ROOT 入门导读

英文原文参阅：https://root.cern.ch/root/htmldoc/guides/primer/ROOTPrimer.html
ROOT 下载地址：https://root.cern.ch/downloading-root
ROOT 安装方法：https://mp.weixin.qq.com/s/VFGWAl2mZnvLwMmIGchv9A

翻译 by：Google and 赵问问
联系方式：zhaoww2013@126.com
2018 年 11 月 17 日星期六

目录

摘要：

ROOT 是一个用于数据分析和 I / O 的软件框架：一个强大的工具，可以应对最先进的科学数据分析的典型任务。 它的突出特点包括高级图形用户界面，非常适合交互式分析，C ++编程语言的解释器，快速高效的原型设计和 C ++对象的持久性机制，还用于写入大型强子对撞机实验记录的每年 PB 级数据（1PB=1024TB 译者注）。 本入门指南说明了 ROOT 的主要特征，这些特征与数据分析的典型问题相关：输入和绘制测量数据和分析功能的拟合。

原创作者 - D. Piparo - G. Quast - M. Zeise

译者注：本文均是 Google 翻译结果，仅对代码和板式作调整，欢迎修改分享

1 软件背景与简介

欢迎来到数据分析 ROOT！

测量与理论模型的比较是实验物理学中的标准任务之一。在最简单的情况下，“模型”只是提供测量数据预测的函数。通常，模型取决于参数。这种模型可以简单地表示“电流 I 与电压 U 成比例”，并且实验者的任务包括从一组测量中确定电阻 R.

作为第一步，需要数据的可视化。接下来，通常必须应用一些操作，例如，校正或参数转换。通常，这些操作是复杂的，并且应该提供强大的数学函数和程序库 - 例如，考虑应用于输入光谱的积分或峰值搜索或傅立叶变换以获得模型描述的实际测量值。

实验物理学的一个特点是影响每个测量的不可避免的不确定性，可视化工具必须包括这些。在随后的分析中，必须正确处理错误的统计性质。

作为最后一步，将测量值与模型进行比较，并且需要在此过程中确定自由模型参数。有关适合数据点的函数（模型）的示例，请参见图 1.1。有几种标准方法可供使用，数据分析工具应能方便地访问其中一种以上。还必须提供量化测量和模型之间一致性水平的方法。通常，要分析的数据量很大 - 考虑借助计算机累积的细粒度测量。因此，可用工具必须包含易于使用且有效的方法来存储和处理数据。

在量子力学中，模型通常仅根据许多参数预测测量的概率密度函数（“pdf”），并且实验分析的目的是从观察到的频率分布中提取参数，其中观察测量。这种测量需要生成和可视化频率分布的装置，所谓的直方图和严格的统计处理，以从纯粹的统计分布中提取模型参数。预期数据的模拟是数据分析的另一个重要方面。通过重复生成“伪数据”，其以与用于真实数据的预期相同的方式进行分析，可以验证或比较分析过程。在许多情况下，测量误差的分布并不是精确已知的，并且模拟提供了测试不同假设的影响的可能性。

满足上述所有要求的强大软件框架是 ROOT，这是一个由日内瓦欧洲核子研究中心欧洲核研究组织协调的开源项目。

ROOT 非常灵活，既可以在自己的应用程序中使用编程接口，也可以提供用于交互式数据分析的图形用户界面。本文档的目的是作为初学者指南，并根据学生实验室中解决的典型问题为您自己的用例提供可扩展的示例。本指南有望为您未来科学工作中更复杂的应用奠定基础，建立在现代，最先进的数据分析工具之上。

本指南以教程的形式向您介绍 ROOT 包。根据“边做边学”的原则，这个目标将通过具体的例子来完成。也正因为这个原因，本指南无法涵盖 ROOT 包的所有复杂性。然而，一旦您对以下章节中介绍的概念有信心，您将能够欣赏 ROOT 用户指南（The Root Users Guide 2015）并浏览类参考（根参考指南 2013）以查找所有详细信息您可能会感兴趣。您甚至可以查看代码本身，因为 ROOT 是一个免费的开源产品。与本教程并行使用这些文档！

ROOT 数据分析框架本身是编写的，并且在很大程度上依赖于 C ++编程语言：需要一些关于 C ++的知识。如果您不了解这种语言的含义，Jus 可以利用有关 C ++的大量文献。ROOT 可用于许多平台（Linux，Mac OS X，Windows ...），但在本指南中我们将隐含地假设您使用的是 Linux。你需要做的第一件事就是安装 ROOT，不是吗？获取最新的 ROOT 版本非常简单。只需在此网页 http://root.cern.ch/downloading-root 上寻找“专业版”。您将找到针对不同体系结构的预编译版本，或者您自己编译的 ROOT 源代码。只需拿起您需要的味道并按照安装说明操作即可。

让我们深入了解 ROOT！

2 ROOT 基础

既然你已经安装了 ROOT，那么你正在运行的这个交互式 shell 是什么？ 就像这样：ROOT 带来了双重功能。 它有一个宏的解释器（Cling（“What is Cling”2015）），您可以从命令行运行或像应用程序一样运行。但它也是一个可以评估任意语句和表达式的交互式 shell。这对于调试，快速黑客攻击和测试非常有用。

我们先来看一些非常简单的例子。

2.1 ROOT 作为计算器

您甚至可以使用 ROOT 交互式 shell 代替计算器！ 使用该命令启动 ROOT 交互式 shell

```
> root
```

在你的 Linux 机器上。 提示应该很快出现：

```
root [0]
```

让我们来看看这里显示的步骤：

```
root [0] 1+1
(int) 2
root [1] 2*(4+2)/12.
(double) 1.000000
root [2] sqrt(3.)
(double) 1.732051
root [3] 1 > 2
(bool) false
root [4] TMath::Pi()
(double) 3.141593
root [5] TMath::Erf(.2)
(double) 0.222703
```

不错。 您可以看到，ROOT 不仅可以输入 C ++语句，还可以输入存在于 TMath 命名空间中的高级数学函数。

现在让我们做一些更详尽的事情。 一个众所周知的几何系列的数字示例：

```
root [6] double x=.5
(double) 0.500000
root [7] int N=30
(int) 30
```

```
root [8] double geom_series=0
(double) 0.000000
root [9] for (int i=0;i<N;++i)geom_series+=TMath::Power(x,i)
root [10]  cout << TMath::Abs(geom_series - (1-TMath::Power(x,N-1))/(1-
x)) <<endl;
1.86265e-09
```

在这里，我们向前迈进了一步。 我们甚至声明了变量并使用了一个控制结构。 请注意，Cling 和标准 C ++语言之间存在一些细微差别。 在交互模式下，您不需要在行尾的“;” - 尝试区别，例如 在 line root [6]使用命令。

2.2 在 ROOT 命令行学习 C++

在 ROOT 提示符后面有一个基于真实编译器工具包的解释器：LLVM。 因此，可以使用 C ++和标准库的许多功能。 例如，在下面的代码片段中，我们定义了一个 lambda 函数，一个向量，我们以不同的方式对它进行排序：

```
root [0] using doubles = std::vector<double>;
root [1] auto pVec = [](const doubles& v){for (auto&& x:v) cout << x <<
endl;};
root [2] doubles v{0,3,5,4,1,2};
root [3] pVec(v);
0
3
5
4
1
2
root [4] std::sort(v.begin(),v.end());
root [5] pVec(v);
0
1
2
3
4
5
root [6] std::sort(v.begin(),v.end(),[](double a, double b){return
a>b;});
root [7] pVec(v);
5
4
3
2
```

```
1
0
```

或者，如果您更喜欢随机数生成：

```
root [0] std::default_random_engine generator;
root [1] std::normal_distribution<double> distribution(0.,1.);
root [2] distribution(generator)
(std::normal_distribution<double>::result_type) -1.219658e-01
root [3] distribution(generator)
(std::normal_distribution<double>::result_type) -1.086818e+00
root [4] distribution(generator)
(std::normal_distribution<double>::result_type) 6.842899e-01
```

令人印象深刻的不是吗？

2.3 ROOT 画函数图像

使用 ROOT 强大的类之一，这里 TF1 将允许我们显示一个变量 x 的函数。 请尝试以下方法：

```
root [11] TF1 f1("f1","sin(x)/x",0.,10.);
root [12] f1.Draw();
```

f1 是 TF1 类的一个实例，参数在构造函数中使用；第一个类型字符串是要在内部 ROOT 内存管理系统中输入的名称，第二个字符串类型参数定义函数，这里是 sin（x）/ x，而 double 类型的两个参数定义变量的范围 X。Draw（）方法，这里没有任何参数，在一个窗口中显示该函数，在您输入上面两行之后应该弹出该窗口。

此示例的稍微扩展版本是带有参数的函数的定义，在 ROOT 公式语法中称为[0]，[1]等。我们现在需要一种为这些参数赋值的方法；这是通过类 TF1 的 SetParameter（<parameter_number>，<parameter_value>）方法实现的。 这是一个例子：

```
root [13] TF1 f2("f2","[0]*sin([1]*x)/x",0.,10.);
root [14] f2.SetParameter(0,1);
root [15] f2.SetParameter(1,1);
root [16] f2.Draw();
```

当然，这个版本显示的结果与初始版本相同。尝试播放参数并再次绘制函数。 TF1 类具有大量非常有用的方法，包括积分和区分。要充分利用此类和其他 ROOT 类，请访问 Internet 上的文档 http://root.cern.ch/drupal/content/reference-guide。 ROOT 中的公式使用类 TFormula 进行评估，因此还要查看相关的类文档以获取示例，实现的函数和语法。

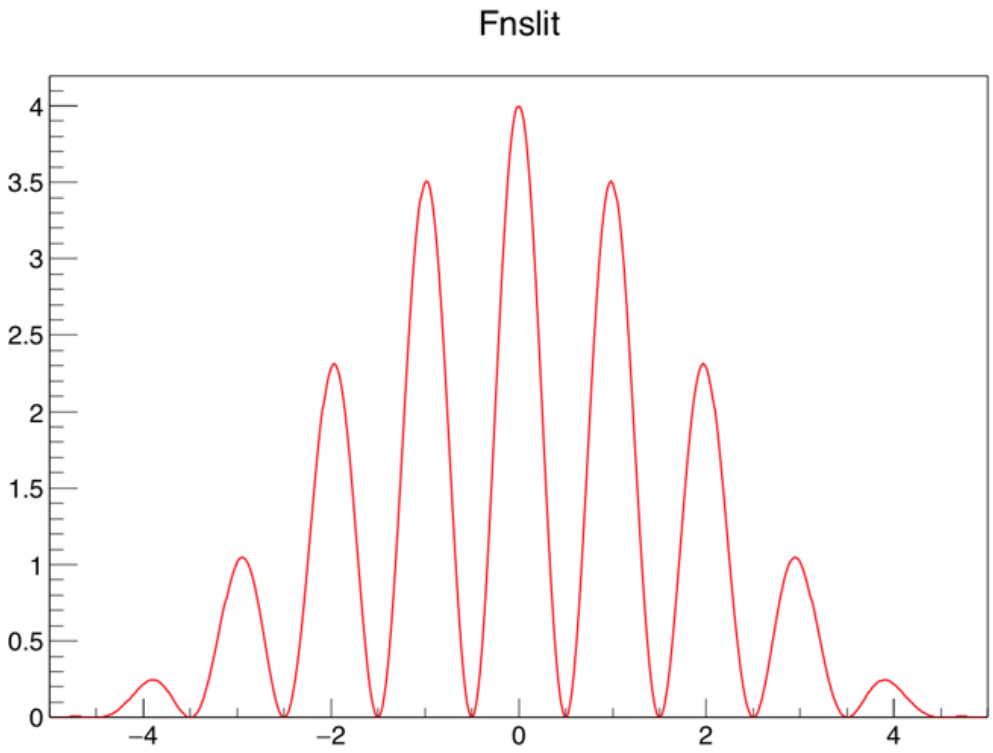
您一定要将本指南下载到您自己的系统，以便随时随地使用。

要在上面的示例中稍微扩展一下，请考虑一个您想要定义的更复杂的函数。您也可以使用标准 C 或 C ++代码执行此操作。

考虑下面的示例，该示例计算并显示由落在多个狭缝上的光产生的干涉图案。请不要在 ROOT 命令行输入以下示例，有一种更简单的方法：确保磁盘上有文件 slits.C，并在 shell 中键入 root slits.C.这将启动 root 并使其读取“宏”slits.C，即文件中的所有行将一个接一个地执行。

```
// Example drawing the interference pattern of light
// falling on a grid with n slits and ratio r of slit
// width over distance between slits.

auto pi = TMath::Pi();

// function code in C
double single(double *x, double *par) {
  return pow(sin(pi*par[0]*x[0])/(pi*par[0]*x[0]),2);
}

double nslit0(double *x,double *par){
  return pow(sin(pi*par[1]*x[0])/sin(pi*x[0]),2);
}

double nslit(double *x, double *par){
  return single(x,par) * nslit0(x,par);
}

// This is the main program
void slits() {
  float r,ns;

  // request user input
  cout << "slit width / g ? ";
  scanf("%f",&r);
  cout << "# of slits? ";
  scanf("%f",&ns);
  cout <<"interference pattern for "<< ns
       <<" slits, width/distance: "<<r<<endl;

  // define function and set options
  TF1 *Fnslit  = new TF1("Fnslit",nslit,-5.001,5.,2);
  Fnslit->SetNpx(500);

  // set parameters, as read in above
```

```
  Fnslit->SetParameter(0,r);
  Fnslit->SetParameter(1,ns);

  // draw the interference pattern for a grid with n slits
  Fnslit->Draw();
}
```



参数为 0.2 和 2 的 slits.C 输出图像

该示例首先要求用户输入，即狭缝宽度与狭缝距离的比率和狭缝的数量。输入此信息后，您应该看到如图 2.1 所示的图形输出。

这是一个比我们之前看到的更复杂的例子，所以花一些时间仔细分析它，你应该在继续之前理解它。让我们详细介绍一下：

第 7-18 行定义了 C ++代码中的必要函数，分成三个独立的函数，如所考虑的问题所示。完全干涉图案由函数的乘积给出，该函数取决于狭缝的宽度和距离的比率，并且第二个取决于狭缝的数量。对我们来说更重要的是定义这些函数的接口，使它们可用于 ROOT 类 TF1：第一个参数是指向 x 的指针，第二个参数指向参数数组。

主程序从第 21 行开始，定义了 void 类型的函数 slits（）。在询问用户输入之后，使用开头给出的 C 类型函数定义 ROOT 函数。我们现在可以使用 TF1 类的所有方法来控制我们函数的行为 - 很好，不是吗？

如果您愿意，可以在 TF1 实例中轻松扩展示例以使用函数双单，或具有窄狭缝的网格，函数双 nslit0 来绘制单个狭缝的干涉图案。

在这里，我们使用了一个宏，某种轻量级程序，与 ROOT，Cling 一起分发的解释器能够执行。这是一个非常特殊的情况，因为 C ++本身不是一种解释语言！还有很多话要说：章确实是专门用于宏的。

2.4 控制 ROOT

此时还有一个注意事项：因为您键入 ROOT 的每个命令通常都由 Cling 解释，所以需要一个“转义字符”来直接将命令传递给 ROOT。 该字符是一行开头的点：

```
root [1] .<command>
```

这是最常见命令的选择。

退出 root，只需键入.q

获取命令列表，使用。？

访问操作系统的 shell，输入。！<OS_command>; 尝试，例如 。！ls 或。！pwd

执行一个宏，输入.x <file_name>; 在上面的示例中，您可能在 ROOT 提示符下使用了.x slits.C.

加载一个宏，输入.L <file_name>; 在上面的例子中，您可能改为使用命令.L slits.C，然后使用函数调用 slits（）;. 请注意，在加载宏之后，ROOT 提示符中提供了其中定义的所有函数和过程。

编译一个宏，输入.L <file_name> +; ROOT 能够在幕后为您管理 C ++编译器，并从您的宏开始生成机器代码。 可以决定编译宏以获得更好的性能或更接近生产环境。

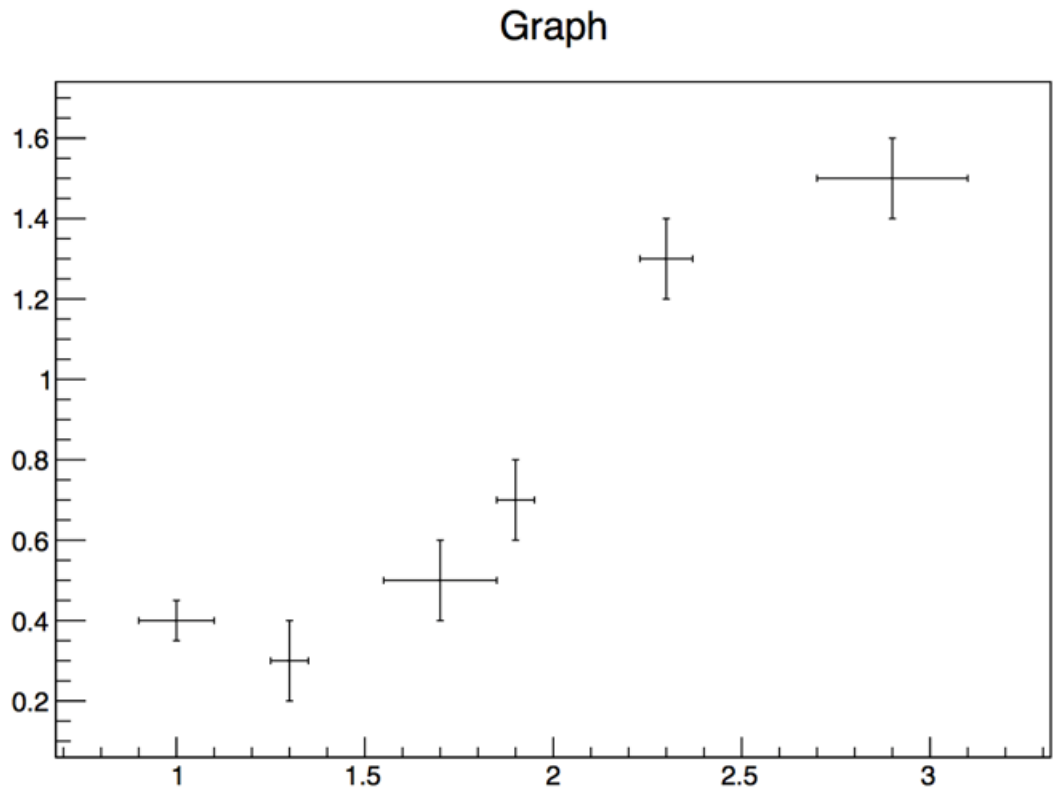
在提示符处使用.help 来检查完整列表。

2.5 ROOT 绘制测量值

要在 ROOT 中显示测量值，包括错误，存在一个强大的类 TGraphErrors，它具有不同类型的构造函数。 在此处的示例中，我们使用文本格式的 ExampleData.txt 文件中的数据：

```
root [0] TGraphErrors gr("ExampleData.txt");
root [1] gr.Draw("AP");
```

您应该看到如图 2.2 所示的输出。



使用 TGraphErrors 类可视化有误差的数据点。

确保文件 ExampleData.txt 在您启动 ROOT 的目录中可用。现在使用您喜欢的编辑器检查此文件，或使用命令 less ExampleData.txt 检查文件，您将看到格式非常简单易懂。以＃开头的行将被忽略。添加一些关于数据类型的注释非常方便。数据本身由具有四个实数的线组成，表示每个数据点的 x 和 y 坐标及其误差。

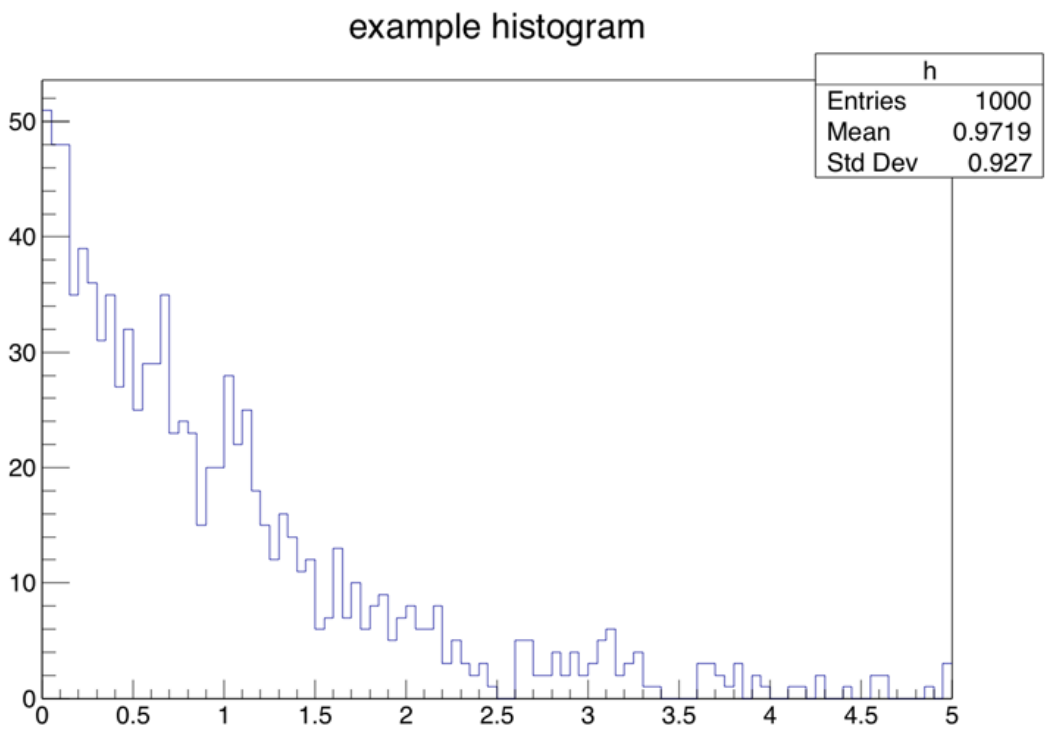
Draw（“AP”）方法的参数在这里很重要。在幕后，它告诉 TGraphPainter 类显示轴并在指定数据点的 x 和 y 位置绘制标记。请注意，此简单示例依赖于 ROOT 的默认设置，包括绘制绘图的画布大小，标记类型以及使用的线条颜色和粗细等。在一个编写良好的完整示例中，需要明确指定所有这些以获得良好且可读的结果。完整的图章将更详细地解释 TGraphErrors 类的更多功能及其与其他 ROOT 类的关系。

2.6 ROOT 绘制直方图

ROOT 中的频率分布由从直方图类 TH1 派生的一组类来处理，在我们的例子中是 TH1F。字母 F 代表“float”，这意味着数据类型 float 用于将条目存储在一个直方图 bin 中。

```
root [0] TF1 efunc("efunc","exp([0]+[1]*x)",0.,5.);
root [1] efunc.SetParameter(0,1);
root [2] efunc.SetParameter(1,-1);
root [3] TH1F h("h","example histogram",100,0.,5.);
root [4] for (int i=0;i<1000;i++) {h.Fill(efunc.GetRandom());}
root [5] h.Draw();
```

此示例的前三行定义了一个函数，在这种情况下为指数，并设置其参数。 在第 3 行中，实例化直方图，其名称，标题，一定数量的箱（其中 100 个，等距，大小相等）在 0 到 5 的范围内。



用指数分布的随机数填充的直方图的可视化。

我们使用 ROOT 的另一个新功能来填充此直方图的数据，即使用方法 TF1 :: GetRandom 生成的伪随机数，后者又使用 ROOT 启动时创建的 ROOT 类 TRandom 的实例。使用方法 TH1F ::填充循环结构，在第 4 行的直方图中输入数据。结果，直方图填充有根据定义的函数分布的 1000 个随机数。使用 TH1F :: Draw（）方法显示直方图。您可以将此示例视为量子力学状态的生命时间的重复测量，其被输入到直方图中，从而给出概率密度分布的视觉印象。该图如图 2.3 所示。

请注意，执行上面的行时，您将无法获得相同的图，具体取决于随机数生成器的初始化方式。

TH1F 类不包含纯文本文件的方便输入格式。以下 C ++代码行完成了这项工作。存储在文本文件“expo.dat”中的每行一个数字通过输入流读入并填充直方图，直到到达文件末尾。

```
root [1] TH1F h("h","example histogram",100,0.,5.);
root [2] ifstream inp; double x;
root [3] inp.open("expo.dat");
root [4] while (inp >> x) { h.Fill(x); }
root [5] h.Draw();
root [6] inp.close();
```

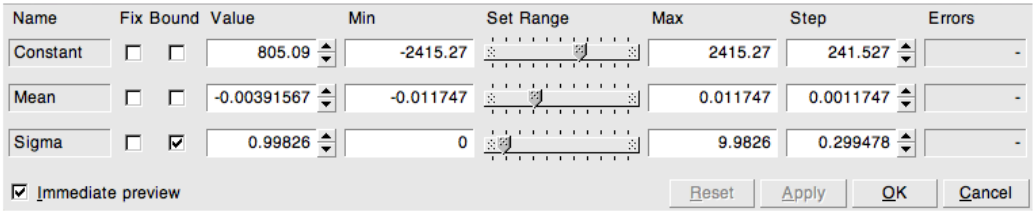
直方图和随机数是统计数据分析中非常重要的工具，整章将专门讨论这一主题。

2.7 与 ROOT 交互

再看一下你的一块图并移动鼠标。 您会注意到这不仅仅是静态图片，因为鼠标指针在触摸绘图上的对象时会改变其形状。 当鼠标悬停在某个对象上时，右键单击会打开一个下拉菜单，在顶行显示您正在处理的 ROOT 类的名称，例如： TCanvas 用于显示窗口本身，TFrame 用于绘图框架，TAxis 用于轴，TPaveText 用于绘图名称。

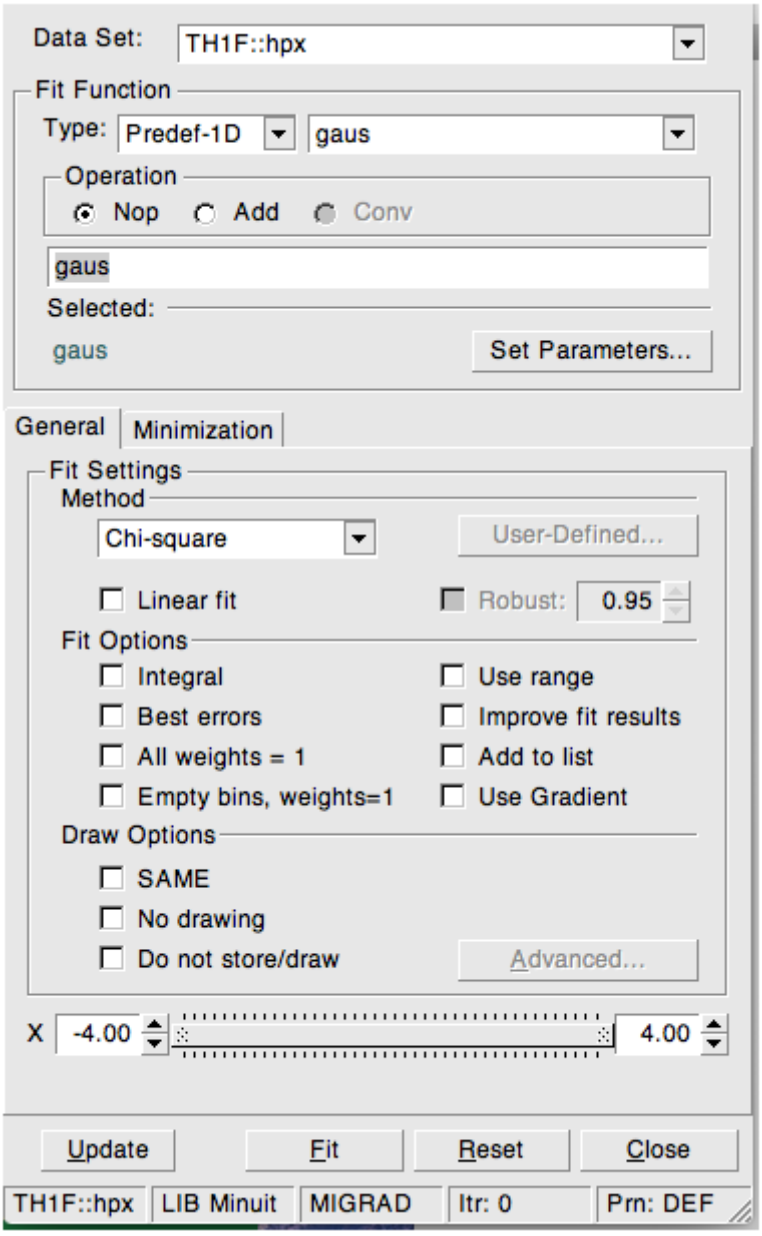
根据您正在调查的绘图，当对相应的图形表示执行右键单击时，将显示 ROOT 类 TF1，TGraphErrors 或 TH1F 的菜单。 菜单项允许直接访问各种类的成员，您甚至可以修改它们，例如 更改轴刻度或标签的颜色和大小，功能线，标记类型等。

试试吧！



交互式 ROOT 面板，用于设置功能参数。

您可能会喜欢以下内容：在示例 slits.C 生成的输出中，右键单击功能行并选择“SetLineAttributes”，然后左键单击“设置参数”。 这样可以访问面板，允许您以交互方式更改功能的参数，如图 2.4 所示。 根据您的喜好，改变狭缝宽度，或从一到两，然后三个或更多狭缝。 单击“应用”时，功能图将更新以反映您已设置的参数的实际值。



作拟合的面板

另一个非常有用的交互式工具是 FitPanel，可用于 TGraphErrors 和 TH1F 类。可以从下拉菜单中选择预定义的拟合函数，包括分别用于高斯和指数函数或 0 到 9 的多项式的“gaus”，“expo”和“pol0”-“pol9”。此外，使用与带参数的函数相同的语法的用户定义函数也是可能的。设置初始参数后，可以执行所选函数与图形或直方图数据的拟合，并在结果图上显示结果。配合面板如图 2.5 所示。拟合面板有许多控制选项，用于选择拟合方法，修复或释放拟合中的各个参数，控制打印在控制台上的输出水平，或提取和显示附加信息，如显示参数相关性的轮廓线。由于函数拟合在任何类型的数据分析中都是最重要的，因此本主题将在以后再次出现。

如果您对您的情节感到满意，您可能想要保存它。只需关闭之前打开的所有选择框，然后从窗口的菜单行中选择菜单项“另存为...”。它将弹出一个文件选择器框，允许您选择格式，文件名和目标目录来存储图像。这里有一个非常明显的功能：您可以将绘图存储为根宏！在

此宏中，您可以找到生成绘图所涉及的所有方法和类的 C ++表示。这是您自己的宏的宝贵信息来源，您希望在完成本教程之后编写这些信息。

使用 ROOT 的交互功能对于初步探索可能性非常有用。您将在本教程中遇到的其他 ROOT 类具有此类图形界面。我们不会对此进行进一步评论，只要知道 ROOT 的交互功能的存在，并在发现它们方便时使用它们。通过大量菜单和参数设置找到自己的方式肯定需要一些反复试验。

2.8 ROOT 初学者常见问题

在指南的这一点上，您可能已经想到了一些基本问题。 我们将尝试解释其中一些，并在下面进一步解释。

2.8.1 数据类型在 ROOT 的声明

在官方ROOT文档中，您可以找到替换正常数据类型的特殊数据类型，例如 Double_t，Float_t 或 Int_t 替换标准的 double，float 或 int 类型。 使用 ROOT 类型可以更轻松地在平台（64/32 位）或操作系统（Windows / Linux）之间移植代码，因为这些类型映射到 ROOT 头文件中的合适类型。 如果需要此类型的自适应代码，请使用 ROOT 类型声明。 但是，通常您不需要这样的自适应代码，并且您可以安全地为您的私有代码使用标准 C 类型声明，正如我们在本指南中所做的那样。但是，如果您打算成为 ROOT 开发人员，则最好遵守官方编码规则！

2.8.2 在启动时配置 ROOT

可以使用.rootrc 文件中的选项来定制 ROOT 会话的行为。 可调参数的示例是与操作和窗口系统，要使用的字体以及启动文件的位置有关的参数。 在启动时，ROOT 按以下顺序查找.rootrc 文件：

```
./.rootrc //local directory
$HOME/.rootrc //user directory
$ROOTSYS/etc/system.rootrc //global ROOT directory
```

如果在上面的搜索路径中找到多个.rootrc 文件，则合并选项，优先级为 local，user，global。该文件的解析和解释由 ROOT 类 TEnv 处理。如果您需要这些相当高级的功能，请查看其文档。文件.rootrc 定义了启动时检查的两个相当重要的文件的位置：rootalias.C 和 rootlogon.C。它们可以包含需要在 ROOT 启动时加载和执行的代码。 rootalias.C 仅加载并最好用于定义一些常用函数。 rootlogon.C 包含将在启动时执行的代码：此文件非常有用，例如为使用 ROOT 创建的绘图预加载自定义样式。这可以通过使用您的首选设置创建一个新的 TStyle 对象来完成，如类参考指南中所述，然后使用命令 gROOT-> SetStyle（“MyStyleName”）;使这个新的样式定义成为默认定义。作为示例，请查看本教程附带的文件 rootlogon.C。另一个相关文件是 rootlogoff.C，它在会话结束时调用。

2.8.3 ROOT 历史命令

在 ROOT 提示符下键入的每个命令都存储在主目录中的.root_hist 文件中。 ROOT 使用此文

件允许使用向上箭头和向下箭头键在命令历史记录中导航。 借助文本编辑器提取成功的 ROOT 命令也可以方便地在您自己的宏中使用。

2.8.4 ROOT 全局指针

ROOT 中的所有全局指针都以小“g”开头。 其中一些已经被隐式引入（例如在启动时配置 ROOT 部分）。 其中最重要的内容如下：
gROOT：gROOT 变量是 ROOT 系统的入口点。 从技术上讲，它是 TROOT 类的一个实例。 使用 gROOT 指针，基本上可以访问基于 ROOT 的程序中创建的每个对象。 TROOT 对象本质上是指向主 ROOT 对象的多个列表的容器。
gStyle：默认情况下，ROOT 创建一个可以通过 gStyle 指针访问的默认样式。 此类包括用于设置以下某些对象属性的函数。
画布
Pad
直方图轴
行
填充区域
文本
标记
功能
直方图统计和标题
等.....
gSystem：定义底层操作系统的通用接口的基类实例，在我们的例子中是 TUnixSystem。
gInterpreter：ROOT 解释器的入口点。 从技术上讲，它是 TCling 单例实例的抽象级别。

此时您已经了解了 ROOT 的一些基本功能。请继续成为专家！

3 ROOT 宏

你知道其他书籍如何继续关于编程基础知识，并最终努力建立一个完整的，有效的程序？让我们跳过这一切。 在本指南中，我们将描述由 ROOT C ++解释器 Cling 执行的宏。

通过向头文件添加一些 include 语句或向任何宏添加一些“修整代码”，编译宏相对容易，无论是作为加载到 ROOT 的预编译库，还是作为独立应用程序。

3.1 宏的简介

如果您有多个行可以在 ROOT 提示符下执行，则可以通过为它们提供与没有扩展名的文件名对应的名称将它们转换为 ROOT 宏。 存储在文件 MacroName.C 中的宏的一般结构是

```
void MacroName() {
       <          ...
         your lines of C++ code
                  ...            >
}
```

通过键入下面的命令来执行宏

```
> root MacroName.C
```

在系统提示符下，或使用.x 执行

```
> root
 root [0] .x MacroName.C
```

在 ROOT 提示下，或者它可以加载到 ROOT 会话中，然后通过键入来执行

```
root [0].L MacroName.C
root [1] MacroName();
```

在 ROOT 提示时，请注意，可以通过这种方式加载多个宏，因为每个宏在 ROOT 名称空间中都有唯一的名称。 一小组选项可以帮助您的情节变得更好。

```
gROOT->SetStyle("Plain");   // set plain TStyle
gStyle->SetOptStat(111111); // draw statistics on plots,
                            // (0) for no output
```

```
gStyle->SetOptFit(1111);    // draw fit results on plot,
                           // (0) for no ouput
gStyle->SetPalette(57);     // set color map
gStyle->SetOptTitle(0);     // suppress title box
   ...
```

接下来，您应该创建一个图形输出画布，其大小，细分和格式适合您的需求，请参阅 TCanvas 类的文档：

```
TCanvas c1("c1","<Title>",0,0,400,300); // create a canvas, specify
position and size in pixels
c1.Divide(2,2); //set subdivisions, called pads
c1.cd(1); //change to pad 1 of canvas c1
```

精心编写的宏的这些部分非常标准，您应该记住包含上面示例中的代码片段，以确保您的绘图始终如您所愿。

下面，在解释和编译部分中，将显示更多代码片段，允许您使用系统编译器编译宏以实现更高效的执行，或将宏转换为与 ROOT 库链接的独立应用程序。

3.2 一个更完整的例子

现在让我们看一个相当完整的数据分析典型任务示例，一个构造带有错误的图形的宏，将（线性）模型拟合到它并将其保存为图像。 要运行此宏，只需在 shell 键入：

```
> root macro1.C
```

代码是围绕 ROOT 类 TGraphErrors 构建的，之前已经介绍过了。 在课程参考指南中查看它，您还可以在其中找到更多示例。 下面显示的宏使用附加类，TF1 定义函数，TCanvas 定义用于我们的绘图的窗口的大小和属性，TLegend 添加一个很好的图例。 目前，忽略头文件的注释 include 语句，它们只会在解释和编译部分的最后重要。

```
// Builds a graph with errors, displays it and saves it as
// image. First, include some header files
// (not necessary for Cling)

#include "TCanvas.h"
#include "TROOT.h"
#include "TGraphErrors.h"
#include "TF1.h"
#include "TLegend.h"
#include "TArrow.h"
```

```
#include "TLatex.h"

void macro1(){
    // The values and the errors on the Y axis
    const int n_points=10;
    double x_vals[n_points]=
           {1,2,3,4,5,6,7,8,9,10};
    double y_vals[n_points]=
           {6,12,14,20,22,24,35,45,44,53};
    double y_errs[n_points]=
           {5,5,4.7,4.5,4.2,5.1,2.9,4.1,4.8,5.43};

    // Instance of the graph
    TGraphErrors graph(n_points,x_vals,y_vals,nullptr,y_errs);
    graph.SetTitle("Measurement XYZ;lenght [cm];Arb.Units");

    // Make the plot estetically better
    graph.SetMarkerStyle(kOpenCircle);
    graph.SetMarkerColor(kBlue);
    graph.SetLineColor(kBlue);

    // The canvas on which we'll draw the graph
    auto  mycanvas = new TCanvas();

    // Draw the graph !
    graph.DrawClone("APE");

    // Define a linear function
    TF1 f("Linear law","[0]+x*[1]",.5,10.5);
    // Let's make the function line nicer
    f.SetLineColor(kRed); f.SetLineStyle(2);
    // Fit it to the graph and draw it
    graph.Fit(&f);
    f.DrawClone("Same");

    // Build and Draw a legend
    TLegend leg(.1,.7,.3,.9,"Lab. Lesson 1");
    leg.SetFillColor(0);
    graph.SetFillColor(0);
    leg.AddEntry(&graph,"Exp. Points");
    leg.AddEntry(&f,"Th. Law");
    leg.DrawClone("Same");

    // Draw an arrow on the canvas
```

```
    TArrow arrow(8,8,6.2,23,0.02,"|>");
    arrow.SetLineWidth(2);
    arrow.DrawClone();

    // Add some text to the plot
    TLatex text(8.2,7.5,"#splitline{Maximum}{Deviation}");
    text.DrawClone();

    mycanvas->Print("graph_with_law.pdf");
}

int main(){
    macro1();
    }
```

让我们详细评论一下：

第 13 行：宏函数中的主函数（它在编译程序中扮演“主”函数的角色）的名称。它必须与没有扩展名的文件名相同。

第 24-25 行：TGraphErrors 类的实例。构造函数获取点的数量和指向 x 值，y 值，x 错误（在这种情况下为 none，由 NULL 指针表示）和 y 错误的数组的指针。第二行一次性定义图形的标题和两个轴的标题，用“;”分隔。

28-30 行：这三行相当直观吗？要更好地理解颜色和样式的枚举器，请参阅 TColor 和 TMarker 类的参考。

第 33 行：将托管绘制对象的画布对象。“内存泄漏”是故意的，以使对象也存在于 macro1 范围之外。

第 36 行：方法 DrawClone 在画布上绘制对象的克隆。它必须是一个克隆，才能在 macro1 的范围之后生存，并在宏执行结束后显示在屏幕上。字符串选项“APE”代表：
A 强加了 Axes 的绘制。

P 强制绘制图形标记。

E 强制绘制图形的误差条。

第 39 行：定义数学函数。有几种方法可以实现这一点，但在这种情况下，构造函数接受函数的名称，公式和函数范围。

第 41 行：maquillage。尝试使用 TLine 类的文档来查看您可以使用的线型。

第 43 行：将 f 函数拟合到图形中，观察指针是否通过。查看屏幕上的输出以查看参数值以及我们将在本指南末尾学习阅读的其他重要信息会更有趣。

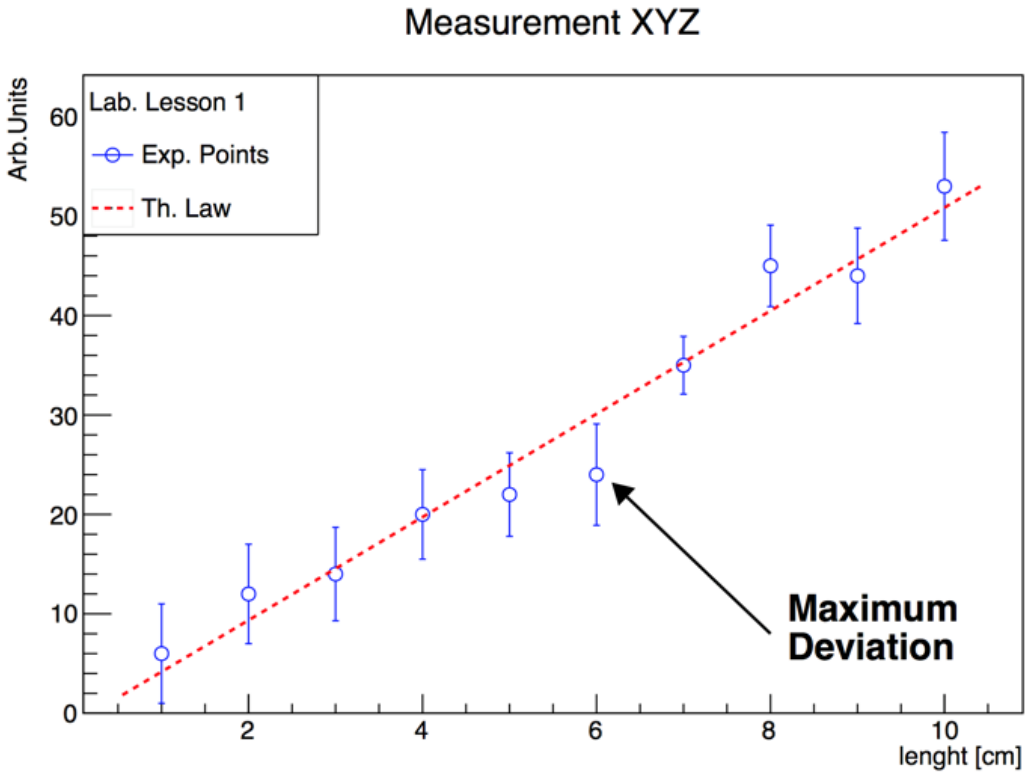
第 44 行：再次在画布上绘制对象的克隆。 “相同”选项可以避免取消已绘制的对象，在我们的示例中是图形。函数 f 将使用先前绘制的图形定义的相同轴系统绘制。

第 47-52 行：使用由 TLegend 实例表示的图例完成绘图。构造函数将相对于画布的总大小（假设为 1）和图例标题字符串的左下角和右上角坐标作为参数。您可以通过 addEntry 方法将先前绘制或未绘制的对象添加到图例中。观察最后绘制的图例：现在看起来很熟悉吧？
第 55-57 行：定义一个右侧有三角形的箭头，厚度为 2 并绘制它。

第 60-61 行：解释一个 Latex 字符串，其左下角位于指定的坐标中。 #splitline {} {}构造允许在同一 TLatex 对象中存储多行。

第 63 行：将画布保存为图像。格式是从文件扩展名自动推断出来的（可能是 eps，gif，...）。

让我们看一下图 3.1 中获得的图。如此小的一堆线条的美丽结局，不是吗？



您的第一个带有数据点的图，一个分析函数的拟合，一个图例和一些图形基元和文本形式的附加信息。 对于读者而言，格式良好的情节对于将结果与读者的相关性进行沟通至关重要。

3.3 图形美化

3.3.1 颜色和图形标记

我们已经看到，要指定颜色，可以为标记，线条，箭头等指定一些标识符，如 kWhite，kRed 或 kBlue。颜色的完整摘要由 ROOT“色轮”表示。 要了解有关完整故事的更多信息，请参阅 TColor 的在线文档。

ROOT 提供了几种图形标记类型。 为点，三角形，十字形或星形中的绘图选择最适合的符号。 可以使用另一组标记的名称。

3.3.2 箭头与直线

宏线 55 显示了如何定义箭头并绘制它。 表示箭头的类是 TArrow，它继承自 TLine。线条和箭头的构造函数始终包含端点的坐标。 箭头还预见参数以指定其形状。 不要低估图中线条和箭头的作用。 由于每个绘图都应包含一条消息，因此使用其他图形基元来强调它是很方便的。

3.3.3 文本

此外，文本在使图表不言自明方面起着重要作用。 TLatex 类提供了在绘图中添加文本的可能性。 该类的对象是使用文本左下角的坐标和包含文本本身的字符串构造的。 真正的转折是普通的乳胶数学符号被自动解释，你只需要用“＃”代替“\”。

如果“\”用作控制字符，则调用 TMathText 接口。 它提供了普通的 TeX 语法，允许访问俄语和日语等字符集。

3.4 解释和编译

正如您所观察到的，到目前为止，我们大量利用 ROOT 的功能来解释我们的代码，而不仅仅是编译然后执行。 这对于广泛的应用程序已经足够了，但您可能已经问过自己“如何编译此代码？”。 有两个答案。

3.4.1 使用 ACLiC 编译宏

除了在第 5-11 行插入适当的头文件外，ACLiC 将为您创建一个为您的宏编译的动态库，无需您的任何努力。 在此示例中，它们已包含在内。 要从解释器类型内部的宏代码生成对象库（请注意“+”）：

```
root [1] .L macro1.C+
```

完成此操作后，宏符号将在内存中可用，您只需通过从解释器内部调用即可执行它：

```
root [2] macro1()
```

3.4.2 用编译器编译宏

有许多优秀的编译器可供选择，包括免费和商业编译器。 我们将在下面引用 GCC 编译器。 在这种情况下，您必须在代码中包含适当的标头，然后利用 root-config 工具自动设置所有编译器标志。root-config 是 ROOT 附带的脚本; 它打印编译代码所需的所有标志和库，并将其与 ROOT 库链接。 为了使代码可以独立运行，需要操作系统的入口点，在 C ++中这是过程 int main（）;. 将 ROOT 宏代码转换为独立应用程序的最简单方法是在宏文件末尾添加以下“修整代码”。 这定义了过程 main，其唯一目的是调用你的宏：

```
int main() {
  ExampleMacro();
  return 0;
}
```

要从名为 ExampleMacro.C 的宏创建独立程序，只需键入即可

```
> g++ -o ExampleMacro ExampleMacro.C `root-config --cflags --libs`
```

并通过键入下面的命令执行它

```
> ./ExampleMacro
```

但是，此过程不允许访问 ROOT 图形，因为既不能控制鼠标或键盘事件，也不能访问 ROOT 的图形窗口。 如果您希望独立应用程序具有显示图形输出并响应鼠标和键盘，则可以使用稍微复杂的代码。 在下面的示例中，宏 ExampleMacro_GUI 由 ROOT 类 TApplication 执行。 作为附加功能，此代码示例提供对从命令行启动时最终传递给程序的参数的访问。 这是代码片段：

```
void StandaloneApplication(int argc, char** argv) {
  // eventually, evaluate the application parameters argc, argv
  // ==>> here the ROOT macro is called
  ExampleMacro_GUI();
}
  // This is the standard "main" of C++ starting
  // a ROOT application
int main(int argc, char** argv) {
   TApplication app("ROOT Application", &argc, argv);
   StandaloneApplication(app.Argc(), app.Argv());
   app.Run();
   return 0;
}
```

用下面命令编译它

```
> g++ -o ExampleMacro_GUI ExampleMacro_GUI `root-config --cflags --ibs`
```

用下面命令执行程序

```
> ./ExampleMacro_GUI
```

4 图表

在本章中，我们将学习如何利用 ROOT 提供的一些功能来显示利用 TGraphErrors 类的数据，这是您之前已经知道的。

4.1 从文件中读取图形点

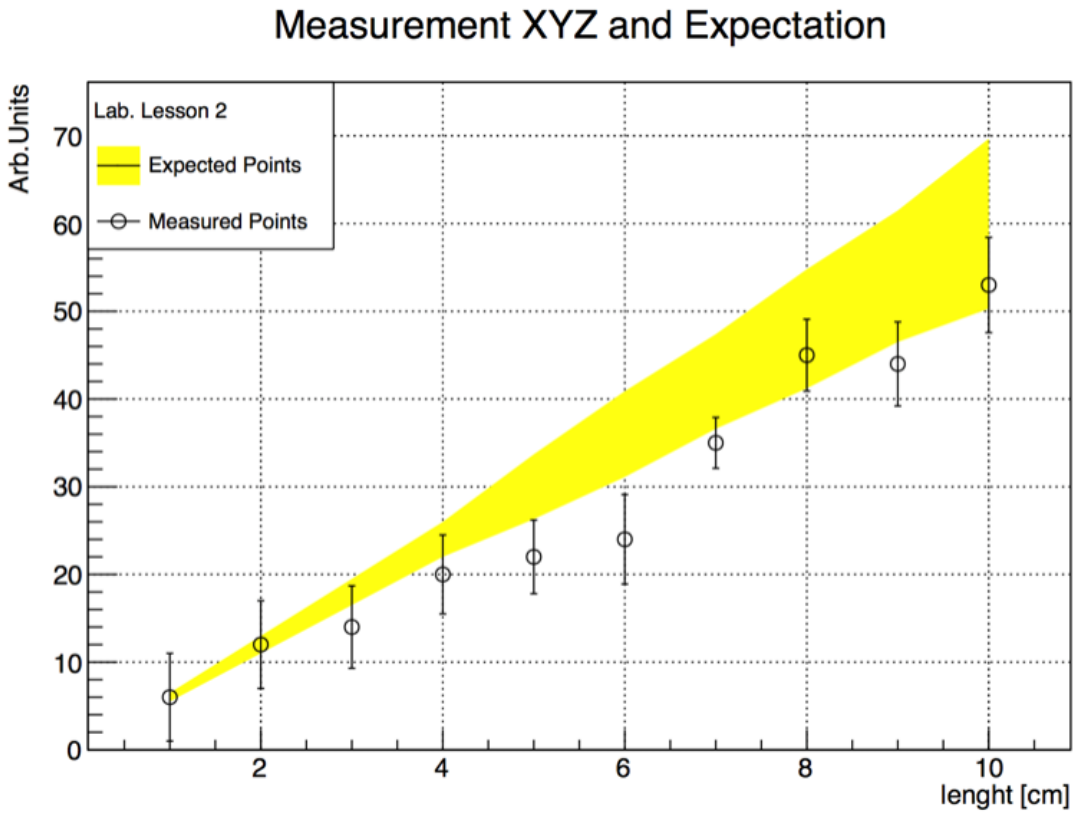
使用实验数据填充图表的最快方法是使用构造函数从 ASCII 文件（即标准文本）格式读取数据点及其误差：

```
TGraphErrors(const char *filename,
const char *format="%lg %lg %lg %lg", Option_t *option="");
```

格式字符串可以是：
“%lg%lg”只读取 2 个第一列到 X，Y
“%lg%lg%lg”只读取 3 个第一列到 X，Y 和 EY 中
“%lg%lg%lg%lg”将 4 个第一列读入 X，Y，EX，EY

该方法具有允许用户将宏重用于许多不同数据集的良好特征。 这是输入文件的示例。显示的漂亮图形结果由下面的宏生成，它读取两个这样的输入文件并使用不同的选项来显示数据点。

```
# Measurement of Friday 26 March
# Experiment 2 Physics Lab

1   6   5
2   12  5
3   14  4.7
4   20  4.5
5   22  4.2
6   24  5.1
7   35  2.9
8   45  4.1
9   44  4.8
10  53  5.43
```




```
// Reads the points from a file and produces a simple graph.
int macro2(){
    auto c=new TCanvas();c->SetGrid();

    TGraphErrors graph_expected("./macro2_input_expected.txt",
                                "%lg %lg %lg");
    graph_expected.SetTitle(
       "Measurement XYZ and Expectation;"
       "lenght [cm];"
       "Arb.Units");
    graph_expected.SetFillColor(kYellow);
    graph_expected.DrawClone("E3AL"); // E3 draws the band

    TGraphErrors graph("./macro2_input.txt","%lg %lg %lg");
    graph.SetMarkerStyle(kCircle);
    graph.SetFillColor(0);
    graph.DrawClone("PESame");

    // Draw the Legend
    TLegend leg(.1,.7,.3,.9,"Lab. Lesson 2");
    leg.SetFillColor(0);
    leg.AddEntry(&graph_expected,"Expected Points");
    leg.AddEntry(&graph,"Measured Points");
    leg.DrawClone("Same");
```

```
    graph.Print();
    return 0;
}
```

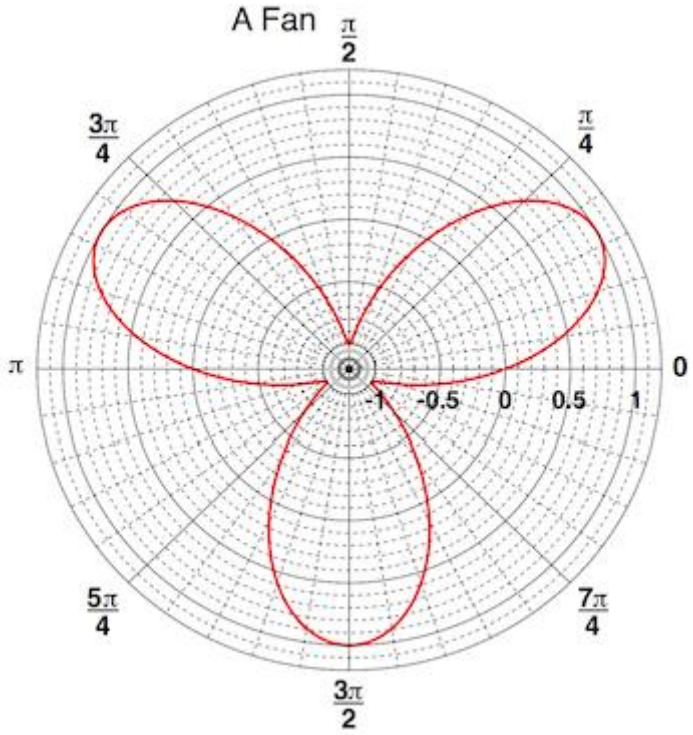
除了检查绘图之外，您还可以随时使用 TGraph :: Print（）方法检查图形的实际内容，获取屏幕上数据点坐标的打印输出。 该宏还向我们展示了如何在图形周围打印彩色条带而不是误差条，这非常有用，例如表示理论预测的误差。

4.2 极坐标图像

除了检查绘图之外，您还可以随时使用 TGraph :: Print（）方法检查图形的实际内容，获取屏幕上数据点坐标的打印输出。 该宏还向我们展示了如何在图形周围打印彩色条带而不是误差条，这非常有用，例如表示理论预测的误差。

```
// Builds a polar graph in a square Canvas.

void macro3(){
    auto c = new TCanvas("myCanvas","myCanvas",600,600);
    Double_t rmin=0.;
    Double_t rmax=TMath::Pi()*6.;
    const Int_t npoints=1000;
    Double_t r[npoints];
    Double_t theta[npoints];
    for (Int_t ipt = 0; ipt < npoints; ipt++) {
        r[ipt] = ipt*(rmax-rmin)/npoints+rmin;
        theta[ipt] = TMath::Sin(r[ipt]);
    }
    TGraphPolar grP1 (npoints,r,theta);
    grP1.SetTitle("A Fan");
    grP1.SetLineWidth(3);
    grP1.SetLineColor(2);
    grP1.DrawClone("L");
}
```

在第 4 行添加了一个新元素，即画布的大小：有时在光学上更好地显示特定画布大小的图。



用 ROOT 获得的极坐标图。

4.3 2D 图表

在特定情况下，绘制一些数量与两个变量可能是有用的，因此创建一个二维图形。 当然，ROOT 可以通过 TGraph2DErrors 类帮助您完成此任务。 下面的宏生成一个表示假设测量的二维图，将二维函数拟合到它，并将其与 x 和 y 投影一起绘制。 将详细解释代码的一些要点。 这次，图表使用随机数填充数据点，引入一个新的非常重要的成分，即使用 Mersenne Twister 算法的 ROOT TRandom3 随机数发生器（Matsumoto 1997）。

```
// Create, Draw and fit a TGraph2DErrors
void macro4(){
   gStyle->SetPalette(kBird);
   const double e = 0.3;
   const int nd = 500;

   TRandom3 my_random_generator;
   TF2 f2("f2",
          "1000*(([0]*sin(x)/x)*([1]*sin(y)/y))+200",
          -6,6,-6,6);
   f2.SetParameters(1,1);
   TGraph2DErrors dte(nd);
   // Fill the 2D graph
   double rnd, x, y, z, ex, ey, ez;
```

```
   for (Int_t i=0; i<nd; i++) {
      f2.GetRandom2(x,y);
      // A random number in [-e,e]
      rnd = my_random_generator.Uniform(-e,e);
      z = f2.Eval(x,y)*(1+rnd);
      dte.SetPoint(i,x,y,z);
      ex = 0.05*my_random_generator.Uniform();
      ey = 0.05*my_random_generator.Uniform();
      ez = fabs(z*rnd);
      dte.SetPointError(i,ex,ey,ez);
   }
   // Fit function to generated data
   f2.SetParameters(0.7,1.5);  // set initial values for fit
   f2.SetTitle("Fitted 2D function");
   dte.Fit(&f2);
   // Plot the result
   auto c1 = new TCanvas();
   f2.SetLineWidth(1);
   f2.SetLineColor(kBlue-5);
   TF2   *f2c = (TF2*)f2.DrawClone("Surf1");
   TAxis *Xaxis = f2c->GetXaxis();
   TAxis *Yaxis = f2c->GetYaxis();
   TAxis *Zaxis = f2c->GetZaxis();
   Xaxis->SetTitle("X Title"); Xaxis->SetTitleOffset(1.5);
   Yaxis->SetTitle("Y Title"); Yaxis->SetTitleOffset(1.5);
   Zaxis->SetTitle("Z Title"); Zaxis->SetTitleOffset(1.5);
   dte.DrawClone("P0 Same");
   // Make the x and y projections
   auto c_p= new TCanvas("ProjCan",
                         "The Projections",1000,400);
   c_p->Divide(2,1);
   c_p->cd(1);
   dte.Project("x")->Draw();
   c_p->cd(2);
   dte.Project("y")->Draw();
}
```

让我们逐步了解代码，了解发生了什么：

第 3 行：这将调色板颜色代码设置为比默认颜色更好的颜色代码。注释这一行试一试。本文提供有关颜色映射选择的更多详细信息。

第 7 行：随机生成器的实例。然后，您可以根据不同的概率密度函数绘制出这个实例随机数，例如第 27-29 行的 Uniform one。请参阅在线文档以了解此 ROOT 功能的全部功能。

第 8 行：您已经熟悉 TF1 类了。这是它的二维版本。在第 16 行，根据 TF2 公式分配的两个随机数用方法 TF2 :: GetRandom2（double＆a，double＆b）绘制。

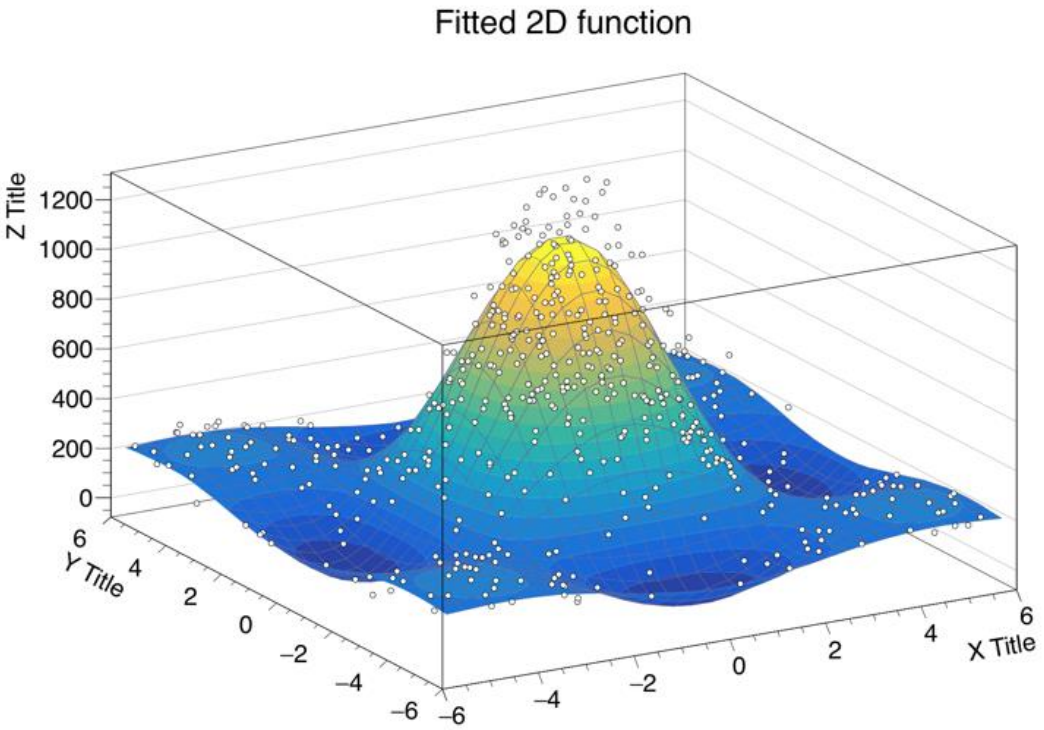
第 27-29 行：拟合二维函数就像在一维情况下一样，即参数的初始化和调用 Fit（）方法。

第 34 行：Surf1 选项将 TF2 对象（以及二维直方图）绘制为彩色表面，在三维画布上使用线框。见图 4.3。

第 35-40 行：检索轴指针并定义轴标题。

第 41 行：在彩色表面上绘制点云。

第 43-49 行：在这里，您将学习如何创建画布，将其分为两个子垫并访问它们。在同一窗口或图像中显示多个图非常方便。



绘制有二维函数的数据集，可视化为彩色表面。

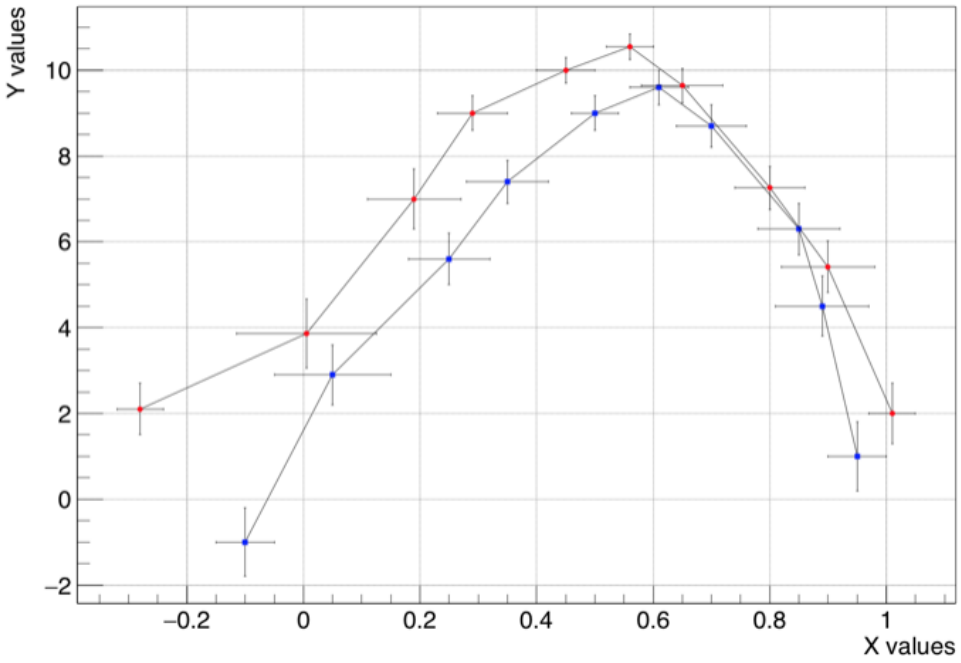
4.4 多个图表

TMultigraph 类允许将一组图操作为单个实体。 它是 TGraph（或派生）对象的集合。绘制时，将自动计算 X 和 Y 轴范围，以便所有图形都可见。

```
// Manage several graphs as a single entity.
void multigraph(){
   TCanvas *c1 = new TCanvas("c1","multigraph",700,500);
   c1->SetGrid();

   TMultiGraph *mg = new TMultiGraph();

   // create first graph
   const Int_t n1 = 10;
```

```
   Double_t px1[] = {-0.1, 0.05, 0.25, 0.35, 0.5,
0.61,0.7,0.85,0.89,0.95};
   Double_t py1[] = {-1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};
   Double_t ex1[] = {.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
   Double_t ey1[] = {.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
   TGraphErrors *gr1 = new TGraphErrors(n1,px1,py1,ex1,ey1);
   gr1->SetMarkerColor(kBlue);
   gr1->SetMarkerStyle(21);
   mg->Add(gr1);

   // create second graph
   const Int_t n2 = 10;
   Float_t x2[]  = {-0.28, 0.005, 0.19, 0.29, 0.45,
0.56,0.65,0.80,0.90,1.01};
   Float_t y2[]  = {2.1,3.86,7,9,10,10.55,9.64,7.26,5.42,2};
   Float_t ex2[] = {.04,.12,.08,.06,.05,.04,.07,.06,.08,.04};
   Float_t ey2[] = {.6,.8,.7,.4,.3,.3,.4,.5,.6,.7};
   TGraphErrors *gr2 = new TGraphErrors(n2,x2,y2,ex2,ey2);
   gr2->SetMarkerColor(kRed);
   gr2->SetMarkerStyle(20);
   mg->Add(gr2);

   mg->Draw("apl");
   mg->GetXaxis()->SetTitle("X values");
   mg->GetYaxis()->SetTitle("Y values");

   gPad->Update();
   gPad->Modified();
}
```

第 6 行创建了多图。
第 9-28 行：创建两个有错误的图形并将它们添加到多图中。
第 30-32 行：绘制多图。 轴限制是自动计算的，以确保所有图形的点都在范围内。



在多图中分组的一组图表。

[^ 3] https://root.cern.ch/drupal/content/rainbow-color-map

5 直方图

直方图在任何类型的物理分析中都发挥着重要作用，不仅可以显示测量结果，还可以作为强大的数据缩减形式。 ROOT 提供了许多代表直方图的类，所有类都继承自 TH1 类。 我们将在本章中重点讨论单位和二维直方图，其中 bin 的内容由浮点数表示，分别为 TH1F 和 TH2F 类。

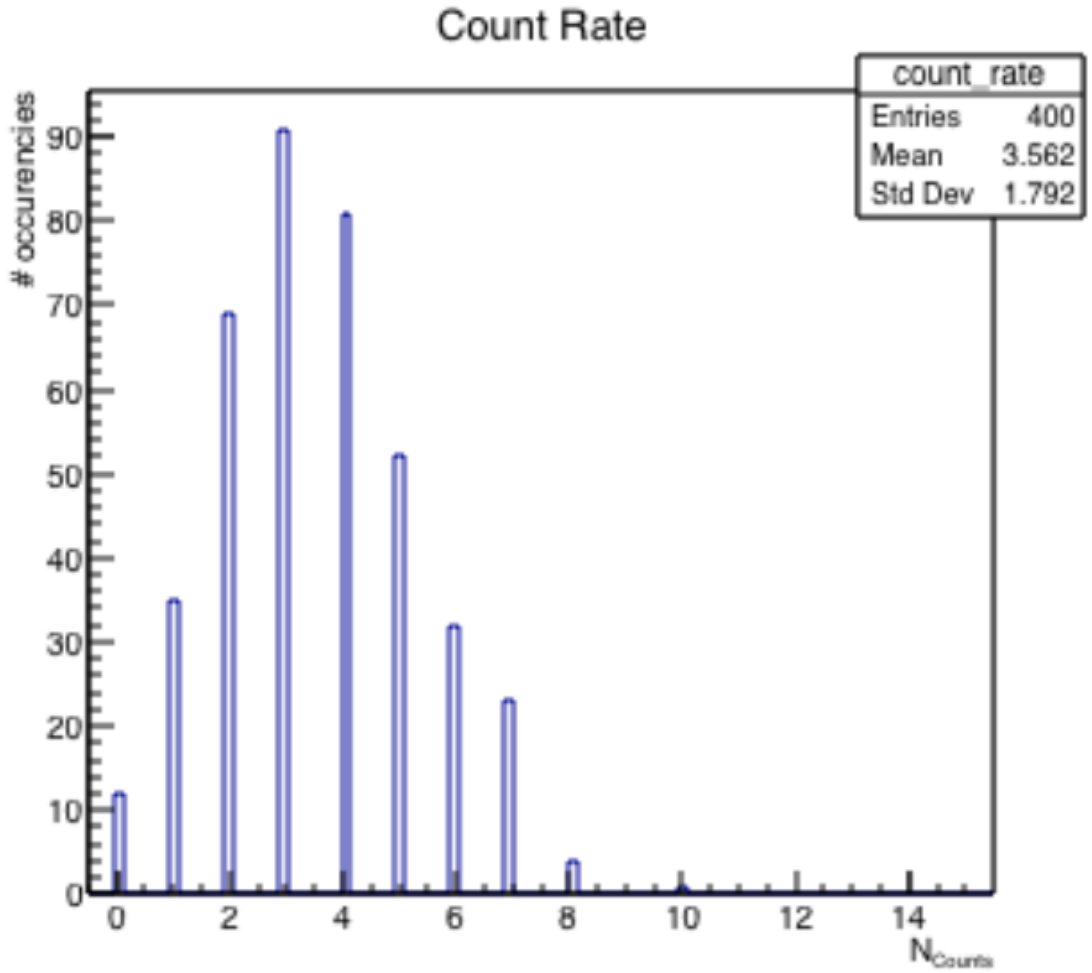
5.1 你的第一个直方图

假设您想要测量在给定时间间隔内位于放射源附近的盖革检测器的计数。 这可以让您了解源的活动。 在这种情况下，计数分布是泊松分布。 让我们看看如何使用以下示例宏来填充和绘制直方图。

```
// Create, Fill and draw an Histogram which reproduces the
// counts of a scaler linked to a Geiger counter.

void macro5(){
    auto cnt_r_h=new TH1F("count_rate",
                "Count Rate;N_{Counts};# occurencies",
                100, // Number of Bins
                -0.5, // Lower X Boundary
                15.5); // Upper X Boundary

    auto mean_count=3.6f;
    TRandom3 rndgen;
    // simulate the measurements
    for (int imeas=0;imeas<400;imeas++)
        cnt_r_h->Fill(rndgen.Poisson(mean_count));

    auto c= new TCanvas();
    cnt_r_h->Draw();

    auto c_norm= new TCanvas();
    cnt_r_h->DrawNormalized();

    // Print summary
    cout << "Moments of Distribution:\n"
         << " - Mean     = " << cnt_r_h->GetMean() << " +- "
                             << cnt_r_h->GetMeanError() << "\n"
         << " - Std Dev  = " << cnt_r_h->GetStdDev() << " +- "
                             << cnt_r_h->GetStdDevError() << "\n"
         << " - Skewness = " << cnt_r_h->GetSkewness() << "\n"
```

```
        << " - Kurtosis = " << cnt_r_h->GetKurtosis() << "\n";
}
```

这画出了以下的图（图 5.1）：



计数（伪）实验的结果。 在给定泊松分布的离散性质的情况下，仅填充对应于整数值的区间。

使用直方图非常简单。 与示例中出现的图形的主要区别在于：

第 5 行：直方图从一开始就有一个名称和一个标题，没有预定义的条目数，但是有多个箱和一个较低的上限。

第 15 行：通过 TH1F :: Fill 方法将条目存储在直方图中。

第 18 行和第 21 行：直方图也可以标准化，ROOT 自动处理必要的重新缩放。

第 24 到 30 行：这个小片段显示了访问直方图的时刻和相关错误是多么容易。

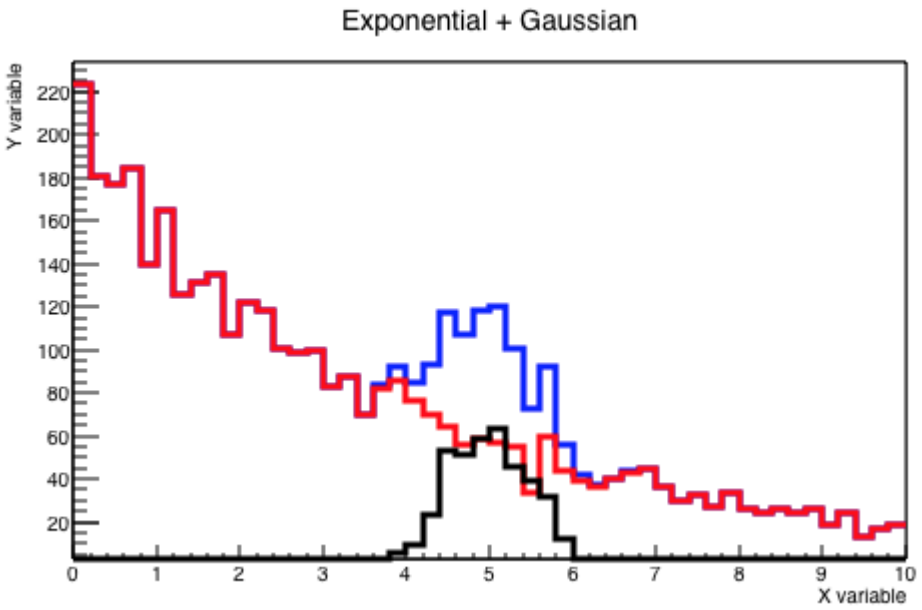
5.2 组合与拆分直方图

使用直方图可以执行大量操作。 最有用的是加法和除法。 在下面的宏中，我们将学习如何在 ROOT 中管理这些过程。

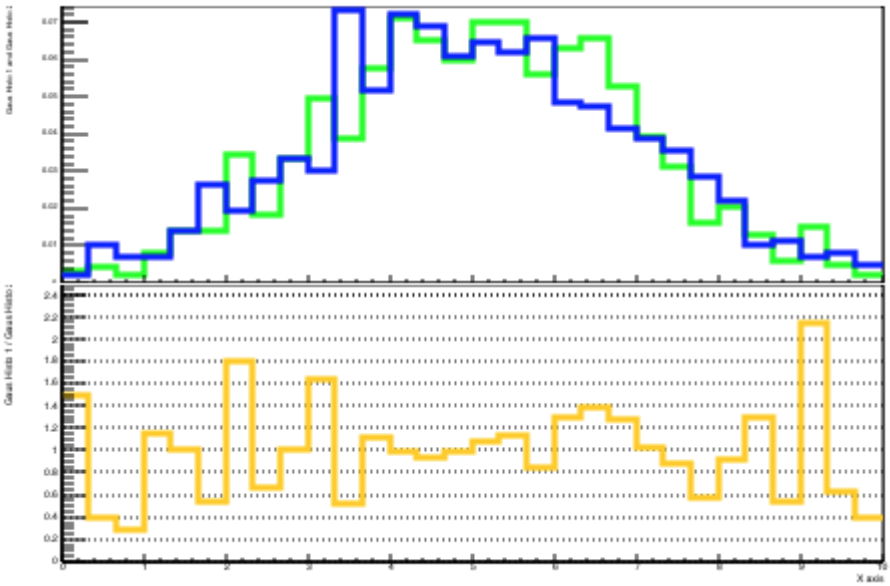
```
// Divide and add 1D Histograms

void format_h(TH1F* h, int linecolor){
    h->SetLineWidth(3);
    h->SetLineColor(linecolor);
    }

void macro6(){

    auto sig_h=new TH1F("sig_h","Signal Histo",50,0,10);
    auto gaus_h1=new TH1F("gaus_h1","Gauss Histo 1",30,0,10);
    auto gaus_h2=new TH1F("gaus_h2","Gauss Histo 2",30,0,10);
    auto bkg_h=new TH1F("exp_h","Exponential Histo",50,0,10);

    // simulate the measurements
    TRandom3 rndgen;
    for (int imeas=0;imeas<4000;imeas++){
        bkg_h->Fill(rndgen.Exp(4));
        if (imeas%4==0) gaus_h1->Fill(rndgen.Gaus(5,2));
        if (imeas%4==0) gaus_h2->Fill(rndgen.Gaus(5,2));
        if (imeas%10==0)sig_h->Fill(rndgen.Gaus(5,.5));}

    // Format Histograms
    int i=0;
    for (auto hist : {sig_h,bkg_h,gaus_h1,gaus_h2})
        format_h(hist,1+i++);

    // Sum
    auto sum_h= new TH1F(*bkg_h);
    sum_h->Add(sig_h,1.);
    sum_h->SetTitle("Exponential + Gaussian;X variable;Y
variable");
    format_h(sum_h,kBlue);

    auto c_sum= new TCanvas();
    sum_h->Draw("hist");
    bkg_h->Draw("SameHist");
    sig_h->Draw("SameHist");

    // Divide
    auto dividend=new TH1F(*gaus_h1);
    dividend->Divide(gaus_h2);

    // Graphical Maquillage
```

```
    dividend->SetTitle(";X axis;Gaus Histo 1 / Gaus Histo 2");
    format_h(dividend,kOrange);
    gaus_h1->SetTitle(";;Gaus Histo 1 and Gaus Histo 2");
    gStyle->SetOptStat(0);

    TCanvas* c_divide= new TCanvas();
    c_divide->Divide(1,2,0,0);
    c_divide->cd(1);
    c_divide->GetPad(1)->SetRightMargin(.01);
    gaus_h1->DrawNormalized("Hist");
    gaus_h2->DrawNormalized("HistSame");

    c_divide->cd(2);
    dividend->GetYaxis()->SetRangeUser(0,2.49);
    c_divide->GetPad(2)->SetGridy();
    c_divide->GetPad(2)->SetRightMargin(.01);
    dividend->Draw();
}
```

您将获得的图显示在图 5.2 和 5.3 中。



两个直方图的总和



两个直方图的比较

有些行现在需要说明一下：

第 3 行：正如我们所知，Cling 也能够为每个文件解释多个函数。 在这种情况下，该函数只是设置一些参数来方便地设置直方图线。

第 19 到 21 行：条件语句的某些 C ++语法用于在循环内用不同数量的条目填充直方图。

第 30 行：两个直方图的总和。 可以为负的权重可以分配给添加的直方图。

第 41 行：两个直方图的划分相当简单。

第 44 到 62 行：当您绘制两个数量及其比率时，如果所有信息都汇总在一个单独的图中，那就更好了。 这些行提供了执行此操作的框架。

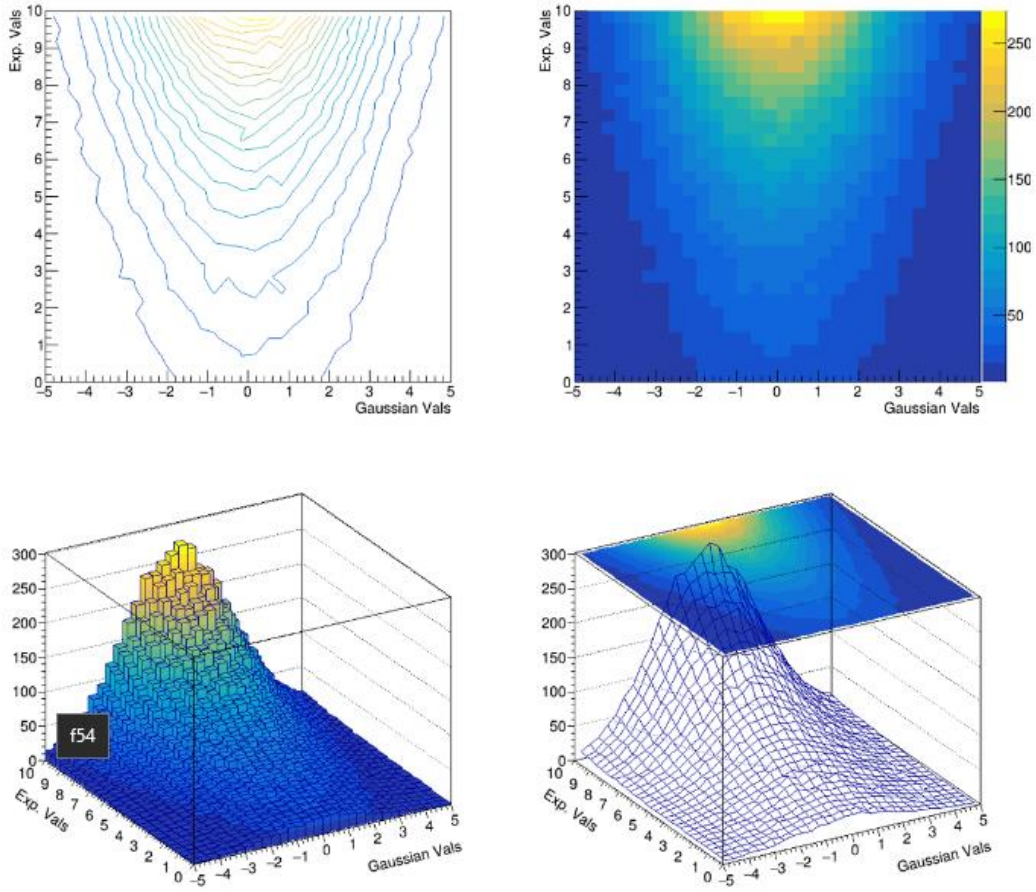
5.3 二维直方图

二维直方图是非常有用的工具，例如用于检查变量之间的相关性。 您可以通过简单的方式利用 ROOT 提供的二维直方图类。 让我们看看这个宏如何：

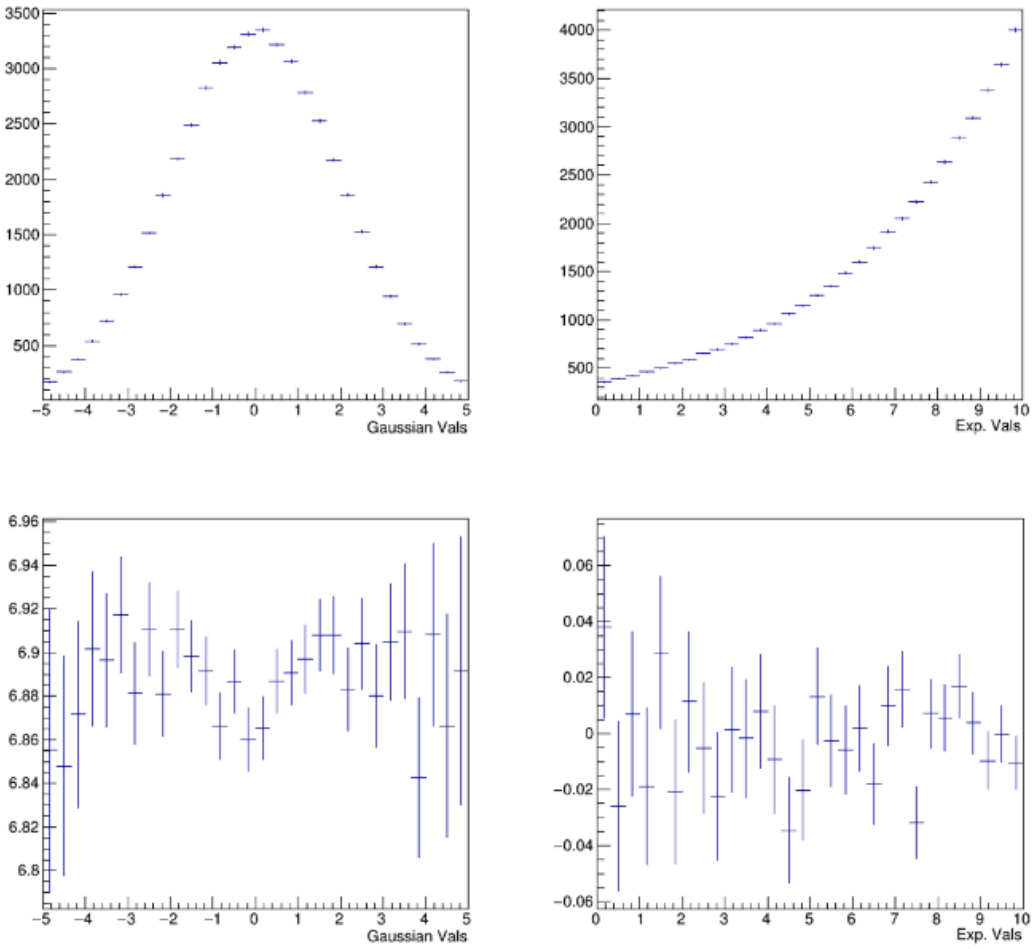
```
// Draw a Bidimensional Histogram in many ways
// together with its profiles and projections

void macro7(){
   gStyle->SetPalette(kBird);
   gStyle->SetOptStat(0);
   gStyle->SetOptTitle(0);

   auto bidi_h = new TH2F("bidi_h","2D Histo;Gaussian Vals;Exp. Vals",
                          30,-5,5,  // X axis
                          30,0,10); // Y axis
```

```
   TRandom3 rgen;
   for (int i=0;i<500000;i++) {
      bidi_h->Fill(rgen.Gaus(0,2),10-rgen.Exp(4),.1);
   }

   auto c=new TCanvas("Canvas","Canvas",800,800);
   c->Divide(2,2);
   c->cd(1); bidi_h->Draw("Cont1");
   c->cd(2); bidi_h->Draw("Colz");
   c->cd(3); bidi_h->Draw("Lego2");
   c->cd(4); bidi_h->Draw("Surf3");

   // Profiles and Projections
   auto c2=new TCanvas("Canvas2","Canvas2",800,800);
   c2->Divide(2,2);
   c2->cd(1); bidi_h->ProjectionX()->Draw();
   c2->cd(2); bidi_h->ProjectionY()->Draw();
   c2->cd(3); bidi_h->ProfileX()->Draw();
   c2->cd(4); bidi_h->ProfileY()->Draw();
}
```

代码中提供了两种图，第一种包含三维表示（图 5.4），第二种包含二维直方图的投影和轮廓（图 5.5）。



表示二维直方图的不同方式。



二维直方图的投影和轮廓

当沿 x（y）方向执行投影时，对于沿 x（y）轴的每个 bin，沿 y（x）轴的所有 bin 内容被相加（图 5.5 的曲线上方）。 当沿 x（y）方向执行轮廓时，对于沿 x（y）轴的每个仓，沿 y（x）的所有仓内容的平均值与其 RMS 一起计算并显示为具有误差的符号 bar（图 5.5 的下两个图）。

变量之间的相关性通过方法 Double_t GetCovariance（）和 Double_t GetCorrelationFactor（）来量化。

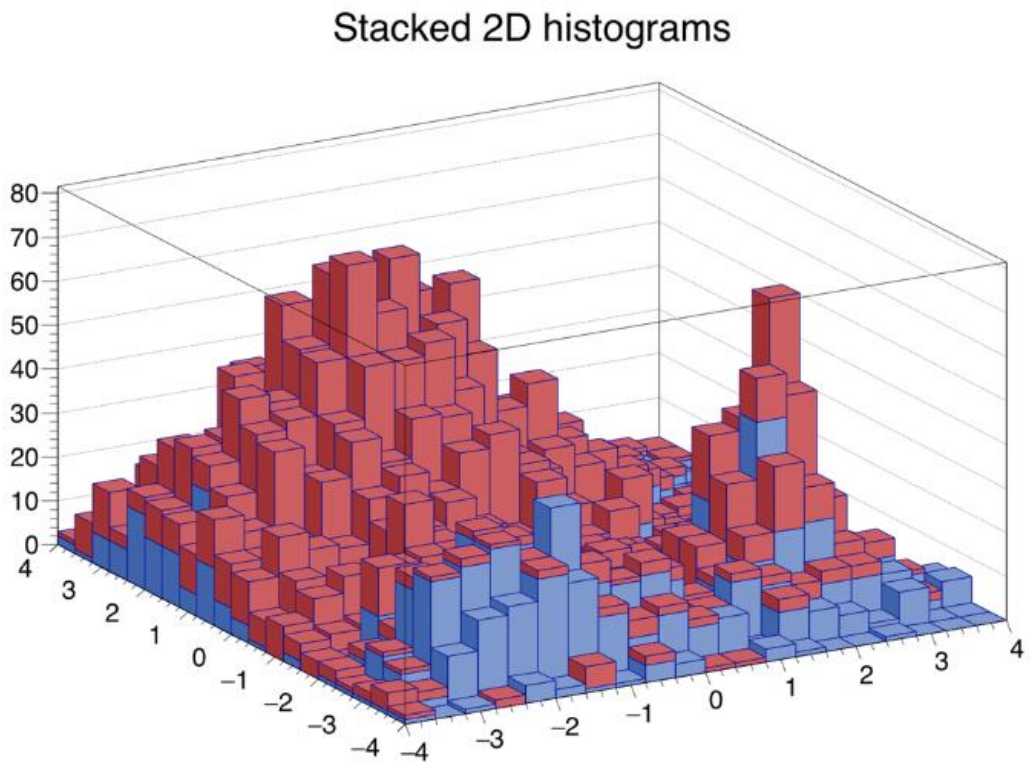
5.4 多个直方图

类 THStack 允许将一组直方图操作为单个实体。 它是 TH1（或派生）对象的集合。 绘制时，将自动计算 X 和 Y 轴范围，例如所有直方图都可见。 多个绘图选项可用于 1D 和 2D 直方图。 下一个宏显示了它如何查找 2D 直方图：

```
// Example of stacked histograms using the class THStack

void hstack(){
   THStack *a = new THStack("a","Stacked 2D histograms");
```

```
   TF2 *f1 = new TF2("f1","xygaus + xygaus(5) + xylandau(10)",-
4,4,-4,4);
   Double_t params1[] = {130,-1.4,1.8,1.5,1, 150,2,0.5,-2,0.5,
3600,-2,0.7,-3,0.3};
   f1->SetParameters(params1);
   TH2F *h2sta = new TH2F("h2sta","h2sta",20,-4,4,20,-4,4);
   h2sta->SetFillColor(38);
   h2sta->FillRandom("f1",4000);

   TF2 *f2 = new TF2("f2","xygaus + xygaus(5)",-4,4,-4,4);
   Double_t params2[] = {100,-1.4,1.9,1.1,2, 80,2,0.7,-2,0.5};
   f2->SetParameters(params2);
   TH2F *h2stb = new TH2F("h2stb","h2stb",20,-4,4,20,-4,4);
   h2stb->SetFillColor(46);
   h2stb->FillRandom("f2",3000);

   a->Add(h2sta);
   a->Add(h2stb);

   a->Draw();
}
```

第 4 行：创建堆栈。
第 4-18 行：创建要在堆栈中添加的两个直方图。
第 20-21 行：在堆栈中添加直方图。
第 23 行：将堆栈绘制为乐高图。 颜色区分两个直方图 5.6。



两个 2D 直方图堆叠在一起

6 函数拟合与参数估计

在完成前面的章节之后，您已经知道如何使用分析函数（类 TF1），并且您可以深入了解数据可视化的图形（TGraphErrors）和直方图类（TH1F）。在本章中，我们将通过将函数拟合到数据来为先前的近似解释添加更多细节，以面对参数估计的基本主题。对于图形和直方图，ROOT 提供了一个易于使用的界面来执行拟合 - 图形界面的拟合面板或 Fit 方法。TFitResult 类允许访问详细结果。

通常有必要研究分析程序的统计特性。通过将分析应用于许多模拟数据集（或“伪数据”），每个模拟数据代表真实实验的一个可能版本，这是最容易实现的。如果模拟仅处理在数据中观察到的最终分布，并且不执行基础物理和实验装置的完整模拟，则经常使用名称“Toy Monte Carlo”.3 因为所有参数的真实值都是已知的在伪数据中，来自分析程序的参数估计值之间的差异可以确定真实值，并且还可以检查分析过程是否提供了正确的误差估计。

6.1 用函数拟合伪随机数

在下面的示例中，生成伪数据集并将模型拟合到其中。

ROOT 提供各种最小化算法，以最小化 chi2 或负对数似然函数。 默认的最小化程序是 MINUIT，最初是用 FORTRAN 编程语言实现的。 还有一个 C ++版本，MINUIT2，以及 Fumili（Silin 1983），这是一种针对拟合优化的算法。 可以使用 ROOT :: Math :: MinimizerOptions 类的静态函数选择最小化算法。 最小化器的转向选项，例如收敛容差或最

大函数调用次数，也可以使用此类的方法进行设置。 所有当前实现的最小化器都记录在 ROOT 的参考文档中：请查看 ROOT :: Math :: Minimizer 类文档。 以下代码的复杂程度有意地略高于前面的示例。 宏的图形输出如图 6.1 所示：

```
void format_line(TAttLine* line,int col,int sty){
    line->SetLineWidth(5); line->SetLineColor(col);
    line->SetLineStyle(sty);}

double the_gausppar(double* vars, double* pars){
    return pars[0]*TMath::Gaus(vars[0],pars[1],pars[2])+
        pars[3]+pars[4]*vars[0]+pars[5]*vars[0]*vars[0];}

int macro8(){
    gStyle->SetOptTitle(0); gStyle->SetOptStat(0);
    gStyle->SetOptFit(1111); gStyle->SetStatBorderSize(0);
    gStyle->SetStatX(.89); gStyle->SetStatY(.89);

    TF1 parabola("parabola","[0]+[1]*x+[2]*x**2",0,20);
    format_line(&parabola,kBlue,2);

    TF1 gaussian("gaussian","[0]*TMath::Gaus(x,[1],[2])",0,20);
    format_line(&gaussian,kRed,2);

    TF1 gausppar("gausppar",the_gausppar,-0,20,6);
    double a=15; double b=-1.2; double c=.03;
    double norm=4; double mean=7; double sigma=1;
    gausppar.SetParameters(norm,mean,sigma,a,b,c);
    gausppar.SetParNames("Norm","Mean","Sigma","a","b","c");
    format_line(&gausppar,kBlue,1);

    TH1F histo("histo","Signal plus background;X vals;Y
Vals",50,0,20);
    histo.SetMarkerStyle(8);

    // Fake the data
    for (int i=1;i<=5000;++i) histo.Fill(gausppar.GetRandom());

    // Reset the parameters before the fit and set
    // by eye a peak at 6 with an area of more or less 50
    gausppar.SetParameter(0,50);
    gausppar.SetParameter(1,6);
    int npar=gausppar.GetNpar();
    for (int ipar=2;ipar<npar;++ipar)
gausppar.SetParameter(ipar,1);
```

```
    // perform fit ...
    auto fitResPtr = histo.Fit(&gausppar, "S");
    // ... and retrieve fit results
    fitResPtr->Print(); // print fit results
    // get covariance Matrix an print it
    TMatrixDSym covMatrix (fitResPtr->GetCovarianceMatrix());
    covMatrix.Print();

    // Set the values of the gaussian and parabola
    for (int ipar=0;ipar<3;ipar++){
        gaussian.SetParameter(ipar,gausppar.GetParameter(ipar));
        parabola.SetParameter(ipar,gausppar.GetParameter(ipar+3));
    }

    histo.GetYaxis()->SetRangeUser(0,250);
    histo.DrawClone("PE");
    parabola.DrawClone("Same"); gaussian.DrawClone("Same");
    TLatex latex(2,220,"#splitline{Signal Peak over}{background}");
    latex.DrawClone("Same");
    return 0;
}
```

有必要逐步解释一下：

第 1-3 行：简化线条构成的简单功能。请记住，类 TF1 继承自 TAttLine。

第 5-7 行：定制函数的定义，即高斯（“信号”）加上抛物线函数，即“背景”。

第 10-12 行：Canvas 的一些化妆品。特别是我们希望拟合的参数在图上非常清晰和美观。

第 20-25 行：定义并初始化 TF1 的实例。

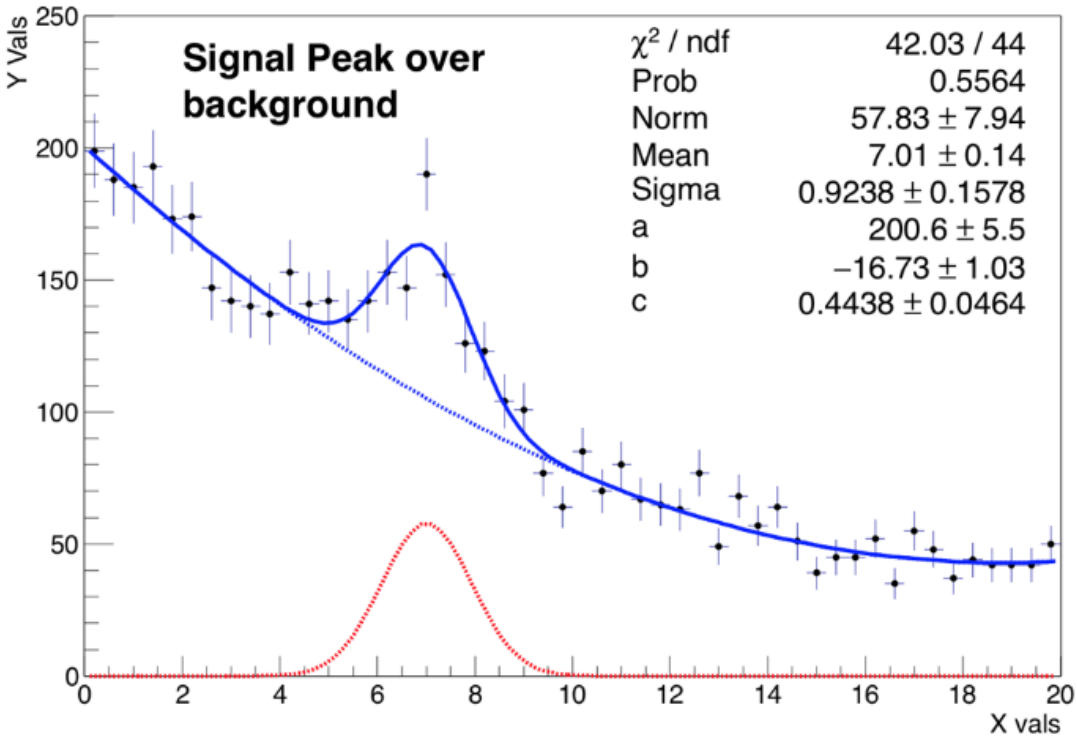
第 27-31 行：定义并填充直方图。

第 33-38 行：为方便起见，在拟合中使用与生成伪数据相同的功能;因此，我们需要重置功能参数。这部分代码对于每个拟合过程都非常重要，因为它设置拟合的初始值。

第 41 行：一个非常简单的命令，现在众所周知：将函数拟合到直方图。

第 42-46 行：从拟合中检索输出。在这里，我们只需打印拟合结果并访问并打印参数的协方差矩阵。

第 54 行结束：以最佳拟合值绘制伪数据，拟合函数以及信号和背景分量。



适合伪数据：背景趋势上的信号形状。 此图是另一个示例，说明如何使图表“不言自明”可以帮助您更好地显示结果。

6.2 Toy Monte Carlo Experiments

让我们看一个玩具实验的简单例子，比较两种方法以使函数适合直方图，χ^2 方法和称为“分箱对数似然拟合”，两者都在 ROOT 中可用。

作为一个非常简单但功能强大的数量来检查拟合结果的质量，我们为每个伪数据集构造所谓的“拉”，即参数的估计值和真值之差，归一化为估计误差 关于参数，

$$\frac{p_{estim} - p_{true}}{\sigma_p}$$ 。

如果一切正常，则拉值的分布是标准正态分布，即以零为中心，标准偏差为 1 的高斯分布。

该宏执行相当多的玩具实验，其中直方图用高斯分布数重复填充，表示该示例中的伪数据。 每次，根据所选择的方法执行拟合，并计算拉力并将其填充到直方图中。 这是代码：

```
// Toy Monte Carlo example.
// Check pull distribution to compare chi2 and binned
// log-likelihood methods.

void pull( int n_toys = 10000,
     int n_tot_entries = 100,
```

```
  int nbins = 40,
  bool do_chi2=true ){

TString method_prefix("Log-Likelihood ");
if (do_chi2)
    method_prefix="#chi^{2} ";

// Create histo
TH1F h4(method_prefix+"h4",
       method_prefix+" Random Gauss",
       nbins,-4,4);
h4.SetMarkerStyle(21);
h4.SetMarkerSize(0.8);
h4.SetMarkerColor(kRed);

// Histogram for sigma and pull
TH1F sigma(method_prefix+"sigma",
          method_prefix+"sigma from gaus fit",
          50,0.5,1.5);
TH1F pull(method_prefix+"pull",
         method_prefix+"pull from gaus fit",
         50,-4.,4.);

// Make nice canvases
auto c0 = new TCanvas(method_prefix+"Gauss",
                      method_prefix+"Gauss",0,0,320,240);
c0->SetGrid();

// Make nice canvases
auto c1 = new TCanvas(method_prefix+"Result",
                      method_prefix+"Sigma-Distribution",
                      0,300,600,400);
c0->cd();

float sig, mean;
for (int i=0; i<n_toys; i++){
 // Reset histo contents
    h4.Reset();
 // Fill histo
    for ( int j = 0; j<n_tot_entries; j++ )
    h4.Fill(gRandom->Gaus());
 // perform fit
    if (do_chi2) h4.Fit("gaus","q"); // Chi2 fit
    else h4.Fit("gaus","lq"); // Likelihood fit
```

```
    // some control output on the way
       if (!(i%100)){
           h4.Draw("ep");
           c0->Update();}

    // Get sigma from fit
       TF1 *fit = h4.GetFunction("gaus");
       sig = fit->GetParameter(2);
       mean= fit->GetParameter(1);
       sigma.Fill(sig);
       pull.Fill(mean/sig * sqrt(n_tot_entries));
      } // end of toy MC loop
    // print result
       c1->cd();
       pull.DrawClone();
}

void macro9(){
    int n_toys=10000;
    int n_tot_entries=100;
    int n_bins=40;
    cout << "Performing Pull Experiment with chi2 \n";
    pull(n_toys,n_tot_entries,n_bins,true);
    cout << "Performing Pull Experiment with Log Likelihood\n";
    pull(n_toys,n_tot_entries,n_bins,false);
}
```

您目前对 ROOT 的了解应该足以理解宏背后的所有技术细节。 请注意，第 61 行中的变量 pull 不同于上面的定义：使用平均值的参数误差，使用分布的拟合标准偏差除以条目数 sig / sqrt（n_tot_entries）的平方根。

什么方法使用默认参数表现出更好的性能？

如果将每个直方图的条目数增加十倍，会发生什么？ 为什么？

这些问题的答案超出了本指南的范围。 基本上所有关于统计方法的书籍都提供了对上述主题的完整处理。

7 文件 I/O 和并行计算

7.1 存储 ROOT 对象

ROOT 提供了将磁盘上的类实例写入 ROOT 文件的可能性（有关详细信息，请参阅 TFile 类）。 一个人说通过将对象存储在磁盘上使对象变得“持久”。 当读回文件时，对象在内存中重建。 满足执行特定类的实例的 I / O 的要求是 ROOT 类型系统知道该类的存储器中的

布局。 本主题超出了本文档的范围：值得一提的是，对于几乎完整的 ROOT 类集，I/O 可以开箱即用。

我们可以使用直方图和两个简单的宏来探索这个功能。

```
void write_to_file(){

    // Instance of our histogram
    TH1F h("my_histogram","My Title;X;# of entries",100,-5,5);

    // Let's fill it randomly
    h.FillRandom("gaus");

    // Let's open a TFile
    TFile out_file("my_rootfile.root","RECREATE");

    // Write the histogram in the file
    h.Write();

    // Close the file
    out_file.Close();
}
```

不错，嗯？ 特别是对于一种不像 C ++本身那样预见持久性的语言。 即使磁盘上存在具有相同名称的文件，RECREATE 选项也会强制 ROOT 创建新文件。

现在，您可以使用 Cling 命令行访问文件中的信息并绘制以前编写的直方图：

```
>  root my_rootfile.root
root [0]
Attaching file my_rootfile.root as _file0...
root [1] _file0->ls()
TFile**     my_rootfile.root
 TFile*     my_rootfile.root
  KEY: TH1F my_histogram;1 My Title
root [2] my_histogram->Draw()
```

或者，您可以使用一个简单的宏来执行该作业：

```
void read_from_file(){

    // Let's open the TFile
    TFile in_file("my_rootfile.root");

    // Get the Histogram out
```

```
    TH1F* h;
    in_file.GetObject("my_histogram",h);

    // Draw it
    auto myCanvas = new TCanvas();
    h->DrawClone();
}
```

7.2 ROOT 的 N-tuples

7.2.1 存储简单的 N-tuples

到目前为止，我们已经看到了如何操作从 ASCII 文件读取的输入。 通过自己的 n-tuple 类，ROOT 提供了比这更好的可能性。 人们可以引用这些类提供的众多优势

- 优化的磁盘 I / O.
- 存储许多 n-tuple 行的可能性。
- 在 ROOT 文件中写入 n-tuple。
- 使用 TBrowser 进行交互式检查。
- 不仅存储数字，还存储列中的对象。

在本节中，我们将简要讨论 TNtuple 类，它是 TTree 类的简化版本。 ROOT 的 TNtuple 对象可以存储多个 float 条目。 让我们按照通常的策略来评论一个最小的例子

```
// Fill an n-tuple and write it to a file simulating measurement of
// conductivity of a material in different conditions of pressure
// and temperature.

void write_ntuple_to_file(){

   TFile ofile("conductivity_experiment.root","RECREATE");

   // Initialise the TNtuple
   TNtuple cond_data("cond_data",
                     "Example N-Tuple",
                     "Potential:Current:Temperature:Pressure");

   // Fill it randomly to fake the acquired data
   TRandom3 rndm;
   float pot,cur,temp,pres;
   for (int i=0;i<10000;++i){
      pot=rndm.Uniform(0.,10.);      // get voltage
      temp=rndm.Uniform(250.,350.);  // get temperature
      pres=rndm.Uniform(0.5,1.5);    // get pressure
```

```
      cur=pot/(10.+0.05*(temp-300.)-0.2*(pres-1.)); // current
                                                    // add some random
smearing (measurement errors)
      pot*=rndm.Gaus(1.,0.01); // 1% error on voltage
      temp+=rndm.Gaus(0.,0.3); // 0.3 abs. error on temp.
      pres*=rndm.Gaus(1.,0.02);// 1% error on pressure
      cur*=rndm.Gaus(1.,0.01); // 1% error on current
                               // write to ntuple
      cond_data.Fill(pot,cur,temp,pres);
   }

   // Save the ntuple and close the file
   cond_data.Write();
   ofile.Close();
}
```

在统计意义上，写入此示例 n-tuple 的数据表示三个独立变量（电势或电压，压力和温度），以及一个根据非常简单的法则依赖于其他变量的变量（电流）和另外的高斯变量拖尾现象。 这组变量模拟了在改变压力和温度的同时测量电阻。

想象一下，你的任务现在在于找到变量之间的关系 - 当然不知道用于生成变量的代码。 您将看到 NTuple 类的可能性使您能够执行此分析任务。 在交互式会话中打开上面宏编写的 ROOT 文件（cond_data.root），并使用 TBrowser 以交互方式检查它：

```
root[0] TBrowser b
```

你会发现你的 n-tuple 的列被写成叶子。 只需单击它们即可获得变量的直方图！
接下来，分别在 shell 提示符和交互式 ROOT shell 中尝试以下命令：

```
> root conductivity_experiment.root
Attaching file conductivity_experiment.root as _file0...
root [0] cond_data->Draw("Current:Potential")
```

您刚刚用一行代码生成了相关图！

例如，尝试扩展语法类型

```
root [1] cond_data->Draw("Current:Potential","Temperature<270")
```

你得到了什么？

现在试试

```
root [2] cond_data->Draw("Current/Potential:Temperature")
```

从这些例子中可以清楚地看到如何在这样的变量的多维空间中导航并使用 n-tuple 揭示变量之间的关系。

7.2.2 读取 N-tuples

为了完整性，您可以在这里找到一个小宏来从 ROOT 的 n-tuple 中读取数据

```
// Read the previously produced N-Tuple and print on screen
// its content

void read_ntuple_from_file(){

    // Open a file, save the ntuple and close the file
    TFile in_file("conductivity_experiment.root");
    TNtuple* my_tuple;in_file.GetObject("cond_data",my_tuple);
    float pot,cur,temp,pres; float* row_content;

    cout << "Potential\tCurrent\tTemperature\tPressure\n";
    for (int irow=0;irow<my_tuple->GetEntries();++irow){
        my_tuple->GetEntry(irow);
        row_content = my_tuple->GetArgs();
        pot = row_content[0];
        cur = row_content[1];
        temp = row_content[2];
        pres = row_content[3];
        cout << pot << "\t" << cur << "\t" << temp
             << "\t" << pres << endl;
        }

    }
```

宏显示了访问 n-tuple 内容的最简单方法：在加载 n-tuple 后，将其分支分配给变量，GetEntry（long）自动使用特定行的内容填充它们。 通过这样做，可以拆分读取 n-tuple 的逻辑和处理它的代码，并且源代码保持清晰。

7.2.3 存储任意类型的 N-tuples

也可以使用 ROOT 的 TBranch 类编写任意类型的 n-tuple。这一点尤其重要，因为 TNtuple::Fill（）只接受浮点数。 以下宏创建与以前相同的 n-tuple，但直接预订分支。 然后，Fill（）函数将连接变量的当前值填充到树中。

```
// Fill an n-tuple and write it to a file simulating measurement of
// conductivity of a material in different conditions of pressure
// and temperature using branches.

void write_ntuple_to_file_advanced(
   const std::string& outputFileName="conductivity_experiment.root"
   ,unsigned int numDataPoints=1000000){

   TFile ofile(outputFileName.c_str(),"RECREATE");

   // Initialise the TNtuple
   TTree cond_data("cond_data", "Example N-Tuple");

   // define the variables and book them for the ntuple
   float pot,cur,temp,pres;
   cond_data.Branch("Potential", &pot, "Potential/F");
   cond_data.Branch("Current", &cur, "Current/F");
   cond_data.Branch("Temperature", &temp, "Temperature/F");
   cond_data.Branch("Pressure", &pres, "Pressure/F");

   for (int i=0;i<numDataPoints;++i){
      // Fill it randomly to fake the acquired data
      pot=gRandom->Uniform(0.,10.)*gRandom->Gaus(1.,0.01);
      temp=gRandom->Uniform(250.,350.)+gRandom->Gaus(0.,0.3);
      pres=gRandom->Uniform(0.5,1.5)*gRandom->Gaus(1.,0.02);
      cur=pot/(10.+0.05*(temp-300.)-0.2*(pres-1.))*
                  gRandom->Gaus(1.,0.01);
      // write to ntuple
      cond_data.Fill();}

   // Save the ntuple and close the file
   cond_data.Write();
   ofile.Close();
}
```

Branch（）函数需要指向变量的指针和变量类型的定义。 下表列出了一些可能的值。 请注意，ROOT 不会检查输入，错误可能会导致严重问题。 如果将值读作另一种类型而不是它们已被写入，则这尤其成立。 将变量存储为 float 并将其读取为 double 时。
可用于在 ROOT 中定义分支类型的变量类型列表：

type	size	C++	identifier
signed integer	32 bit	int	I
	64 bit	long	L
unsigned integer	32 bit	unsigned int	i
	64 bit	unsigned long	l
floating point	32 bit	float	F
	64 bit	double	D
boolean	-	bool	O

7.2.4 处理跨文件的 n-tuple

通常， n-tuple 或树跨越许多文件，并且很难手动添加它们。 因此，ROOT 以 TChain 的形式提供帮助类。 它的用法如下面的宏所示，它与前面的例子非常相似。 TChain 的构造函数将 TTree（或 TNuple）的名称作为参数。 这些文件添加了 Add（fileName）函数，其中一个也可以使用外卡，如示例所示。

```
// Read several previously produced N-Tuples and print on screen its
// content.
//
// you can easily create some files with the following statement:
//
// for i in 0 1 2 3 4 5; \\
// do root -l -x -b -q \\
// "write_ntuple_to_file.cxx \\
// (\"conductivity_experiment_${i}.root\", 100)"; \\
//  done

void read_ntuple_with_chain(){
   // initiate a TChain with the name of the TTree to be processed
   TChain in_chain("cond_data");
   in_chain.Add("conductivity_experiment*.root"); // add files,
                                                  // wildcards work

   // define variables and assign them to the corresponding branches
   float pot, cur, temp, pres;
   in_chain.SetBranchAddress("Potential", &pot);
```

```
   in_chain.SetBranchAddress("Current", &cur);
   in_chain.SetBranchAddress("Temperature", &temp);
   in_chain.SetBranchAddress("Pressure", &pres);

   cout << "Potential\tCurrent\tTemperature\tPressure\n";
   for (size_t irow=0; irow<in_chain.GetEntries(); ++irow){
      in_chain.GetEntry(irow); // loads all variables that have
                               // been connected to branches
      cout << pot << "\t" << cur << "\t" << temp <<
                        "\t" << pres << endl;
   }
}
```

7.2.5 *对于进阶用户：使用选择器脚本处理树*

通过 TChain :: Process（）方法提供了另一种处理 TChain 的非常通用且强大的方法。 此方法将 TSelector 类型的 -user-implemented-类的实例作为参数，并且 - 可选 - 将条目数和要处理的第一个条目作为参数。 类 TSelector 的模板由 TTree :: MakeSelector 方法提供，如下面的小宏 makeSelector.C 所示。

它打开上面例子中的 n-tupleconductivity_experiment.root，并从中创建头文件 MySelector.h 和一个模板，用于插入自己的分析代码 MySelector.C。

```
{
// create template class for Selector to run on a tree
//////////////////////////////////////////////////////
//
// open root file containing the Tree
    TFile f("conductivity_experiment.root");
// create TTree object from it
    TTree *t; f.GetObject("cond_data",t);
// this generates the files MySelector.h and MySelector.C
    t->MakeSelector("MySelector");
}
```

该模板包含在处理 TChain 开始之前调用的入口点 Begin（）和 SlaveBegin（），为链的每个条目调用 Process（），以及在处理完最后一个条目之后调用的 SlaveTerminate（）和 Terminate（）。 通常，在 SlaveBegin（）中执行直方图预订的初始化，分析，即条目的选择，计算和直方图的填充，在 Process（）中完成，最终操作如结果的绘图和存储在 SlaveTerminate（）中进行 或终止（）。

只有在启用了通过 PROOF 或 PROOF lite 的并行处理时，才会在所谓的从节点上调用入口点 SlaveBegin（）和 SlaveTerminate（），如下所述。

宏 MySelector.C 中显示了一个选择器类的简单示例。 该示例使用以下命令序列执行：

```
> TChain *ch=new TChain("cond_data", "Chain for Example N-Tuple");
> ch->Add("conductivity_experiment*.root");
> ch->Process("MySelector.C+");
```

像往常一样，附加到要执行的宏名称的“+”启动 MySelector.C 与系统编译器的编译，以提高性能。

MySelector.C 中的代码（如下面的清单所示）在 SlaveBegin（）中预览了一些直方图，并将它们添加到实例 fOutput 中，该实例属于 TList 类.4 Terminate（）中的最终处理允许访问直方图和存储， 显示或将其保存为图片。 这通过 TList fOutput 在示例中显示。 有关详细信息，请参阅下面评论的列表; 大多数文本实际上是由 TTree :: MakeSelector 自动生成的注释。

```
#define MySelector_cxx
// The class definition in MySelector.h has been generated
automatically
// by the ROOT utility TTree::MakeSelector(). This class is derived
// from the ROOT class TSelector. For more information on the TSelector
// framework see $ROOTSYS/README/README.SELECTOR or the ROOT User
Manual.

// The following methods are defined in this file:
//    Begin():       called every time a loop on the tree starts,
//                   a convenient place to create your histograms.
//    SlaveBegin():   called after Begin(), when on PROOF called only on
the
//                   slave servers.
//    Process():     called for each event, in this function you decide
what
//                   to read and fill your histograms.
//    SlaveTerminate: called at the end of the loop on the tree, when on
PROOF
//                   called only on the slave servers.
//    Terminate():    called at the end of the loop on the tree,
//                   a convenient place to draw/fit your histograms.
//
// To use this file, try the following session on your Tree T:
//
// root> T->Process("MySelector.C")
// root> T->Process("MySelector.C","some options")
// root> T->Process("MySelector.C+")
//
```

```
#include "MySelector.h"
#include <TH2.h>
#include <TStyle.h>


void MySelector::Begin(TTree * /*tree*/)
{
   // The Begin() function is called at the start of the query.
   // When running with PROOF Begin() is only called on the client.
   // The tree argument is deprecated (on PROOF 0 is passed).

   TString option = GetOption();

}

void MySelector::SlaveBegin(TTree * /*tree*/)
{
   // The SlaveBegin() function is called after the Begin() function.
   // When running with PROOF SlaveBegin() is called on each slave
server.
   // The tree argument is deprecated (on PROOF 0 is passed).

   TString option = GetOption();

}

Bool_t MySelector::Process(Long64_t entry)
{
   // The Process() function is called for each entry in the tree (or
possibly
   // keyed object in the case of PROOF) to be processed. The entry
argument
   // specifies which entry in the currently loaded tree is to be
processed.
   // It can be passed to either MySelector::GetEntry() or
TBranch::GetEntry()
   // to read either all or the required parts of the data. When
processing
   // keyed objects with PROOF, the object is already loaded and is
available
   // via the fObject pointer.
   //
   // This function should contain the "body" of the analysis. It can
contain
```

```
   // simple or elaborate selection criteria, run algorithms on the data
   // of the event and typically fill histograms.
   //
   // The processing can be stopped by calling Abort().
   //
   // Use fStatus to set the return value of TTree::Process().
   //
   // The return value is currently not used.

   return kTRUE;
}

void MySelector::SlaveTerminate()
{
   // The SlaveTerminate() function is called after all entries or
objects
   // have been processed. When running with PROOF SlaveTerminate() is
called
   // on each slave server.

}

void MySelector::Terminate()
{
   // The Terminate() function is the last function to be called during
   // a query. It always runs on the client, it can be used to present
   // the results graphically or save the results to file.

}
```

7.2.6 对于进阶用户：使用 PROOF lite 进行多核处理

通过 TCein :: Process（）类型 TSelector 的选择器函数处理 n-tuple，如上一节末尾所述，特别是对于非常大的数据集提供了额外的优势：在分布式系统或多核上 在体系结构中，数据的一部分可以并行处理，从而大大减少了执行时间。 在启用了多核 CPU 或硬件线程的现代计算机上，由于使用了所有可用的 CPU 功率，因此可以更快地完成分析周转。

在分布式系统上，必须设置 PROOF 服务器和工作节点，如 ROOT 文档中详细描述的那样。 在具有多个核心的单台计算机上，可以使用 PROOF lite。 尝试使用以下小宏 RunMySelector.C，它包含两个额外的行，与上面的示例相比较（根据 CPU 核心数调整 worker 数）：

```
{// set up a TChain
TChain *ch=new TChain("cond_data", "My Chain for Example N-Tuple");
 ch->Add("conductivity_experiment*.root");
// eventually, start Proof Lite on cores
TProof::Open("workers=4");
ch->SetProof();
ch->Process("MySelector.C+");}
```

第一个命令 TProof :: Open（const char *）启动一个本地 PROOF 服务器（如果没有指定参数，将使用所有核心），命令 ch-> SetProof（）;使用 PROOF 可以处理链条。现在，当发出命令 ch-> Process（“MySelector.C +）;”时，MySelector.C 中的代码在每个从节点上编译并执行。方法 Begin（）和 Terminate（）仅在 master 上执行。分析 n-tuple 文件列表，并将部分数据分配给可用的从属进程。在 SlaveBegin（）中预订的直方图存在于从属节点的进程中，并相应地填充。终止时，PROOF 主机收集来自从属的直方图并合并它们。在 Terminate（）中，所有合并的直方图都可用，可以检查，分析或存储。直方图通过每个从属进程中 TList 类的实例 fOutput 处理，并且可以在此列表中检索合并终止。

为了探索这种机制的强大功能，使用存储任意 N-TUPLE 的部分中的脚本生成一些非常大的 n-tuple - 你可以尝试 10 000 000 个事件（这会产生一个大约 160 MByte 的大 n-tuple）。您还可以生成大量文件并使用通配符将其添加到 TChain。现在执行：> root -l RunMySelector.C 并观察会发生什么：

```
Processing RunMySelector.C...
 +++ Starting PROOF-Lite with 4 workers +++
Opening connections to workers: OK (4 workers)
Setting up worker servers: OK (4 workers)
PROOF set to parallel mode (4 workers)

Info in <TProofLite::SetQueryRunning>: starting query: 1
Info in <TProofQueryResult::SetRunning>: nwrks: 4
Info in <TUnixSystem::ACLiC>: creating shared library
                         ~/DivingROOT/macros/MySelector_C.so
*==* ----- Begin of Job ----- Date/Time = Wed Feb 15 23:00:04 2012
Looking up for exact location of files: OK (4 files)
Looking up for exact location of files: OK (4 files)
Info in <TPacketizerAdaptive::TPacketizerAdaptive>:
                    Setting max number of workers per node to 4
Validating files: OK (4 files)
Info in <TPacketizerAdaptive::InitStats>:
                    fraction of remote files 1.000000
Info in <TCanvas::Print>:
      file ResistanceDistribution.png has been created
*==* ----- End of Job ----- Date/Time = Wed Feb 15 23:00:08 2012
Lite-0: all output objects have been merged
```

整个处理链的日志文件保存在每个工作节点的~.proof 目录中。 这对于调试或出现问题非常有用。 由于此处描述的方法也可以在不使用 PROOF 的情况下工作，因此分析脚本的开发工作可以在标准方式上对一小部分数据进行，并且仅对于完整处理，可以通过 PROOF 使用并行性。

值得提醒读者的是，典型数据分析程序的速度受 I / O 速度的限制（例如，从硬盘读取数据所隐含的延迟）。 因此，预计使用任何并行分析工具包都不能消除这种限制。

7.2.7 关于 N-tuples 的优化

ROOT 自动在 n-tuple 上应用压缩算法以减少内存消耗。在大多数情况下相同的值将仅消耗磁盘上的小空间（但必须在读取时解压缩）。然而，一旦处理时间超过几分钟，你应该考虑你的 n-tuple 的设计和你的分析。

尽量保持你的 n-tuple 简单并使用适当的变量类型。如果您的测量精度有限，则无需以双精度存储。

每次测量不会改变的实验条件应存储在单独的树中。虽然压缩可以处理冗余值，但处理时间会随着每个必须填充的变量而增加。

函数 SetCacheSize(long)指定用于从文件读取 TTree 对象的缓存大小。默认值为 30MB。手动增加可能在某些情况下有所帮助。请注意，缓存机制每个 TFile 对象只能覆盖一个 TTree 对象。

您可以使用 AddBranchToCache 选择要由缓存算法覆盖的分支，并使用 SetBranchStatus 停用不需要的分支。这种机制可以显着加快对具有许多分支的树的简单操作。

您可以使用 TTreePerfStats 轻松测量性能。此类的 ROOT 文档还包括一个介绍性示例。例如，TTreePerfStats 可以向您显示分别存储元数据和有效负载数据是有益的，即在工作结束时将元数据树批量写入文件，而不是将两个树交错写入。

8 ROOT in Python

ROOT 提供了通过一组名为 PyROOT 的绑定与 Python 接口的可能性。 Python 用于各种应用领域，是当今最常用的脚本语言之一。 在 PyROOT 的帮助下，可以将脚本语言的强大功能与 ROOT 工具结合起来。Python 的介绍材料可从网上的许多来源获得，参见 e。G。 http://docs.python.org。

8.1 PyROOT

除了 Python 的特殊语言功能外，对 PyROOT 中 ROOT 类及其方法的访问几乎与 C ++宏相同，最重要的是在赋值时的动态类型声明。 回到我们的第一个例子，简单地在 ROOT 中绘制一个函数，以下 C ++代码：

```
TF1 *f1 = new TF1("f2","[0]*sin([1]*x)/x",0.,10.);
f1->SetParameter(0,1);
```

```
f1->SetParameter(1,1);
f1->Draw();
```

在 Python 中成为：

```
import ROOT
f1 = ROOT.TF1("f2","[0]*sin([1]*x)/x",0.,10.)
f1.SetParameter(0,1);
f1.SetParameter(1,1);
f1.Draw();
```

稍微更高级的示例将宏中定义的数据移交给 ROOT 类 TGraphErrors。 请注意，Python 数组可用于在 Python 和 ROOT 之间传递数据。 Python 脚本中的第一行允许它直接从操作系统执行，而无需从 python 或强烈推荐的强大交互式 shell ipython 启动脚本。 python 脚本中的最后一行是允许您在脚本终止后消失之前查看 ROOT 画布中的图形输出。

这是 C ++版本：

```
void TGraphFit(){
   //
   // Draw a graph with error bars and fit a function to it
   //
   gStyle->SetOptFit(111) ; //superimpose fit results
   // make nice Canvas
   TCanvas *c1 = new TCanvas("c1" ,"Daten" ,200 ,10 ,700 ,500) ;
   c1->SetGrid( ) ;
   //define some data points ...
   const Int_t n = 10;
   Float_t x[n] = {-0.22, 0.1, 0.25, 0.35, 0.5, 0.61, 0.7, 0.85, 0.89,
1.1};
   Float_t y[n] = {0.7, 2.9, 5.6, 7.4, 9., 9.6, 8.7, 6.3, 4.5, 1.1};
   Float_t ey[n] = {.8 ,.7 ,.6 ,.5 ,.4 ,.4 ,.5 ,.6 ,.7 ,.8};
   Float_t ex[n] = {.05 ,.1 ,.07 ,.07 ,.04 ,.05 ,.06 ,.07 ,.08 ,.05};
   // and hand over to TGraphErros object
   TGraphErrors *gr = new TGraphErrors(n,x,y,ex,ey);
   gr->SetTitle("TGraphErrors with Fit") ;
   gr->Draw("AP");
   // now perform a fit (with errors in x and y!)
   gr->Fit("gaus");
   c1->Update();
}
```

在 Python 中它看起来像这样：

```
#
# Draw a graph with error bars and fit a function to it
#
from ROOT import gStyle, TCanvas, TGraphErrors
from array import array
gStyle.SetOptFit (111) # superimpose fit results
c1=TCanvas("c1" ,"Data" ,200 ,10 ,700 ,500) #make nice
c1.SetGrid ()
#define some data points . . .
x = array('f', (-0.22, 0.1, 0.25, 0.35, 0.5, 0.61, 0.7, 0.85, 0.89,
1.1) )
y = array('f', (0.7, 2.9, 5.6, 7.4, 9., 9.6, 8.7, 6.3, 4.5, 1.1) )
ey = array('f', (.8 ,.7 ,.6 ,.5 ,.4 ,.4 ,.5 ,.6 ,.7 ,.8) )
ex = array('f', (.05 ,.1 ,.07 ,.07 ,.04 ,.05 ,.06 ,.07 ,.08 ,.05) )
nPoints=len ( x )
# . . . and hand over to TGraphErros object
gr=TGraphErrors ( nPoints , x , y , ex , ey )
gr.SetTitle("TGraphErrors with Fit")
gr.Draw ( "AP" ) ;
gr.Fit("gaus")
c1.Update ()
# request user action before ending (and deleting graphics window)
raw_input('Press <ret> to end -> ')
```

比较这两个示例中的 C ++和 Python 版本，现在应该清楚将 C ++中的任何 ROOT 宏转换为 Python 版本是多么容易。

再举一个例子，让我们重温第 4 章中的 macro3。一个依赖于 ROOT 类 TMath 的直接 Python 版本：

```
# Builds a polar graph in a square Canvas.

from ROOT import TGraphPolar, TCanvas, TMath
from array import array

c = TCanvas("myCanvas","myCanvas",600,600)
rmin = 0.
rmax = TMath.Pi()*6.
npoints = 300
r = array('d',[0]*npoints)
theta  = array('d',[0]*npoints)
for ipt in xrange(0,npoints):
    r[ipt] = ipt*(rmax-rmin)/npoints+rmin
    theta[ipt] = TMath.Sin(r[ipt])
```

```
grP1 = TGraphPolar(npoints,r,theta)
grP1.SetTitle("A Fan")
grP1.SetLineWidth(3)
grP1.SetLineColor(2)
grP1.DrawClone("L")
raw_input("Press enter to exit.")
```

8.1.1 More Python- less C++

您可能已经注意到，有一些 Python 模块提供类似于 ROOT 类的功能，它们更适合您的 Python 代码。

上述 macro3 的更“pythonic”版本将使用 ROOT 类 TMath 的替换来向 TGraphPolar 提供数据。 使用数学包，代码的一部分就变成了

```
import math
from array import array
from ROOT import TCanvas , TGraphPolar
...
ipt=range(0,npoints)
r=array('d',map(lambda x: x*(rmax-rmin)/(npoints-1.)+rmin,ipt))
theta=array('d',map(math.sin,r))
e=array('d',npoints*[0.])
...
```

8.1.1.1 自定义分箱

此示例结合了 Python 中对数组的舒适处理，以定义 ROOT 直方图的变量 bin 大小。 我们需要知道的是相关 ROOT 类及其方法的接口（来自 ROOT 文档）：

```
TH1F(const char* name , const char* title , Int_t nbinsx , const
Double_t* xbins)
```

这是 Python 代码：

```
import ROOT
from array import array
arrBins = array('d' ,(1 ,4 ,9 ,16) ) # array of bin edges
histo = ROOT.TH1F("hist", "hist", len(arrBins)-1, arrBins)
# fill it with equally spaced numbers
for i in range (1 ,16) :
```

```
   histo.Fill(i)
histo.Draw ()
```

8.2 自定义代码：从 C ++到 Python

ROOT 解释器和类型系统在 JITting C ++代码方面提供了有趣的可能性。 以此头文件为例，包含类和函数。

```
// file cpp2pythonExample.h
#include "stdio.h"

class A{
public:
 A(int i):m_i(i){}
 int getI() const {return m_i;}
private:
 int m_i=0;
};

void printA(const A& a ){
  printf ("The value of A instance is %i.\n",a.getI());
}
```

```
>>> import ROOT
>>> ROOT.gInterpreter.ProcessLine('#include "cpp2pythonExample.h"')
>>> a = ROOT.A(123)
>>> ROOT.printA(a)
The value of A instance is 123.
```

这个例子看似微不足道，但它显示了一个强大的 ROOT 功能。C ++代码可以在 PyROOT 中进行 JIT，并且在 C ++中定义的实体可以在 Python 中透明地使用！

9 结束语

这是我们通过 ROOT 为初学者导览的结束。还有很多想法可以说，但是现在你已经足够经验使用 ROOT 文档，最重要的是 ROOT 主页和 ROOT 参考指南以及所有 ROOT 类的文档或 ROOT 用户指南。

继续探索 ROOT 的一种非常有用的方法是研究任何 ROOT 安装的子目录教程/中的示例。

ROOT 的一些强大功能在本文档中未被处理，例如名为 RooFit 和 RooStats 的软件包为模型构建，拟合和统计分析提供了高级框架。ROOT 命名空间 TMVA 提供多变量分析工具，包括人工神经网络和许多其他用于分类问题的高级工具。本指南中已经提到了 ROOT 处理大量数据的卓越能力，通过 TTree 类实现。但是你还有更多东西需要探索！

本指南结束.....但希望不要与 ROOT 交互的结束！

References

[1] Matsumoto, Makoto. 1997. “Mersenne Twister Home Page.” http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

[2] Silin, I.N. 1983. “FUMILI.” *CERN Program Library* d510.

[3] *The Root Reference Guide*. 2013. http://root.cern.ch/drupal/content/reference-guide.

[4] *The Root Users Guide*. 2015. http://root.cern.ch/drupal/content/users-guide.

[5] “What is Cling.” 2015. https://root.cern.ch/drupal/content/cling.