将重新讨论数据集并解释如何处理多维数据。

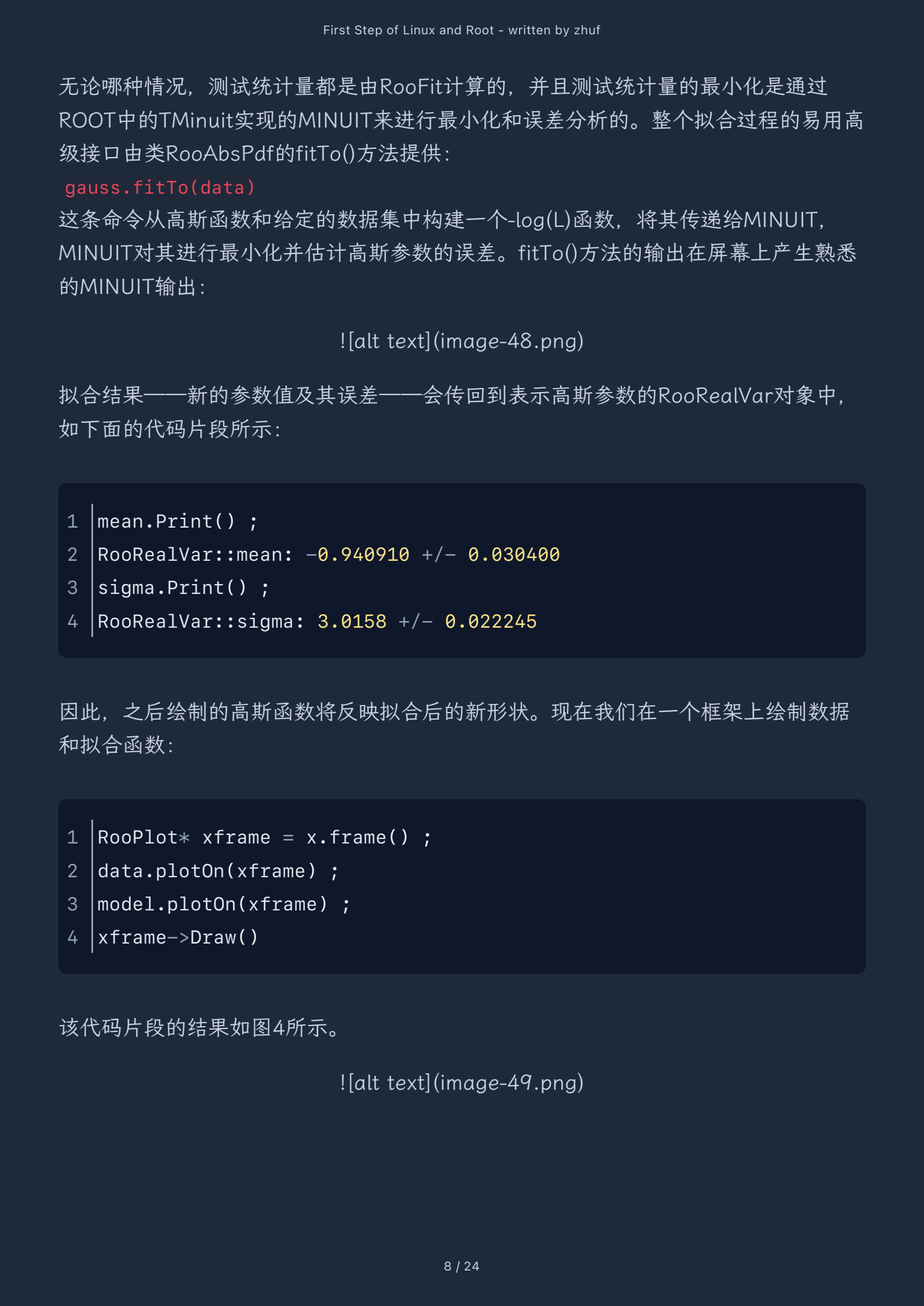
1.4 用模型拟合数据

将模型拟合到数据中涉及从模型和数据构建检验统计量——最常见的选择是χ^2和负对数似然——并针对所有未固定的参数最小化该检验统计量。RooFit中的默认拟合方法是对无分箱数据进行无分箱最大似然拟合，对分箱数据进行分箱最大似然拟合。

无论哪种情况，测试统计量都是由RooFit计算的，并且测试统计量的最小化是通过ROOT中的TMinuit实现的MINUIT来进行最小化和误差分析的。整个拟合过程的易用高级接口由类RooAbsPdf的fitTo()方法提供：

`gauss.fitTo(data)`

这条命令从高斯函数和给定的数据集中构建一个-log(L)函数，将其传递给MINUIT，MINUIT对其进行最小化并估计高斯参数的误差。fitTo()方法的输出在屏幕上产生熟悉的MINUIT输出：



拟合结果——新的参数值及其误差——会传回到表示高斯参数的RooRealVar对象中，如下面的代码片段所示：

```
mean.Print() ;
RooRealVar::mean: -0.940910 +/- 0.030400
sigma.Print() ;
RooRealVar::sigma: 3.0158 +/- 0.022245
```

因此，之后绘制的高斯函数将反映拟合后的新形状。现在我们在一个框架上绘制数据和拟合函数：

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

该代码片段的结果如图4所示。

请注意，PDF 的归一化（本质上归一化为一）会自动调整为图中事件的数量。RooFit 的一个强大功能和其创建的主要原因之一是示例3中的拟合调用对分组数据和未分组数据都有效。在后一种情况下，会执行未分组的最大似然拟合。未分组的 -log(L) 拟合在统计上比分组拟合更有力（即，您将获得更小的均值误差），并且避免了由选择分组定义引入的任意性。这些优势在拟合小数据集和多维数据集时尤为明显。

拟合接口高度可定制。例如，如果您想在拟合中固定一个参数，只需将其指定为 RooRealVar 参数对象的一个属性，那么这个代码片段：

```
mean.setConstant(kTRUE) ;
gauss.fitTo(data) ;
```

将参数 `mean` 固定为当前值重复拟合。同样，您可以选择将浮动参数限定在允许值范围内：

```
sigma.setRange(0.1, 3) ;
gauss.fitTo(data) ;
```

所有这些拟合配置信息都会自动传递给 MINUIT。可以通过传递给 `fitTo()` 命令的可选命名参数来控制 MINUIT 的高级方面。这个示例启用 MINOS 方法来计算不对称误差，并将 MINUIT 的详细级别设置为最低值：

```
gauss.fitTo(data, RooFit::Minos(true), RooFit::PrintLevel(-1))
;
```

范围拟合

通过同样的接口可以影响似然函数的构建方式。要将似然（以及拟合）限制在指定范围内的事件子集上，可以这样做：

```
gauss.fitTo(data, Range(-5, 5)) ;
```

随后对该拟合的绘图默认只会显示拟合范围内的曲线。

关于拟合中似然函数的构建、进阶使用选项以及 χ^2 拟合的构建细节将在第12章中详细介绍。所有可用的 `fitTo()` 命令参数的参考指南见附录E以及方法 `RooAbsPdf::fitTo()` 的在线代码文档。

从模型生成数据

所有 RooFit 的概率密度函数 (p.d.f.) 都有一个通用接口，用于从其分布中生成事件。各种从分布中采样事件的技术已实现并在附录A中描述。RooAbsPdf 的内部逻辑会自动为每个使用案例选择最有效的技术。

最简单的形式中，可以从 p.d.f. 生成一个 RooDataSet，如下所示：

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

这个示例创建了一个包含 10000 个事件的 RooDataSet，这些事件的可观测量 x 是从 p.d.f. gauss 中采样的。

参数和可观测量

在本章的简单示例中，我们一直使用高斯概率密度函数（p.d.f.），并且明确假设变量 (x) 是可观测量，而变量 (\mu)（均值）和 (\sigma)（标准差）是我们的参数。这一区分非常重要，因为它直接关系到对象的函数表达式：概率密度函数相对于其可观测量是单位归一化的，但相对于其参数则不是。

然而，RooFit 的 p.d.f 类本身在参数和可观测量之间没有固有的静态区分概念。这乍看之下可能令人困惑，但为我们在构建复合对象时提供了必要的灵活性。参数和可观测量之间的区分总是有的，但它是从每个使用上下文中动态产生的。以下示例展示了如何将高斯函数用作可观测量均值的 p.d.f.：

```
RooDataSet* data = gauss.generate(mean,1000);
RooPlot* mframe = mean.frame();
data->plotOn(mframe);
gauss.plotOn(mframe);
```

鉴于高斯函数的数学表达式在 (x) 和 (m) 的互换下是对称的，因此不出所料，这在以 (x) 和 (\sigma) 为参数时，得到的是一个以 (m) 为可观测量的高斯分布。沿着同样的思路，也可以将高斯函数用作 (\sigma) 的 p.d.f.，以 (x) 和均值为参数。在许多情况下，不需要明确说明哪些变量是可观测量，因为其定义是从使用上下文中隐含地得出的。具体来说，只要一个使用上下文同时涉及 p.d.f. 和数据集，可观测量的隐含和自动定义是那些同时出现在数据集和 p.d.f. 定义中的变量。这种自动定义在拟合中很有效，因为拟合涉及一个显式的数据集，但也适用于绘图：RooPlot 框架变量始终被认为是可观测量。在其他所有涉及区分的上下文中，必须手动提供哪些变量被视为可观测量的定义。这就是为什么调用 `generate()` 时必须在每次调用中指定你认为的可观测量。

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

然而，在高斯函数的所有三种可能的用例中，它相对于（隐含声明的）可观测量都是一个适当归一化的概率密度函数。这突显了 RooFit “动态可观测量”概念的重要后果：RooAbsPdf 对象没有唯一的返回值，它取决于可观测量的局部定义。通过在 `RooAbsPdf::getVal()` 中的一个显式的事后归一化步骤实现了这种功能，这个步骤对于每种可观测量的定义是不同的。

```
Double_t gauss_raw = gauss.getVal(); // 未归一化的原始值
Double_t gauss_pdfX = gauss.getVal(x); // 用作 x 的 p.d.f 时的值
Double_t gauss_pdfM = gauss.getVal(mean); // 用作 mean 的 p.d.f 时的值
Double_t gauss_pdfS = gauss.getVal(sigma); // 用作 sigma 的 p.d.f 时的值
```

计算模型上的积分

在 RooFit 中，概率密度函数（p.d.f.）和函数的积分被表示为独立的对象。因此，与其将积分定义为一个动作，不如说积分是通过一个继承自 RooAbsReal 的对象来定义的，其值是通过积分操作计算得到的。这样的对象可以通过 `createIntegral()` 方法或 RooAbsReal 来构造。

例如：

```
1 RooAbsReal* intGaussX = gauss.createIntegral(x);
```

任何 RooAbsReal 函数或 RooAbsPdf 都可以通过这种方式进行积分。注意，对于 p.d.f.s，上述配置会对 gauss 的原始（未归一化）值进行积分。实际上，gauss 的归一化返回值 `gauss.getVal(x)` 恰好是 `gauss.getVal() / intGaussX->getVal()` 。

大多数积分由 RooRealIntegral 类的对象表示。构造此类时，会确定执行积分请求的最有效方法。如果被积分函数支持对所请求的可观测量进行解析积分，则会使用解析实现【5】；否则，将选择数值技术。实际的积分并不是在构造时进行的，而是在调用 `RooRealIntegral::getVal()` 时按需进行。一旦计算出来，积分值会被缓存，并在积分参数的值发生变化或者（一个或多个）积分可观测量的归一化范围发生变化时失效。

你可以通过打印积分对象来检查选择的积分策略：

```
1 intGaussX->Print("v");
```

输出示例：

```
--- RooRealIntegral ---
Integrates g[ x=x mean=m sigma=s ]
operating mode is Analytic
Summed discrete args are ()
Numerically integrated args are ()
Analytically integrated args using mode 1 are (x)
Arguments included in Jacobian are ()
Factorized arguments are ()
Function normalization set <none>
```

这一输出说明了以下几点：

- 被积分的函数（ g ）和涉及的参数（ x, \mu, \sigma ）。
- 操作模式为解析积分（Analytic）。
- 无需求和的离散参数。
- 无需数值积分的参数。
- （ x ）参数使用了模式 1 的解析积分。
- 雅可比矩阵中包含的参数为空。
- 无需因式分解的参数。
- 无设置的函数归一化。

归一化概率密度函数(p.d.f.)

也可以构造归一化p.d.f.的积分：

```
RooAbsReal* intGaussX = gauss.createIntegral(x,NormSet(x)) ;
```

这个例子并没有太大实际用途，因为它总是返回1，但使用相同的接口，也可以在观测变量的预定义子区间上进行积分，

```
x.setRange("signal",-2,2) ;
RooAbsReal* intGaussX =
gauss.createIntegral(x,NormSet(x),Range("signal")) ;
```

以提取模型在“信号”区间内的部分。诸如“信号”之类的命名区间的概念将在第3章和第7章详细阐述。归一化p.d.f.积分的返回值自然位于[0,1]范围内。

累积分布函数

积分p.d.f.的一种特殊形式是累积分布函数（c.d.f.），其定义如下，并且可以通过专门的方法*createCdf()*从任何p.d.f.构造：

```
RooAbsReal* cdf = pdf->createCdf(x) ;
```

图6展示了从高斯p.d.f.创建的c.d.f.的一个例子。对于这种形式的积分，*createCdf()*相对于*createIntegral()*的优势在于前者能够更有效地处理需要数值积分的p.d.f.：*createIntegral()*在一个或多个参数变化后会从头开始重新计算整个数值积分，而*createCdf()*则会缓存数值积分采样阶段的结果，只重新计算求和部分。有关积分和累积分布函数的更多细节见附录C。

2 信号和背景 – 复合模型介绍

2.1 简介

数据模型通常用于描述包含多种事件假设的样本，例如信号和（一个或多个类型的）背景。为了描述这种性质的样本，可以构建一个复合模型。对于事件假设‘信号’和‘背景’，复合函数M(x)可以通过描述信号的函数S(x)和描述背景的函数B(x)构建，如下所示：

$$M(x) = fS(x) + (1 - f)B(x)$$

在这个公式中，（f）是样本中信号事件的比例。多个假设的通用表达式为：

$$M(x) = \sum_{i=1}^{N-1} f_i F_i(x) + \left(1 - \sum_{i=1}^{N-1} f_i\right) F_N(x)$$

这种方式添加p.d.f.的一个特性是，M(x)不需要特意地归一化为1：如果S(x)和B(x)都是归一化为1的，那么通过这种构造，M(x)也是归一化的。RooFit提供了一个特殊的‘加法运算符’p.d.f.在RooAddPdf类中，以简化构建和使用这种复合p.d.f.。

扩展似然方法

测量结果通常以事件数而非事件比例的形式引用，因此，直接用信号和背景事件的数量而不是信号事件的比例（和总事件数）来表示数据模型是很有用的。具体表达如下：

$$M_E(x) = N_S S(x) + N_B B(x)$$

在这个表达式中，（ M_E(x) ）不是归一化为1，而是归一化为（ N_S + N_B = N ），即数据样本中的事件总数。因此，这不是一个严格的概率密度函数，而是两个表达式的简写：分布的形状和期望事件数。

$$M(x) = \left(\frac{N_S}{N_S+N_B}\right) S(x) + \left(\frac{N_B}{N_S+N_B}\right) B(x)$$

$$\text{Expected } N = N_S + N_B$$

在扩展似然方法中，可以联合约束这些表达式：

$$-\log L(p) = -\sum_{\text{data}} \log M(x_i) - \log \text{Poisson}(N_{\text{expected}} - N_{\text{expected}})$$

在RooFit中，普通的加和（（ N*{\text{coef}} = N{\text{pdf}} - 1 ））和扩展似然加和（（ N*{\text{coef}} = N{\text{pdf}} ））都由运算符类RooAddPdf表示，后者会自动构建扩展似然项。

2.2 构建具有分数系数的复合模型

我们首先从简单（非扩展）复合模型的描述开始。以下是使用分数系数通过RooAddPdf构建复合概率密度函数（PDF）的一个简单示例。

```
RooRealVar x("x", "x", -10, 10);
RooRealVar mean("mean", "mean", 0, -10, 10);
RooRealVar sigma("sigma", "sigma", 2, 0., 10.);
RooGaussian sig("sig", "signal p.d.f.", x, mean, sigma);

RooRealVar c0("c0", "coefficient #0", 1.0, -1., 1.);
RooRealVar c1("c1", "coefficient #1", 0.1, -1., 1.);
RooRealVar c2("c2", "coefficient #2", -0.1, -1., 1.);
RooChebychev bkg("bkg", "background p.d.f.", x, RooArgList(c0, c1, c2));

// model(x) = fsig*sig(x) + (1-fsig)*bkg(x)
RooAddPdf model("model", "model", RooArgList(sig, bkg), fsig);
```

在这个示例中，我们首先构建一个高斯概率密度函数(sig)和一个平坦的背景概率密度函数(bkg)，然后使用信号分数(fsig)将它们加在一起构成模型。请注意，使用容器类RooArgList来作为函数的单一参数传递对象列表。RooFit有两个容器类：RooArgList和RooArgSet。每个容器类可以包含任意数量的RooFit值对象，即任何继承自RooAbsArg的对象，如RooRealVar、RooAbsPdf等。区别在于列表list是有序的，可以通过位置引用（第2个，第3个等）访问元素，并且可以包含多个同名对象，而集合set是无序的，但要求每个成员有唯一的名称。

RooAddPdf实例可以将任意数量的成分相加，要用两个系数添加三个概率密度函数，可以写成：

```
// model2(x) = fsig*sig(x) + fbkg1*bkg1(x) + (1-fsig-fbkg)*bkg2(x)
RooAddPdf model2("model2","model2",RooArgList(sig,bkg1,bkg2), RooArgList(fsig,fbkg1)) ;
```

要构建一个非扩展的概率密度函数，其中系数被解释为分数，那么系数的数量应总是比概率密度函数的数量少一个。

使用RooAddPdf递归

请注意，RooAddPdf的输入p.d.f.不需要是基本的p.d.f.，它们本身可以是复合p.d.f.。请看下面这个使用了Example 7中的sig和bkg作为输入的例子：

例子5 – 通过递归添加两项来添加三个p.d.f.

```
// 构造第三个pdf bkg_peak
RooRealVar mean_bkg("mean_bkg","mean",0,-10,10);
RooRealVar sigma_bkg("sigma_bkg","sigma",2,0.,10.);
RooGaussian bkg_peak("bkg_peak","peaking bkg
p.d.f.",x,mean_bkg,sigma_bkg);

// 首先将sig和peak以fpeak的比例相加
RooRealVar fpeak("fpeak","peaking background
fraction",0.1,0.,1.);
RooAddPdf
sigpeak("sigpeak","sig+peak",RooArgList(bkg_peak,sig),fpeak);

// 然后将(sig+peak)以fbkg的比例与bkg相加
RooRealVar fbkg("fbkg","background fraction",0.5,0.,1.);
RooAddPdf model("model","bkg+
(sig+peak)",RooArgList(bkg,sigpeak),fbkg);
```

最终的p.d.f.模型表示如下表达式：

$$M(x) = f_{\text{bkg}} B(x) + (1 - f_{\text{bkg}}) \left[f_{\text{peak}} P(x) + (1 - f_{\text{peak}}) S(x) \right]$$

也可以通过单个RooAddPdf的递归模式构建这样的递归加法公式。在这种构造模式下，系数的解释如下：

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

例如，要构建与上面模型对象功能等价的模型，可以写成：

例子6 – 使用RooAddPdf的递归模式递归地添加三个p.d.f.

```
RooAddPdf model("model","recursive addition model",
RooArgList(bkg, bkg_peak, sig), RooArgList(fbkg, fpeak, fsig),
kTRUE);
```

这样，最终模型的形式与例子5中的递归加法表达式等效。
$M(x) = (f_1F_1 + (1 - f_1)(f_2F_2 + (1 - f_2)(f_3F_3 + (1 - f_3)(f_4F_4 + (1 - f_4)F_5))))$

2.3 绘制复合模型

复合p.d.f.的模块化结构允许你处理各个单独的组件。例如，可以在模型上绘制复合模型的各个组件，以可视化其结构。

```
RooPlot* frame = x.frame();
model.plotOn(frame);
model.plotOn(frame, Components(bkg), LineStyle(kDashed));
frame->Draw();
```

上述代码片段的输出如图7所示。组件图以虚线样式绘制。有关绘图样式选项的完整概述，请参见附录C。

你可以通过对象引用来识别组件，如上所示，或者通过名称来识别组件：

```
model.plotOn(frame, Components("bkg"),LineStyle(kDashed)) ;
```

如果你的绘图代码无法访问组件对象，例如，如果你的模型是在一个只返回顶级RooAddPdf对象的独立函数中构建的，那么后一种方法非常方便。

如果你想绘制多个组件的和，也可以通过两种方式实现：

```
model.plotOn(frame, Components(RooArgSet(bkg1, bkg2)),
LineStyle(kDashed));
model.plotOn(frame, Components("bkg1,bkg2"),
LineStyle(kDashed));
```

请注意，在后一种形式中，允许使用通配符，因此如果选择一个合适的组件命名方案，例如，可以这样做：

```
RooAddPdf model("model", "bkg+(sig+peak)", RooArgList(bkg,
peak, bkg), RooArgList(fbkg, fpeak), kTRUE);
model.plotOn(frame, Components("bkg"), LineStyle(kDashed));
model.plotOn(frame, Components("bkg*"), LineStyle(kDashed));
```

如果需要，可以在逗号分隔的列表中指定多个通配符表达式。

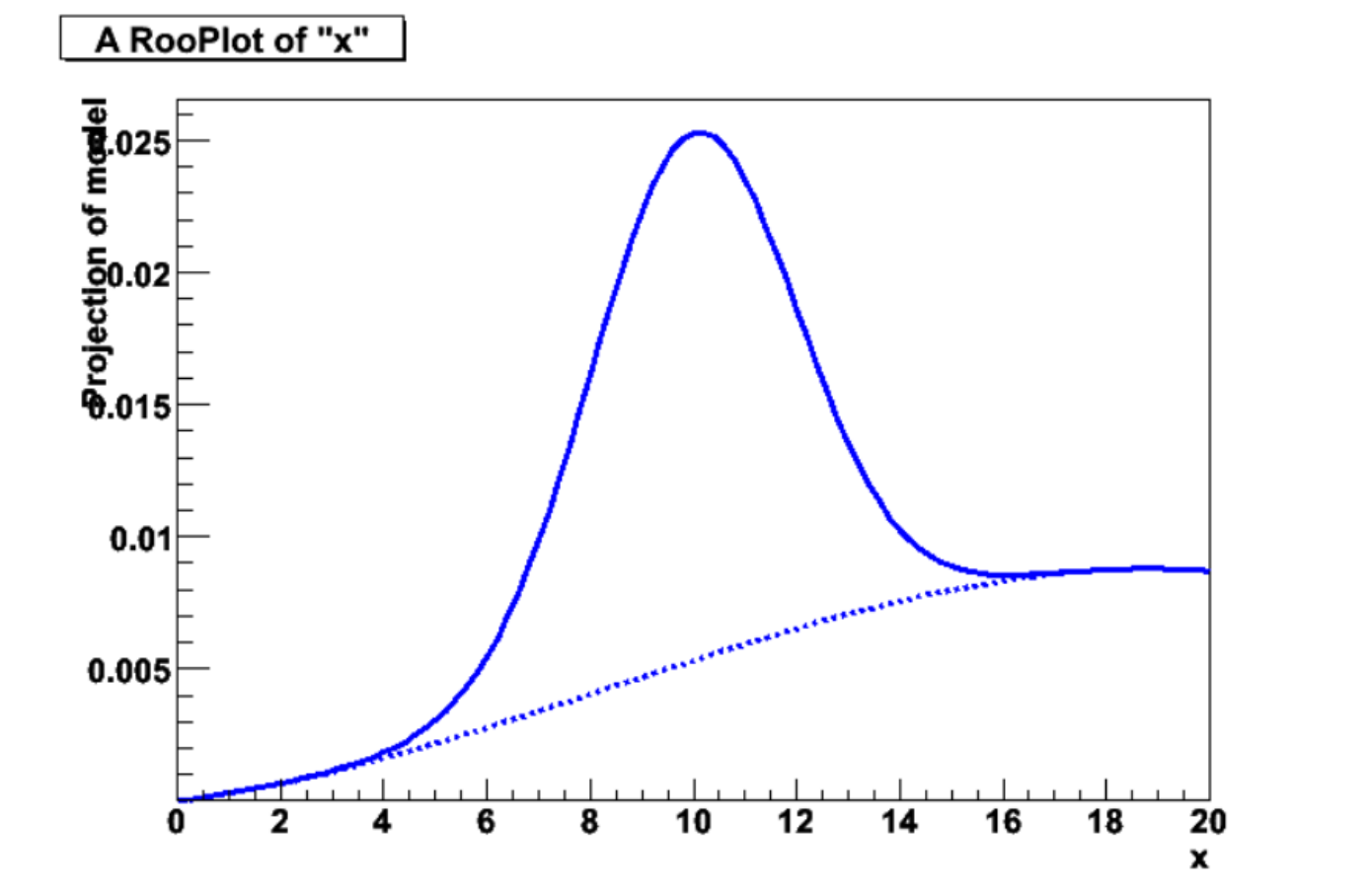


Figure 7 – Drawing of composite model and its components

2.4 使用复合模型

拟合复合模型

拟合带有分数系数的复合模型与拟合任何其他模型没有区别：

```
model.fitTo(data);
```

模型的参数包括组件p.d.f.的参数以及由加法运算符类引入的分数参数。

使用多个分数拟合的常见陷阱

在涉及直（非递归）加法多个组件的模型中定义分数参数的允许范围时，需要注意一些事项。如果两个组件通过一个分数相加，那么该分数的自然范围是$0, 1$。但是，如果添加了多个组件，就会有多个分数。尽管将每个分数的允许范围保持在$0, 1$是合法的，但这可能会导致系数之和超过1的配置，例如当$f_1 = f_2 = 0.7$时。如果发生这种情况，最后的系数（自动计算为$1 - \sum_{i=1}^{N-1} f_i$）将变为负数。

如果在拟合过程中出现这种配置，RooFit将在每次发生时打印警告消息，但只要在似然评估的每个点上RooAddPdf的返回值仍为正，就不会采取任何措施。如果你想避免这种配置，有几种选择。

一种方法是使用RooRealVar::setRange()收紧所有分数的允许范围，使它们相加时永远不会超过1。这种方法需要一些关于你拟合的分布的知识，以避免禁止最佳拟合配置。

另一种方法是使用递归加法，在这种方法中，分数值范围$0, 1$的每种排列都会生成有效的正定复合pdf。这种方法改变了系数的解释，但对要建模的分布形状不做任何假设。

第三种方法是使用扩展的似然拟合，其中所有系数都明确指定，没有隐式计算的剩余分数变为负数的可能性。

使用复合模型生成数据

使用复合模型生成事件的接口与使用基本模型生成事件的接口相同。

```
// 生成10000个事件
RooDataSet* x = model.generate(x,10000);
```

在内部，RooFit将利用p.d.f.的复合结构，并将事件生成委托给组件p.d.f.的方法，这通常更有效。

2.5 构建扩展复合模型

为了构建可以与扩展似然拟合一起使用的复合p.d.f.，需要为每个组件指定相应的系数：

```
RooRealVar nsig("nsig","signal fraction",500,0.,10000.) ;
RooRealVar nbkg("nbkg","background fraction",500,0.,10000.) ;
RooAddPdf model("model", "model", RooArgList(sig, bkg),
RooArgList(nsig, nbkg));
```

示例 7 – 使用两个事件计数系数添加两个 p.d.f.

在这个例子中，系数参数的允许范围已经调整为可以容纳事件计数而不是分数。从实际角度来看，示例 7 和示例 4 构建的模型之间的区别在于，示例 7 中的 RooAbsPdf 对象模型可以通过其成员函数 `expectedEvents()` 预测预期的数据事件数（即 nsig + nbkg），而示例 4 中的模型不能。示例 7 形式通过将每个系数除以所有系数的总和来得到组件分数。

还可以构建两个或多个已经是扩展 p.d.f. 的组件 p.d.f. 的和，在这种情况下，无需提供系数来构建扩展和 p.d.f.。这些输入可以是之前构建的 RooAddPdfs（使用扩展模式选项）或通过 RooExtendPdf 实用程序 p.d.f. 扩展的普通 p.d.f.。

```
RooRealVar nsig("nsig", "signal fraction", 500, 0., 10000.);
RooRealVar nbkg("nbkg", "background fraction", 500, 0., 10000.);
RooExtendPdf esig("esig", "esig", sig, nsig);
RooExtendPdf ebkg("ebkg", "ebkg", bkg, nbkg);
 RooAddPdf model("model", "model", RooArgList(esig, ebkg));
```

范围内信号事件产量

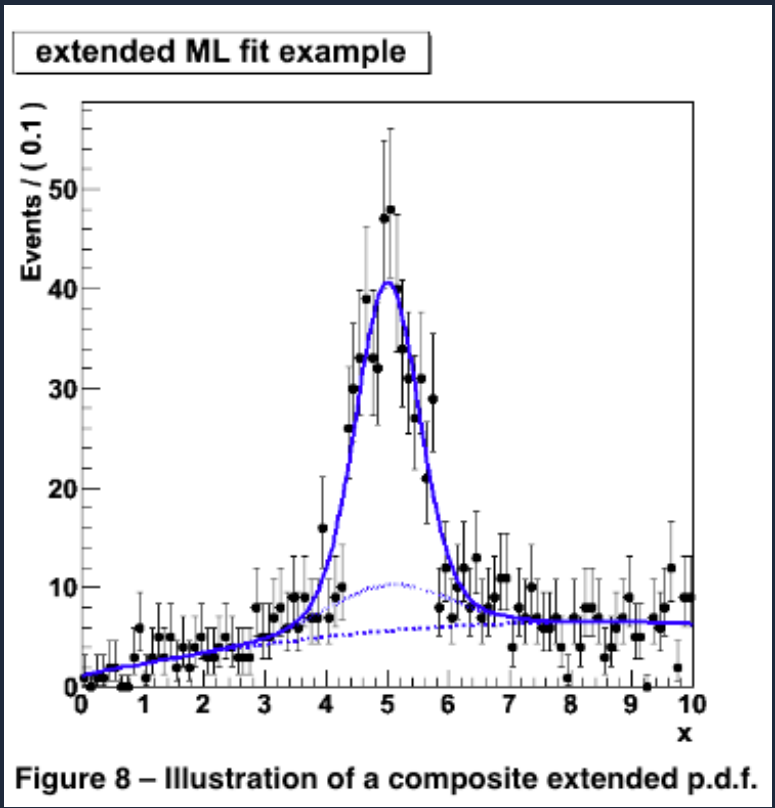


Figure 8 – Illustration of a composite extended p.d.f.

假设你对图 8 中模型的范围$4,6$内的信号事件产量感兴趣：

你可以通过将总信号产量乘以信号 p.d.f. 形状在范围$4,6$内的分数来计算这一点，但仍然需要手动传递信号产量和形状分数积分的误差到最终结果。RooExtendPdf 类提供了在预期事件数的计算中立即应用转换的可能性，以便似然函数，从而拟合结果，直接以 nsigWindow 表示，并且所有误差都会自动正确传播。

这种修改的效果是 *esig* 返回的预期事件数变为：

$$N_{\text{sig}}^{\text{expected}} = N_{\text{sig}}^{\text{window}} \int_4^6 S(x)\,dx$$

这样，在最小化扩展最大似然后，nsigw 等于信号窗口中事件数的最佳估计值。关于范围内积分和归一化操作的更多详细信息，请参见附录 D。

```
x.setRange("window", 4, 6);
RooAbsReal* fracSigRange = sig.createIntegral(x, x, "window");
Double_t nsigWindow = nsig.getVal() * fracSigRange->getVal();
//先计算窗口区的信号比例，再用总信号数*窗口区比例=窗口区的信号数，但不能传递误差
RooRealVar nsigw("nsigw", "nsignal in window", 500, 0, 10000.);
RooExtendPdf esig("esig", "esig", sig, nsigw, "window");
```

通过这种方式，您可以在拟合时直接得到信号窗口内的事件数及其误差。

使用扩展复合模型生成事件

从扩展模型生成事件

一些额外的特性适用于为扩展似然形式构建的复合模型。由于这些模型预测了一定数量的事件，因此可以省略请求生成的事件数量：

```
RooDataSet* x = model.generate(x);
```

在这种情况下，将生成由概率密度函数（p.d.f.）预测的事件数量。您还可以选择通过 `Extended()` 参数引入泊松波动到生成的事件数量中：

```
RooDataSet* x = model.generate(x, Extended(kTRUE));
```

如果您在研究中生成了许多样本并查看拉量分布，这是非常有用的。为了使事件计数参数的拉量分布正确，生成的事件总数应存在泊松波动。关于拟合研究和拉量分布的详细内容，请参见第14章。

拟合

复合扩展概率密度函数（p.d.f.）只有在包含扩展似然项进行最小化时才能成功拟合，因为它们在参数化中有一个由该扩展项约束的额外自由度。如果一个概率密度函数能够计算扩展项（例如任何扩展的 `RooAddPdf` 对象），则扩展项会自动包含在似然计算中。您可以通过在 `fitTo()` 调用中添加 `Extended()` 参数手动覆盖此默认行为：

```
model.fitTo(data, Extended(kTRUE)); // 可选
```

绘图

扩展似然模型的可视化默认程序与常规概率密度函数相同：用于归一化的事件计数是添加到绘图框架中的最后一个数据集的事件计数。您可以选择覆盖此行为并使用概率密度函数的预期事件计数进行归一化，如下所示：

```
model.plotOn(frame, Normalization(1.0,
RooAbsReal::RelativeExtended));
```