ROOT

February 26, 2017

root : shows the ROOT splash screen and calls root.exe

root.exe : the executable that root calls, if you use a debugger such as gdb, you will need to run root.exe directly

rootcint : is the utility ROOT uses to create a class dictionary for CINT

rmkdepend : a modified version of makedepend that is used by the ROOT build system

root-config : a script returning the needed compile flags and libraries for projects that compile and link with ROOT

The behavior of a ROOT session can be configured with the options in the rootrc file. At start-up, ROOT looks for a .rootrc file in the following order :

./.rootrc //local directory

$HOME/.rootrc //user directory

$ROOTSYS/etc/system.rootrc //global ROOT directory

to see current settings, you can do:

root[] gEnv->Print()

Default Logon and Logoff Scripts:

in local directory:

rootalias.C (only loaded when root starts up)

rootlogon.C (executed when root starts up)

rootlogoff.C (executed when root ends)

in $HOME directory: .rootlogon.C

To start ROOT, you can type root at the system prompt. This starts up CINT, the ROOT command line C/C++ interpreter, and it gives you the ROOT prompt (root[0]).

- To quit ROOT, type .q at the ROOT prompt
- It is possible to launch ROOT with some command line options, as shown below:

> root -h

Usage: root [-l] [-b] [-n] [-q] [file1.C ... fileN.C]

Options:

-b : run in batch mode without graphics

-n : do not execute logon and logoff macros as specified in .rootrc

-q : exit after processing command line script files

-l : do not show the image logo (splash screen)

a powerful C/C++ interpreter giving you access to all available ROOT classes, global variables, and functions via the command line.

By typing C++ statements at the prompt, you can create objects, call functions, execute scripts, etc.

Use up and down arrows to recall commands: $HOME/.root_hist

Use emacs commands to navigate

Root> .x demos.C 出一个菜单，选择运行示例

Root> .x benchmark.C 运行后以此运行示例，显示画出的图，并给出测试结果

root <filename.C> to check the separate macro outputs

Un-named Scripts: script1.C

{

#include<iostream.h>

cout << “ Hello” << endl;

float x = 3.; float y = 5.; int i = 101;

cout << “$x =$ ” << x << “$y =$ ” << y << “$i =$ ” << i << endl;

}

root[] .x script1.C <enter>

Named Scripts: script2.C

#include <iostream.h>

int run(int j=10) {

cout << “ Hello” << endl;

float x = 3.; float y = 5.; int i= j;

cout <<“ x = ”<< x <<“ y = ”<< y <<“ i = ”<< i << endl;

return 0;

}

Macro : a file that is interpreted by CINT

ACLic : Automatic Compiler of Libraries for CINT

load code as a shared lib, root[0] .x mymacro.C+(43)

write a C++ code file, where ROOT class can be directly used

execute the C++ code file without compilation (take it as Macros)

Use ACLiC to build a shared library

Based on ROOT libraries to produce your own libraries or executables

.L macro.C　　Load 文件 macro.C

.x macro.C　　执行文件 macro.C

.ls　　显示 ROOT 当前环境的所有信息

.! ls　　显示 Linux 系统当前目录的所有信息

注：ROOT 环境中，ROOT 指令都以“.”开头，系统指令都以“.!”开头

可以直接在 ROOT 环境中运行 macro 文件 (自动调用 cint 编译器)，也可以在 makefile 中设置好相关参数用 g++ 编译得到可执行文件运行。

Macro 文件

用花括号括起来，后缀名一般用 “.C”

定义数学函数

利用 c++ 数学表达式

TF1* f1 = new TF1(“f1”,“sin(x)/x”,0,10);

利用 TMath 定义的函数

TF1 *f1 = new TF1(“f1”,“TMath::DiLog(x)”,0,10);

利用自定义 c++ 数学函数

Double_t myFun(x) {

return x+sqrt(x);

}

TF1* f1 = new TF1(“f1”,“myFun”,0,10);

ROOT 中数学函数的使用

在命令提示行下 root-lex32.C

在 ROOT 环境下.x ex32.C

1) 脚本中 void 函数的名字必须与文件名相同

2)ROOT 环境中定义类指针之后，如 TF1 *f1，之后输入 “f1→”，然后按一下 Tab 键，可以自动列出该类对象的成员函数和成员变量

一个 root 文件就像 UNIX 中的一个目录，它可以包含目录和没有层次限制的目标模块。即，在可以在 root 文件创建不同的目录子目录，目录中存放不同的类对象或普通数据。

TTree：减小磁盘空间和增加读取速度方面被优化

TNtuple：只能存储浮点数的 TTree；尽量避免使用

TTree 减少了每个目标模块的 header，但仍保留类的名字，而每个同类的目标模块名字可以被压缩。TTree 采用了 branch 的体系，每个 branch 的读取可以与别的 branch 无关。可以把 tree 看成 root 文件中的子目录，branch 看成子目录中的文件或者子目录。TTree 适用于大量的类型相同的对象，可以存储包括类对象、数组等各种类型数据。一般情况下，tree 的 Branch，Leaf 信息就是一个事例的完整信息。有了 tree 之后，可以很方便对事例进行循环处理。TTree 存储方式占用空间少，读取速度快。

TTree 的定义：

构造函数：

TTree(const char* name, const char* title, Int_t splitlevel = 99);

Branch 成员函数：

virtual TBranch*Branch(const char* name, void* address, const char* leaflist, Int_t buf-size = 32000);

创建 TTree，并设置 Branch，比如：

Int_t RunID;

TTree *t1 = new TTree(“t1”, “test tree”);

TBranch *br = t1→Branch(“RunID”,&RunID,“RunID/I”);

Branch 可以是单独的变量，也可以是一串变量，也可以是定长或不定长数组，也可以是 C 结构体，或者类对象 (继承自 TObject，如 TH1F 对象)。

查看 Tree 的信息

>root -l tree1.root 打开 root 文件

root[1].ls 查看文件信息，

发现 TTree t1

root[2]t1->Show(0); 显示第 0 个 event 的信息

root[3]t1->GetEntries() 总事例数

root[4]t1->Scan();

root[5]t1->Print();

root[6]t1->Draw(“px”);

也可以

>root-l 进入 root

root[0]TFile *f1=new TFile(“tree1.root”);

root[1]t1->Draw(“sqrt(px*px+py*py)”);

root[2]TH1F *h1;

root[3]t1->Draw(“px>>h1”);

root[4]t1->Draw(“py”,“px> 0”,“sames”);

root[5]t1->Draw(“py”,“”,“sames”);

TChain: 分析多个 root 文件

TChain 对象是包含相同 tree 的 ROOT 文件的列表。

ROOT 文件中的子目录

//创建 root 文件，创建后默认目录就是在 myfile.root 目录中，

//实际上，root 文件被当作一个目录。

TFile *fname=new TFile(“myfile.root”,“recreate”);

//gDirectory 只想当前目录，即 myfile.root

//可以调用 mkdir 函数在 ROOT 文件中创建一个子目录，如 subdir

gDirectory->mkdir(“subdir”);

gDirectory->cd(“subdir”); //进入到 subdir 子目录

TTree *tree = new TTree(“tree”,“tree in subdir”); //创建 TTree

.....

tree->Write();

//将 tree 写到当前目录，即 subdir 中

如果 root 文件中的类 t3 是在子目录 subdir 下，则可以这样定义：

TChain* fChain= new TChain(“subdir/t3”);

或者将子目录和类的名字全部放在 Add() 函数中

TChain* fChain=new TChain();

fChain->Add(“rootfiles/*.root/subdir/t3”);

注意; 从这里可以看出,

1) ROOT 文件中的目录 subdir 与系统的子目录 rootfiles 同等地位;

2) ROOT 文件的文件名在这里也类似于目录

3) TTree，即 t3 在这里也类似于目录

由此可以得到，当 ROOT 文件中存在多个 TTree，比如 t3，t4 时，需要哪个就使用哪个，其它的与我们无关。

直方图的归一化

void TH1::Scale(Double_t c1, Option_t *option)

默认 c1=1，把直方图每个区间的值 (BinContent) 乘以 c1

假设 sum=h1->Integral()

h1->Scale(c1) 之后，

h1->Integral() = c1*sum

不加参数时，h1->Scale() 没有变化 (默认 c1=1)

“归一化” 后，不仅 BinContent 变化了，BinError 也变化了

归一化常用于比较两种分布，找出区别。所以，将 2 个直方图归一化到积分相同进行比较才直观。

如果需要正确处理统计误差，需要在对 ROOT 脚本中调用 TH1 的某个静态成员函数，即 TH1::SetDefaultSumw2();

void SetDefaultSumw2(Bool_t sumw2 = kTRUE) //static function. When this static function is called with sumw2=kTRUE, all new histograms will automatically activate the storage of the sum of squares of errors, ie TH1::Sumw2 is automatically called.

相加：常用于相同实验的数据叠加，增加统计量。

......

root[1]TH1::SetDefaultSumw2();

root[1]TH1F *h3=new TH1F(*h1); root[2]h3->Add(h1,h2,a,b); 结果:h3 的 BinContent 被 a*h1+b*h2 替换，一般 a=b=1

相减：常用于从实验测量的分布中扣除本底。

......

root[1]TH1F *h3=new TH1F(*h1);

root[2]h3->Sumw2();//也可在定义 h3 前 TH1::SetDefaultSumw2();

root[3]h3->Add(h1,h2,a,-b);

结果:h3 的 BinContent 被 a*h1+b*h2 替换，一般 a=-b=1

误差:

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2} = \sqrt{n_1 + n_2} \tag{1}$$

(假设 h1 和 h2 独立)

相除：常用于效率的计算。常用于效率的计算。

root[1]TH1F *h3=new TH1F(*h1);

root[2]h3->Sumw2();

root[3]h3->Divide(h1,h2,a,b);

root[4]h3->Divide(h1,h2,a,b);

误差:

$$\sigma = \frac{n_1}{n_2}\sqrt{\frac{1}{n_1} + \frac{1}{n_2}} \tag{2}$$

(假设 h1 和 h2 独立)

如果 h1 和 h2 不独立 root[4]h3->Divide(h1,h2,a,b,“B”);

$$\sigma = \sqrt{\frac{\frac{n_1}{n_2}\left(1 - \frac{n_1}{n_2}\right)}{n_2}} \tag{3}$$

二项分布误差

相乘：常用于对分布进行诸如效率等的修正。

root>TH1F *h3=new TH1F(*h1); root>h3->Sumw2(); root>h3->Multiply(h1,h2,a,b);

包括归一化和加减乘除在内,

如果希望使用直方图的误差，都需要调用

TH1::SetDefaultSumw2();

或者，对每个直方图 (如 hist) 调用或者，对每个直方图 (如 hist) 调用

hist->Sumw2();

拟合直方图

将鼠标放到直方图上，右键，出现直方图操作选项，选择 FitPanel，可以在 FitPanel 中选作选项，选择 FitPanel，可以在 FitPanel 中选择拟合的各个选项，比如用什么函数拟合，择拟合的各个选项，比如用什么函数拟合，拟合的区间，等等。

gStyle->SetOptFit();//设置拟合选项

拟合前往往需要给出合理的参数初值

fcn->SetParameters(500,mean,sigma);

拟合后取出拟合得到的参数

Double_t mypar[3];

fcn->GetParameters(&mypar[0]);

MakeClass 或 MakeSelector 自动产生分析文件和头文件

root [0] TFile f(”ex51.root”);

root [1] .ls

TFile** ex51.root

TFile* ex51.root

KEY: TTree t4;1 Reconst events

root [2] t4->MakeClass();

或: t4->MakeClass(“MyClass”);

类的定义以及 Branch 地址设定、分析框架都已经自动完成。

1 ROOT Basics

It has an interpreter for macros (Cling) that you can run from the command line or run like applications. But it is also an interactive shell that can evaluate arbitrary statements and expressions.

Launch the ROOT interactive shell with the command

> root

The prompt should appear shortly:

root [0]

1.1 Learn C++ at the ROOT prompt

Behind the ROOT prompt there is an interpreter based on a real compiler toolkit: LLVM. It is therefore possible to exercise many features of C++ and the standard library.

1.2 ROOT as function plotter

Using one of ROOT's classes, TF1, allows us to display a function of one variable, x. All ROOT classes' names start with the letter T. A notable exception is RooFit. In this context all classes'names are of the form Roo*. f1 is an instance of a TF1 class, the arguments are used in the constructor; the first one of type string is a name to be entered in the internal ROOT memory management system, the second string type parameter defines the function, here $\sin(x)/x$, and the two parameters of type double define the range of the variable x. The Draw() method, here without any parameters, displays the function in a window which should pop up after you typed the above two lines.

Formulae in ROOT are evaluated using the class TFormula.

If having the file slits.C on disk, type root slits.C in the shell. This will start root and make it read the "macro"slits.C, i.e. all the lines in the file will be executed one after the other.

1.3 Controlling ROOT

As every command you type into ROOT is usually interpreted by Cling, an "escape character" is needed to pass commands to ROOT directly. This character is the dot at the beginning of a line.

root [1] .<command>

common commands:

quit root, type .q

obtain a list of commands, use .?

access the shell of the operating system, type .!<OS_command>; try, e.g. .!ls or .!pwd

execute a macro, enter .x <file_name>; in the above example, you might have used .x slits.C at the ROOT prompt

load a macro, type .L <file_name>; in the above example, you might instead have used the command .L slits.C followed by the function call slits();. Note that after loading a macro all functions and procedures defined therein are available at the ROOT prompt.

compile a macro, type .L <file_name> +; ROOT is able to manage for you the C++ compiler behind the scenes and to produce machine code starting from your macro. One could decide to compile a macro in order to obtain better performance or to get nearer to the production environment.

Use .help at the prompt to inspect the full list

1.4 Plotting Measurements

To display measurements in ROOT, including errors, there exists a powerful class TGraphErrors with different types of constructors.

root [0] TGraphErrors gr("ExampleData.txt");

root [1] gr.Draw("AP");

In data file, lines beginning with # are ignored.

The argument of the method Draw("AP") tells the TGraphPainter class to show the axes and to plot markers at the x and y positions of the specified data points.

1.5 Histograms in ROOT

Frequency distributions in ROOT are handled by a set of classes derived from the histogram class TH1, in our case TH1F. The letter F stands for "float", meaning that the data type float is used to store the entries in one histogram bin.

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();

In line 3 a histogram is instantiated, with a name, a title, a certain number of bins (100 of them, equidistant, equally sized) in the range from 0 to 5. Here use pseudo-random numbers generated with the method TF1::GetRandom to fill this histogram with data, which in turn uses an instance of the ROOT class TRandom created when ROOT is started. Data is entered in the histogram at line 4 using the method TH1F::Fill in a loop construct. As a result, the histogram is filled with 1000 random numbers distributed according to the defined function. The histogram is displayed using the method TH1F::Draw(). The class TH1F does not contain a convenient input format from plain text files. One number per line stored in the text file “expo.dat”is read in via an input stream and filled in the histogram until end of file is reached.

```
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();
```

1.6 Interactive ROOT

The mouse pointer changes its shape when touching objects on the plot. When the mouse is over an object, a right-click opens a pull-down menu displaying in the top line the name of the ROOT class you are dealing with, e.g. TCanvas for the display window itself, TFrame for the frame of the plot, TAxis for the axes, TPaveText for the plot name. Depending on which plot you are investigating, menus for the ROOT classes TF1, TGraphErrors or TH1F will show up when a right-click is performed on the respective graphical representations. The menu items allow direct access to the members of the various classes, and you can even modify them.

1.7 ROOT Beginners'FAQ

Special data types replacing the normal ones, e.g. Double_t, Float_t or Int_t replacing the standard double, float or int types.

The behavior of a ROOT session can be tailored with the options in the .rootrc file. At start-up, ROOT looks for a .rootrc file in the following order

./.rootrc //local directory

$HOME/.rootrc //user directory

$ROOTSYS/etc/system.rootrc //global ROOT directory

The parsing and interpretation of this file is handled by the ROOT class TEnv. The file .rootrc defines the location of two rather important files inspected at start-up: rootalias.C and rootlogon.C. They can contain code that needs to be loaded and executed at ROOT startup. rootalias.C is only loaded and best used to define some often used functions. rootlogon.C contains code that will be executed at startup: this file is extremely useful for example to pre-load a custom style for the plots created with ROOT. This is done most easily by creating a new TStyle object with your preferred settings, and then use the command gROOT→SetStyle("MyStyleName"); to make this new style definition the default one. rootlogoff.C is called when the session is finished.

Every command typed at the ROOT prompt is stored in a file .root_hist in your home directory. ROOT uses this file to allow for navigation in the command history with the up-arrow and down-arrow keys.

1.7.1 ROOT Global Pointers

All global pointers in ROOT begin with a small “g”.

gROOT: the gROOT variable is the entry point to the ROOT system. Technically it is an instance of the TROOT class. Using the gROOT pointer one has access to basically every object created in a ROOT based program. The TROOT object is essentially a container of several lists pointing to the main ROOT objects.

gStyle: By default ROOT creates a default style that can be accessed via the gStyle pointer. This class includes functions to set some of the following object attributes.

– Canvas

– Pad

– Histogram axis

– Lines

– Fill areas

– Text

– Markers

– Functions

– Histogram Statistics and Titles

– etc . . .

gSystem: An instance of a base class defining a generic interface to the underlying Operating System, in our case TUnixSystem.

gInterpreter: The entry point for the ROOT interpreter. Technically an abstraction level over a singleton instance of TCling.

2 ROOT Macros

macros executed by the ROOT C++ interpreter Cling

If you have a number of lines which you were able to execute at the ROOT prompt, they can be turned into a ROOT macro by giving them a name which corresponds to the file name without extension. The general structure for a macro stored in file MacroName.C is

void MacroName()

$< \cdots$

your lines of C++ code

$\cdots >$

The macro is executed by typing

> root MacroName.C

at the system prompt, or executed using .x

> root

root [0] .x MacroName.C

at the ROOT prompt. Or it can be loaded into a ROOT session and then be executed by typing

root [0].L MacroName.C

root [1] MacroName();

at the ROOT prompt.

2.1 A more complete example

root macro1.C

TF1 defines a function, TCanvas defines size and properties of the window used for our plot, and TLegend add a nice legend.

Line 13 : the name of the principal function (it plays the role of the "main" function in compiled programs) in the macro file. It has to be the same as the file name without extension.

Line $24 - 25$: instance of the TGraphErrors class. The constructor takes the number of points and the pointers to the arrays of x values, y values, x errors (in this case none, represented by the NULL pointer) and y errors. The second line defines in one shot the title of the graph and the titles of the two axes, separated by a ";".

Line 28-30 : These three lines are rather intuitive right ? To understand better the enumerators for colors and styles see the reference for the TColor and TMarker classes.

Line 33 : the canvas object that will host the drawn objects. The "memory leak" is intentional, to make the object existing also out of the macro1 scope.

Line 36 : the method DrawClone draws a clone of the object on the canvas. It has to be a clone, to survive after the scope of macro1, and be displayed on screen after the end of the macro execution. The string option "APE" stands for:

– A imposes the drawing of the Axes.

– P imposes the drawing of the graph's markers.

– E imposes the drawing of the graph's error bars.

Line 39 : define a mathematical function. There are several ways to accomplish this, but in this case the constructor accepts the name of the function, the formula, and the function range.

Line 41 : maquillage. Try to give a look to the line styles at your disposal visiting the documentation of the TLine class.

Line 43 : fits the f function to the graph, observe that the pointer is passed. It is more interesting to look at the output on the screen to see the parameters values and other crucial information that we will learn to read at the end of this guide.

Line 44 : again draws the clone of the object on the canvas. The "Same" option avoids the cancellation of the already drawn objects, in our case, the graph. The function f will be drawn using the same axis system defined by the previously drawn graph.

Line 47-52 : completes the plot with a legend, represented by a TLegend instance. The constructor takes as parameters the lower left and upper right corners coordinates with respect to the total size of the canvas, assumed to be 1, and the legend header string. You can add to the legend the objects, previously drawn or not drawn, through the addEntry method. Observe how the legend is drawn at the end: looks familiar now, right ?

Line 55-57 : defines an arrow with a triangle on the right hand side, a thickness of 2 and draws it.

Line 60-61 : interpret a Latex string which hast its lower left corner located in the specified coordinate. The #splitline{}{} construct allows to store multiple lines in the same TLatex object.

Line 63 : save the canvas as image. The format is automatically inferred from the file extension (it could have been eps, gif, . . .).

2.2 Summary of Visual e!ects

The complete summary of colours is represented by the ROOT "colour wheel". Refer to the online documentation of TColor. ROOT provides several graphics markers types.

The class representing arrows is TArrow, which inherits from TLine. The constructors of lines and arrows always contain the coordinates of the endpoints. Arrows also foresee parameters to specify their shapes.

Adding text in your plot is provided by the TLatex class. The objects of this class are constructed with the coordinates of the bottom-left corner of the text and a string which contains the text itself. The real twist is that ordinary Latex mathematical symbols are automatically interpreted, you just need to replace the "\" by a "#".

If "\" is used as control character , then the TMathText interface is invoked. It provides the plain TeX syntax and allow to access character's set like Russian and Japenese.

2.3 Interpretation and Compilation

2.3.1 Compile a Macro with ACLiC

ACLiC will create a compiled dynamic library for your macro, without any effort from your side, except the insertion of the appropriate header files. To generate an object library from the macro code, from inside the interpreter type (please note the "+"):

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

Once this operation is accomplished, the macro symbols will be available in memory and you will be able to execute it simply by calling from inside the interpreter:

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

2.3.2 Compile a Macro with the Compiler

Include the appropriate headers in the code and then exploit the root-config tool for the automatic settings of all the compiler flags. root-config is a script that comes with ROOT; it prints all flags and libraries needed to compile code and link it with the ROOT libraries. In order to make the code executable stand-alone, an entry point for the operating system is needed, in C++ this is the procedure int main();. The easiest way to turn a ROOT macro code into a stand-alone application is to add the following "dressing code"at the end of the macro file. This defines the procedure main, the only purpose of which is to call your macro:

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

To create a stand-alone program from a macro called ExampleMacro.C, type

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

and execute it by typing

```
> ./ExampleMacro
```

This procedure will, however, not give access to the ROOT graphics, as neither control of mouse or keyboard events nor access to the graphics windows of ROOT is available. If you want your stand-alone application have display graphics output and respond to mouse and keyboard, a slightly more complex piece of code can be used. A macro ExampleMacro_GUI is executed by the ROOT class TApplication.

3 Graphs

4 Histograms

ROOT supports histograms up to three dimensions. The histogram classes are split into further categories, depending on the set of possible bin values:

TH1C, TH2C and TH3C contain one byte per bin (maximum bin content = 255)

TH1S, TH2S and TH3S contain one short per bin (maximum bin content = 65 535)

TH1I, TH2I and TH3I contain one integer per bin (maximum bin content = 2 147 483 647)

TH1F, TH2F and TH3F contain one float per bin (maximum precision = 7 digits)

TH1D, TH2D and TH3D contain one double per bin (maximum precision = 14 digits)

ROOT also supports profile histograms. The inter-relation of two measured quantities X and Y can always be visualized with a two-dimensional histogram or scatter-plot. Profile histograms are used to display the mean value of Y and its RMS for each bin in X. If Y is an unknown but single-valued approximate function of X, it will have greater precision in a profile histogram than in a scatter plot.

TProfile : one dimensional profiles

TProfile2D : two dimensional profiles

All ROOT histogram classes are derived from the base class TH1.

4.1 Creating Histograms

Calling the Clone() method of an existing histogram

Making a projection from a 2-D or 3-D histogram

Reading a histogram from a file

// using various constructors

TH1* h1 = new TH1I(“h1”, “h1 title”, 100, 0.0, 4.0);

TH2* h2 = new TH2F(“h2”, “h2 title”, 40, 0.0, 2.0, 30, -1.5, 3.5);

TH3* h3 = new TH3D(“h3”, “h3 title”, 80, 0.0, 1.0, 100, -2.0, 2.0, 50, 0.0, 3.0);

// cloning a histogram

TH1* hc = (TH1*)h1->Clone();

// projecting histograms

// the projections always contain double values !

TH1* hx = h2->ProjectionX(); // ! TH1D, not TH1F

TH1* hy = h2->ProjectionY(); // ! TH1D, not TH1F

4.1.1 Constant or Variable Bin Width

To construct a histogram, provide the name and title of histogram and for each dimension: the number of bins, the minimum x (lower edge of the first bin) and the maximum x (upper edge of the last bin).

TH2* h = new TH2D(

/* name */ “h2”,

/* title */ “Hist with constant bin width”,

/* X-dimension */ 100, 0.0, 4.0,

/* Y-dimension */ 200, -3.0, 1.5);

When employing this constructor, you will create a histogram with constant (fixed) bin width on each axis.

To create histograms with variable bin widths, pass an array (single or double precision) of bin edges. When the histogram has n bins, then there are $n + 1$ distinct edges, so the array you pass must be of size $n + 1$.

const Int_t NBINS = 5;

Double_t edges[NBINS + 1] = $\{0.0, 0.2, 0.3, 0.6, 0.8, 1.0\}$;

// Bin 1 corresponds to range [0.0, 0.2]

// Bin 2 corresponds to range [0.2, 0.3] etc...

TH1* h = new TH1D(

/* name */ “h1”, /* title */ “Hist with variable bin width”,

/* number of bins */ NBINS,

/* edge array */ edges

);

Each histogram object contains three TAxis objects: fXaxis , fYaxis, and fZaxis. The bin edges are always stored internally in double precision.

Examine the actual edges / limits of the histogram bins by accessing the axis parameters.

const Int_t XBINS = 5; const Int_t YBINS = 5;

Double_t xEdges[XBINS + 1] = $\{0.0, 0.2, 0.3, 0.6, 0.8, 1.0\}$;

Double_t yEdges[YBINS + 1] = $\{-1.0, -0.4, -0.2, 0.5, 0.7, 1.0\}$;

TH2* h = new TH2D(”h2”, ”h2”, XBINS, xEdges, YBINS, yEdges);

TAxis* xAxis = h->GetXaxis(); TAxis* yAxis = h->GetYaxis(); cout << “Third bin on

Y-dimension: ” << endl; // corresponds to [-0.2, 0.5]

cout << “\tLower edge: ” << yAxis->GetBinLowEdge(3) << endl;

cout << “\tCenter: ” << yAxis->GetBinCenter(3) << endl;

cout << “\tUpper edge: ” << yAxis->GetBinUpEdge(3) << endl;

4.2 Bin Numbering

All histogram types support fixed or variable bin sizes. 2-D histograms may have fixed size bins along X and variable size bins along Y or vice-versa.

For all histogram types: nbins , xlow , xup

Bin#0 contains the underflow. Bin#1 contains the first bin with low-edge (xlow INCLUDED). The second to last bin (bin# nbins) contains the upper-edge (xup EXCLUDED). The Last bin (bin# nbins+1) contains the overflow. In case of 2-D or 3-D histograms, a “global bin”number is defined. For example, assuming a 3-D histogram h with binx, biny, binz, the function returns a global/linear bin number. This global bin is useful to access the bin information independently of the dimension.

A histogram can be re-binned via the TH1::Rebin() method. It returns a new histogram with the re-binned contents. If bin errors were stored, they are recomputed during the re-binning.

4.3 Filling Histograms

h1->Fill(x);

h1->Fill(x,w); // with weight

h2->Fill(x,y);

h2->Fill(x,y,w);

h3->Fill(x,y,z);

h3->Fill(x,y,z,w);

The Fill method computes the bin number corresponding to the given x, y or z argument and increments this bin by the given weight. The Fill() method returns the bin number for 1-D histograms or global bin number for 2-D and 3-D histograms. If TH1::Sumw2() has been called before filling, the sum of squares is also stored. One can increment a bin number directly by calling TH1::AddBinContent(), replace the existing content via TH1::SetBinContent(), and access the bin content of a given bin via TH1::GetBinContent().

Double_t binContent = h->GetBinContent(bin);

By default, the number of bins is computed using the range of the axis. You can change this to re-bin automatically by setting the automatic re-binning option:

h->SetBit(TH1::kCanRebin);

Once this is set, the Fill() method will automatically extend the axis range to accommodate the new value specified in the Fill() argument. The used method is to double the bin size until the new value fits in the range, merging bins two by two. The TTree::Draw() method extensively uses this automatic binning option when drawing histograms of variables in TTree with an unknown range. The automatic binning option is supported for 1-D, 2-D and 3-D histograms. During filling, some statistics parameters are incremented to compute the mean value and root mean square with the maximum precision. In case of histograms of type TH1C, TH1S, TH2C, TH2S, TH3C, TH3S a check is made that the bin contents do not exceed the maximum positive capacity (127 or 65 535). Histograms of all types may have positive or/and negative bin contents.

4.4 Random Numbers and Histograms

TH1::FillRandom() can be used to randomly fill a histogram using the contents of an existing TF1 function or another TH1 histogram (for all dimensions).

root[] TH1F h1("h1", "Histo from a Gaussian", 100, -3, 3);

root[] h1.FillRandom("gaus", 10000);

TH1::GetRandom() can be used to get a random number distributed according the contents of a histogram. To fill a histogram following the distribution in an existing histogram you can use the second signature of TH1::FillRandom(). Next code snipped assumes that h1 is an existing histogram (TH1).

root[] TH1F h2("h2", "Histo from existing histo", 100, -3, 3);

root[] h2.FillRandom(&h1, 1000);

The distribution contained in the histogram h1 (TH1) is integrated over the channel contents. It is normalized to one. The second parameter (1000) indicates how many random numbers are generated.

Getting 1 random number implies:

Generating a random number between 0 and 1 (say r1)

Find the bin in the normalized integral for r1

Fill histogram channel

4.5 Adding, Dividing, and Multiplying

4.6 Projections

4.7 Drawing Histograms

4.8 Making a Copy of an Histogram

4.9 Normalizing Histograms

4.10 Saving/Reading Histograms to/from a File

4.11 Miscellaneous Operations

4.11.1 χ^2 test

Double_t TH1::Chi2Test(const TH1* h2, Option_t* option = “UU”, Double_t* res = 0) const

χ^2 test for comparing weighted and unweighted histograms

Function: Returns p-value. Other return values are specified by the 3rd parameter

Parameters h2 the second histogram

option

“UU” = experiment experiment comparison (unweighted-unweighted)

“UW” = experiment MC comparison (unweighted-weighted). Note that the first histogram should be unweighted

“WW” = MC MC comparison (weighted-weighted)

“NORM” = to be used when one or both of the histograms is scaled but the histogram originally was unweighted by default underflows and overflows are not included:

“OF” = overflows included

“UF” = underflows included

“P” = print chi2, ndf, p_value, igood

“CHI2” = returns chi2 instead of p-value

“CHI2/NDF” = returns χ^2/ndf

res not empty - computes normalized residuals and returns them in this array

Comparison of two histograms expect hypotheses that two histograms represent identical distributions. To make a decision p-value should be calculated. The hypotheses of identity is rejected if the p-value is lower then some significance level. Traditionally significance levels 0.1, 0.05 and 0.01 are used. The comparison procedure should include an analysis of the residuals which is often helpful in identifying the bins of histograms responsible for a significant overall χ^2 value. Residuals are the difference between bin contents and expected bin contents. Most convenient for analysis are the normalized residuals. If hypotheses of identity are valid, normalized residuals are approximately independent and identically distributed random variables having $N(0,1)$ distribution. Analysis of residuals expect test of above mentioned properties of residuals. Notice that indirectly the analysis of residuals increase the power of χ^2 test.

Consider two histograms with the same binning and the number of bins equal to r. Let us denote the number of events in the i-th bin in the first histogram as n_i and as m_i in the second one. The total number of events in the first and second histogram are respectively equal to:

$$N = \sum_{i=1}^{r} n_i \ ,$$
$$M = \sum_{i=1}^{r} m_i \ .$$

The hypothesis of identity (homogeneity) is that the two histograms represent random values with identical distributions. It is equivalent that there exist r constants $p_1, \cdots, p_r$, such that

$$\sum_{i=1}^{r} p_i = 1 \ . \tag{4}$$

The probability of belonging to the i-th bin for some measured value in both experiments is equal to p_i. The number of events in the i-th bin is a random variable with a distribution approximated by a Poisson probability distribution

$$\frac{e^{-Np_i}(Np_i)^{n_i}}{n_i!} \tag{5}$$

for the first histogram and with distribution

$$\frac{e^{-Mp_i}(Mp_i)^{m_i}}{m_i!} \tag{6}$$

for the second histogram. If the hypothesis of homogeneity is valid, then the maximum likelihood estimator of p_i, $i = 1, \cdots, r$, is

$$\hat{p}_i = \frac{n_i + m_i}{N + M} \tag{7}$$

and then

$$\chi^2 = \sum_{i=1}^{r} \frac{(n_i - N\hat{p}_i)^2}{N\hat{p}_i} + \sum_{i=1}^{r} \frac{(m_i - M\hat{p}_i)^2}{M\hat{p}_i} = \frac{1}{MN} \sum_{i=1}^{r} \frac{(Mn_i - Nm_i)^2}{n_i + m_i} \tag{8}$$

has approximately a $\chi^2_{(r-1)}$ distribution. The comparison procedure can include an analysis of the residuals which is often helpful in identifying the bins of histograms responsible for a significant overall χ^2 value. Most convenient for analysis are the adjusted (normalized) residuals

$$r_i = \frac{n_i - N\hat{p}_i}{\sqrt{N\hat{p}_i}\sqrt{(1 - N/(N+M))(1 - (n_i + m_i)/(N+M))}} \tag{9}$$

If hypotheses of homogeneity are valid then residuals r_i are approximately independent and identically distributed random variables having $N(0,1)$ distribution. The application of the χ^2 test has restrictions related to the value of the expected frequencies Np_i, Mp_i, $i = 1, \cdots, r$. A conservative rule formulated is that all the expectations must be 1 or greater for both histograms. In practical cases when expected frequencies are not known the estimated expected frequencies $M\hat{p}_i$, $N\hat{p}_i$, $i = 1, \cdots, r$ can be used.

4.11.2 Kolmogorov–Smirnov test

Double_t TMath::KolmogorovTest(Int_t na, const Double_t* a, Int_t nb, const Double_t* b, Option_t* option)

Statistical test whether two one-dimensional sets of points are compatible with coming from the same parent distribution, using the Kolmogorov test. That is, it is used to compare two experimental distributions of unbinned data.

Input: a,b: One-dimensional arrays of length n_a, n_b, respectively. The elements of a and b must be given in ascending order. option is a character string to specify options "D" Put out a line of "Debug" printout "M" Return the Maximum Kolmogorov distance instead of prob

Output: The returned value prob is a calculated confidence level which gives a statistical test for compatibility of a and b. Values of prob close to zero are taken as indicating a small probability of compatibility. For two point sets drawn randomly from the same parent distribution, the value of prob should be uniformly distributed between zero and one. in case of error the function return -1 If the 2 sets have a different number of points, the minimum of the two sets is used.

Method: The Kolmogorov test is used. The test statistic is the maximum deviation between the two integrated distribution functions, multiplied by the normalizing factor rdmax $\times \sqrt{\frac{n_a \times n_b}{(n_a + n_b)}}$.

The nuts-and-bolts of the TMath::KolmogorovTest() algorithm is a for-loop over the two sorted arrays a and b representing empirical distribution functions. The for-loop handles 3 cases: when the next points to be evaluated satisfy $a > b$, $a < b$, or $a = b$:

for(Int_t i=0;i<na+nb;i++) { if (a[ia-1] < b[ib-1]) { rdiff -= sa; ia++; if (ia > na) {ok = kTRUE; break;} }else if (a[ia-1] > b[ib-1]) { rdiff += sb; ib++; if (ib > nb) {ok = kTRUE; break;} } else{ rdiff += sb - sa; ia++; ib++; if (ia > na){ok = kTRUE; break;} if (ib > nb) {ok = kTRUE; break;} } rdmax = TMath::Max(rdmax,TMath::Abs(rdiff)); }

For the last case, $a = b$, the algorithm advances each array by one index in an attempt to move through the equality. However, this is incorrect when one or the other of a or b (or both) have a repeated value, call it x. For the KS statistic to be computed properly, rdiff needs to be calculated after all of the a and b at x have been tallied (this is due to the definition of the empirical distribution function; another way to convince yourself that the old CERNLIB method is wrong is that it implies that the function defined as the difference between a and b is multi-valued at x – besides being ugly, this would invalidate Kolmogorov's theorem).

4.12 Alphanumeric Bin Labels

4.13 Histogram Stacks

5 Functions and Parameter Estimation

6 File I/O and Parallel Analysis

```
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();
}
```

RECREATE option forces ROOT to create a new file even if a file with the same name exists on disk.

Use the Cling command line to access information in the file and draw the previously written histogram

```
> 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()
```

Or use a macro to carry out the job:

```
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

h→Draw();

}
```

6.1 N-tuples in ROOT

6.1.1 Storing simple N-tuples

TNtuple class is a simplified version of the TTree class. A ROOT TNtuple object can store rows of float entries.

Open the ROOT file (cond_data.root) written by the macro above in an interactive session and use a TBrowser to interactively inspect it

root[0] TBrowser b

You find the columns of your n-tuple written as leafs. Simply clicking on them you can obtain histograms of the variables!

Try the following commands at the shell prompt and in the interactive ROOT shell

> root conductivity_experiment.root

Attaching file conductivity_experiment.root as _file0· · ·

root [0] cond_data→Draw("Current:Potential")

extend the syntax typing for example

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

root [2] cond_data→Draw("Current/Potential:Temperature")

Here show how to navigate in such a multi-dimensional space of variables and unveil relations between variables using n-tuples.

6.1.2 Reading N-tuples

accessing the content of a n-tuple: after loading the n-tuple, its branches are assigned to variables and GetEntry(long) automatically fills them with the content for a specific row.

6.1.3 Storing Arbitrary N-tuples

Write n-tuples of arbitrary type by using ROOT's TBranch class. TNtuple::Fill() accepts only floats

The following macro creates the same n-tuple as before but the branches are booked directly. The Fill() function then fills the current values of the connected variables to the tree

6.1.4 Processing N-tuples Spanning over Several Files

6.1.5 For the advanced user: Processing trees with a selector script

6.1.6 For power-users: Multi-core processing with PROOF lite

6.1.7 Optimisation Regarding N-tuples

6.2 The Physical Layout of ROOT Files

A ROOT file is like a UNIX file directory. It can contain directories and objects organized in unlimited number of levels. It also is stored in machine independent format.

TFile constructor describes the ROOT file which has the extension ".root". To view ROOT file, we need to open it again, and to create a TBrowser object by

```
root[] TFile f("demo.root")
root[] TBrowser browser;
```

You can check if the file is correctly opened by:

```
TFile f("demo.root");
if(f.IsZombie()){
cout << "Error opening file" << endl;
exit(-1);
}else{
...
}
```

Once we have the TFile object, we can call the TFile::Map() method to view the physical layout. The output prints the date/time, the start record address, the number of bytes in the record, the class name of the record and the compression factor.

Here we see the fifteen histograms (TH1F's) with the first one starting at byte 148. We also see an entry TFile. You may notice that the first entry starts at byte 100. The first 100 bytes are taken by the file header.

6.2.1 The File Header

The table shows the file header information. When fVersion is greater than 1000000, the file is a large file (> 2 GB) and the offsets will be 8 bytes long. The location in brackets are the location in the case of a large file.

The first four bytes of the file header contain the string "root"which identifies a file as a ROOT file. Because of this identifier, ROOT is not dependent on the ".root"extension. The nfree and value is the number of free records. This variable along with FNBytesFree keeps track of the free space in terms of records and bytes. This count also includes the deleted records, which are available again.

6.2.2 The Top Directory Description

The 84 bytes after the file header contain the top directory description, including the name, the date and time it was created, and the date and time of the last modification.

6.2.3 The Histogram Records

The first 4 bytes of each record is an integer holding the number of bytes in this record. A negative number flags the record as deleted, and makes the space available for recycling in the next writing. The rest of bytes in the header contain all the information to identify uniquely a data block on the file. It is followed by the object data.

If the key is located past the 32 bit file limit (> 2 GB) then some fields will be 8 bytes instead of 4 bytes (values between the brackets). There is a reference to TKey.

6.2.4 The Class Description List (StreamerInfo List)

The histogram records are followed by the StreamerInfo list of class descriptions. The list contains the description of each class that has been written to file.

The class description is recursive, because to fully describe a class, its ancestors and object data members have to be described also. The class description list contains the description for:

TH1F

all classes in the TH1F inheritance tree

all classes of the object data members

all classes in the object data members' inheritance tree

This description is implemented by the TStreamerInfo class, and is often referred to as simply StreamerInfo. You can print a file's StreamerInfo list with the TFile::ShowStreamerInfo method.

ROOT allows a class to have multiple versions, and each version has its own description in form of a StreamerInfo.

6.2.5 The List of Keys and the List of Free Blocks

The last three entries on the output of TFile::Map() are the list of keys, the list of free segments, and the address where the data ends. When a file is closed, it writes a linked list of keys at the end of the file.

6.2.6 File Recovery

A file may become corrupted or it may be impossible to write it to disk and close it properly. If a file that has been not properly closed is opened again, it is scanned and rebuilt according to the information in the record header. The recovery algorithm reads the file and creates the saved objects in memory according to the header information. It then rebuilds the directory and file structure. If the file is opened in write mode, the recovery makes the correction on disk when the file is closed; however if the file is opened in read mode, the correction can not be written to disk. You can also explicitly invoke the recovery procedure by calling the TFile::Recover() method. You can recover the directory structure, but you cannot save what you recovered to the file on disk. In the following example, we interrupted and aborted the previous ROOT session, causing the file not to be closed. When we start a new session and attempt to open the file, it gives us an explanation and status on the recovery attempt.

6.3 The Logical ROOT File: TFile and TKey

TFile::Map() method reads the file sequentially and prints information about each record while scanning the file. Beside supporting sequential access and hence ROOT provides random or direct access, i.e. reading a specified object at a time. To do so, TFile keeps a list of TKeys, which is essentially an index to the objects in the file. The TKey class describes the record headers of objects in the file. For example, we can get the list of keys and print them. To find a specific object on the file we can use the TFile::Get() method. The TFile::Get() finds the TKey object with name "h9". Using the TKey info, it will import in memory the object in the file at the file address #3352 (see the output from the

TFile::Map above). This is done by the Streamer method. Since the keys are available in a TList of TKeys, we can iterate over the list of keys.

In addition to the list of keys, TFile also keeps two other lists: TFile::fFree is a TList of free blocks used to recycle freed up space in the file. ROOT tries to find the best free block. If a free block matches the size of the new object to be stored, the object is written in the free block and this free block is deleted from the list. If not, the first free block bigger than the object is used. TFile::fListHead contains a sorted list (TSortedList) of objects in memory.

6.3.1 Viewing the Logical File Contents

TFile is a descendent of TDirectory. We can list the contents, print the name, and create subdirectories. In a ROOT session, you are always in a directory and the directory you are in is called the current directory and is stored in the global variable gDirectory.

create a ROOT file by executing a script

root[] .x $ROOTSYS/tutorials/hsimple.C

We open the file with the intent to update it, and list its contents.

root[] TFile f (“hsimple.root”,“UPDATE”)

root[] f.ls()

TFile** hsimple.root

TFile* hsimple.root

KEY: TH1F hpx;1 This is the px distribution

KEY: TH2F hpxpy;1 py vs px

KEY: TProfile hprof;1 Profile of pz versus px

KEY: TNtuple ntuple;1 Demo ntuple

It shows the two lines starting with TFile followed by four lines starting with the word "KEY". The four keys tell us that there are four objects on disk in this file. The syntax of the listing is:

KEY:<class><variable>;<cycle number><title>

For example, the first line in the list means there is an object in the file on disk, called hpx. It is of the class TH1F (one-dimensional histogram of floating numbers). The object' s title is "This is the px distribution". If the line starts with OBJ, the object is in memory. The <class> is the name of the ROOT class (T-something). The <variable> is the name of the object. The cycle number along with the variable name uniquely identifies the object. The <title> is the string given in the constructor of the object as title.

6.3.2 The Current Directory

When you create a TFile object, it becomes the current directory. Therefore, the last file to be opened is always the current directory. To check your current directory you can type:

root[] gDirectory→pwd()

Rint:/

This means that the current directory is the ROOT session (Rint). When you create a file, and repeat the command the file becomes the current directory.

root[] TFile f1("AFile1.root");

root[] gDirectory→pwd()

AFile1.root:/

If you create two files, the last becomes the current directory.

```
root[] TFile f2(“AFile2.root”);
root[] gDirectory→pwd()
AFile2.root:/
```

To switch back to the first file, or to switch to any file in general, you can use the TDirectory::cd method.

```
root[] f1.cd();
root[] gDirectory→pwd()
AFile1.root:/
```

changes the current directory back to the first file

You open the file in “READ”mode, it still becomes the current directory. Cling also offers a shortcut for gDirectory→pwd() and gDirectory→ls(), you can type:

```
root[] .pwd
AFile1.root:/
root[] .ls
TFile** AFile1.root
TFile* AFile1.root
```

To return to the home directory where we were before:

```
root[] gROOT→cd()
(unsigned char)1
root[] gROOT→pwd()
Rint:/
```

6.3.3 Objects in Memory and Objects on Disk

The TFile::ls() method has an option to list the objects on disk ("-d") or the objects in memory ("-m"). If no option is given it lists both, first the objects in memory, then the objects on disk.

root[] TFile *f = new TFile("hsimple.root");

root[] gDirectory→ls("-m")

TFile** hsimple.root

TFile* hsimple.root

This correctly states that no objects are in memory. gDirectory is the current directory and here is equivalent to "f", i.e.

.ls("-m")

f→ls("-m")

Next command lists the objects on disk in the current directory

root[] gDirectory→ls("-d")

TFile** hsimple.root

TFile* hsimple.root

KEY: TH1F hpx;1 This is the px distribution

KEY: TH2F hpxpy;1 py vs px

KEY: TProfile hprof;1 Profile of pz versus px

KEY: TNtuple ntuple;1 Demo ntuple

To bring an object from disk into memory, we have to use it or "Get"it explicitly. When we use the object, ROOT gets it for us. Any reference to hprof will read it from the file.

For example drawing hprof will read it from the file and create an object in memory.

```
root[] hprof→Draw()
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root[] f→ls()
TFile** hsimple.root
TFile* hsimple.root
OBJ: TProfile hprof Profile of pz versus px : 0
KEY: TH1F hpx;1 This is the px distribution
KEY: TH2F hpxpy;1 py vs px
KEY: TProfile hprof;1 Profile of pz versus px
KEY: TNtuple ntuple;1 Demo ntuple
```

There is a new line that starts with OBJ. An object of class TProfile, called hprof has been added in memory to this directory. This new hprof in memory is independent from the hprof on disk. If we make changes to the hprof in memory, they are not propagated to the hprof on disk. A new version of hprof will be saved once we call Write.

hprof is of the class TProfile that inherits from TH1D, which inherits from TH1. TH1 is the basic histogram.

```
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

It tells us that a TCanvas was created and it named it c1. The canvas is not added to the current directory, because by default ONLY histograms and trees are added to the object list of the current directory. It was added to the list of canvases. This list can be obtained by the command gROOT→GetListOfCanvases()→ls(). The ls() will print the contents of the list. In our list, we have one canvas called c1. It has a TFrame, a TProfile, and a TPaveStats. TEventList objects are also added to the current directory. This list can be obtained by the command gROOT→GetListOfCanvases()→ls(). The ls() will print the contents of the list. In our list, we have one canvas called c1. It has a TFrame, a TProfile, and a TPaveStats.

```
root[] gROOT→GetListOfCanvases()→ls()
root[] hpx→Draw()
root[] f→ls()
TFile** hsimple.root
TFile* hsimple.root
OBJ: TProfile hprof Profile of pz versus px : 0
OBJ: TH1F hpx This is the px distribution : 0
KEY: TH1F hpx;1 This is the px distribution
KEY: TH2F hpxpy;1 py vs px
KEY: TProfile hprof;1 Profile of pz versus px
```

KEY: TNtuple ntuple;1 Demo ntuple → TFile::ls() loops over the list of objects in memory and the list of objects on disk. In both cases, it calls the ls() method of each object. The implementation of the ls method is specific to the class of the object, all of these objects are descendants of TObject and inherit the TObject::ls() implementation. The histogram classes are descendants of TNamed that in turn is a descent of TObject. In this case, TNamed::ls() is executed, and it prints the name of the class, and the name and title of the object. Each directory keeps a list of its objects in the memory. You can get this list by TDirectory::GetList(). To see the lists in memory contents, root[]f→GetList()→ls()

or root[] gDirectory→GetList()→ls()

OBJ: TProfile hprof Profile of pz versus px : 0

OBJ: TH1F hpx This is the px distribution : 0

6.3.4 Saving Histograms to Disk

6.3.5 Histograms and the Current Directory

6.3.6 Saving Objects to Disk

6.3.7 Saving Collections to Disk

6.3.8 A TFile Object Going Out of Scope

6.3.9 Retrieving Objects from Disk

6.3.10 Subdirectories and Navigation

7 Trees

In case you want to store large quantities of same-class objects, ROOT has designed the TTree and TNtuple classes specifically for that purpose. The TTree class is optimized to reduce disk space and enhance access speed. A TNtuple is a TTree that is limited to only hold floating-point numbers; a TTree on the other hand can hold all kind of data, such as objects or arrays in addition to all the simple types.

When using a TTree, we fill its branch buffers with leaf data and the buffers are written to disk when it is full. Each object is not written individually, but rather collected and written a bunch at a time.

Since the unit to be compressed is a buffer, and the TTree contains many same-class objects, the header of the objects can be compressed. The TTree reduces the header of each object, but it still contains the class name. Using compression, the class name of each same-class object has a good chance of being compressed, since the compression algorithm recognizes the bit pattern representing the class name. Using a TTree and compression the header is reduced to about 4 bytes compared to the original 60 bytes. However, if compression is turned off, you will not see these large savings.

A tree uses a hierarchy of branches, and each branch can be read independently from any other branch.

7.1 A Simple TTree

Creates one branch with the TTree::Branch method. The first parameter of the Branch method is the branch name. The second parameter is the address from which the first leaf is to be read. In this example it is the address of the structure staff. Once the branch is defined, the script reads the data from the ASCII file into the staff_t structure and fills the tree. The ASCII file is closed, and the ROOT file is written to disk saving the tree.

7.2 Show an Entry with TTree::Show

Use the TTree::Show method to access one entry of a tree. For example to look at the 10th entry in the staff.root tree:

root[] TFile f("staff.root")

root[] T→Show(10)

7.3 Print the Tree Structure with TTree::Print

TTree::Print can be used to see the tree structure meaning the number of entries, the branches and the leaves.

7.4 Scan a Variable the Tree with TTree::Scan

TTree::Scan method shows all values of the list of leaves separated by a colon.

root[] T→Scan(”Cost:Age:Children”)

7.5 The Tree Viewer

The tree viewer is a quick and easy way to examine a tree. To start the tree viewer, open a file and object browser. Right click on a TTree and select StartViewer. You can also start the tree viewer from the command line. First load the viewer library.

root[] TFile f(“staff.root”)

root[] T→StartViewer()

If you want to start a tree viewer without a tree, you need to load the tree player library first:

root[] gSystem→Load(“libTreeViewer.so”)

root[] new TTreeViewer()

tree viewer

The left panel contains the list of trees and their branches. You can add more trees with the File-Open command to open the file containing the new tree, then use the context menu on the right panel, select SetTreeName and enter the name of the tree to add. On the right are the leaves or variables in the tree. You can double click on any leaf to a histogram it.

The toolbar in the upper part can be used for user commands, changing the drawing option and the histogram name. The lower part contains three picture buttons that draw a histogram, stop the current command, and refresh the tree. The three check buttons toggle the following:

Hist– the histogram drawing mode;

Scan– enables redirecting of TTree::Scan command in an ASCII file;

Rec– enables recording of the last issued command.

To draw more than one dimension you can drag and drop any leaf to the X,Y,Z boxes. Then push the Draw button, witch is marked with the purple icon on the bottom left. All commands can be interrupted at any time by pressing this button.

The method TTree::Refresh is called by pressing the refresh button in TTreeViewer. It redraws the current exposed expression. Calling TTree::Refresh is useful when a tree is produced by a writer process and concurrently analyzed by one or more readers.

To add a cut/weight to the histogram, enter an expression in the "cut box". The cut box is the one with the scissor icon.

Below them there are two text widgets for specifying the input and output event lists. A Tree Viewer session is made by the list of user-defined expressions and cuts, applying to a specified tree. A session can be saved using File/SaveSource menu or the SaveSource method from the context menu of the right panel. This will create a macro having as default name treeviewer.C that can be ran at any time to reproduce the session.

Besides the list of user-defined expressions, a session may contain a list of RECORDS. A record can be produced in the following way: dragging leaves/expression on X/Y/Z; changing drawing options; clicking the RED button on the bottom when happy with the histogram.

NOTE that just double clicking a leaf will not produce a record: the histogram must be produced when clicking the DRAW button on the bottom-left. The records will appear on the list of records in the bottom right of the tree viewer.

Selecting a record will draw the corresponding histogram. Records can be played using the arrow buttons near to the record button. When saving the session, the list of records is being saved as well.

Records have a default name corresponding to the Z: Y: X selection, but this can be changed using SetRecordName() method from the right panel context menu. You can create a new expression by right clicking on any of the E() boxes. The expression can be dragged and dropped into any of the boxes (X, Y, Z, Cut, or Scan). To scan one or more variables, drop them into the Scan box, then double click on the box. You can also redirect the result of the scan to a file by checking the Scan box on top.

When the "Rec" box is checked, the Draw and Scan commands are recorded in the history file and echoed on the command line. The "Histogram" text box contains the name of the resulting histogram. By default it is htemp. You can type any name, if the histogram does not exist it will create one. The Option text box contains the list of Draw options. See "Draw Options". You can select the options with the Options menu. The Command box lets you enter any command that you could also enter on the command line. The vertical slider on the far left side can be used to select the minimum and maximum of an event range. The actual start and end index are shown in on the bottom in the status window.

There is an extensive help utility accessible with the Help menu. The IList and OList are to specify an input list of entry indices and a name for the output list respectively. Both need to be of type TList and contain integers of entry indices. These lists are described below in the paragraph “Error! Reference source not found.”.

7.6 Creating and Saving Trees

To create a TTree we use its constructor. Then we design our data layout and add the branches. A tree can be created by giving a name and title: → TTree t(”MyTree”,”Example Tree”);

7.6.1 Creating a Tree from a Folder Hierarchy

An alternative way to create a tree and organize it is to use folders. You can build a folder structure and create a tree with branches for each of the sub-folders:

TTree folder_tree(“MyFolderTree”,“/MyFolder”)

The second argument “/MyFolder”is the top folder, and the “/” signals the TTree constructor that this is a folder not just the title. You fill the tree by placing the data into the folder structure and calling TTree::Fill.

7.6.2 Tree and TRef Objects

MyTree→BranchRef();

This call requests the construction of an optional branch supporting table of references (TRefTable). This branch (TBranchRef) will keep all the information needed to find the branches containing referenced objects at each Tree::Fill, the branch numbers containing the referenced objects are saved in the table of references. When the Tree header is saved (via TTree::Write for example), the branch is saved, keeping the information with the pointers to the branches having referenced objects. Enabling this optional table, allow TTree::Draw to automatically load the branches needed to dereference a TRef (or TRefArray) object

7.6.3 Autosave

Autosave gives the option to save all branch buffers every n byte. We recommend using Autosave for large acquisitions. If the acquisition fails to complete, you can recover the file and all the contents since the last Autosave. To set the number of bytes between Autosave you can use the TTree::SetAutosave() method. You can also call TTree::Autosave in the acquisition loop every nentry.

7.6.4 Trees with Circular Buffers

void TTree::SetCircular(Long64_t maxEntries);

where maxEntries is the maximum number of entries to be kept in the buffers. When the number of entries exceeds this value, the first entries in the Tree are deleted and the buffers used again.

7.6.5 Size of TTree in the File

When writing a TTree to a file, if the file size reaches the value stored in the TTree::GetMaxTreeSize(), the current file is closed and a new file is created. If the original file is named "myfile.root" , subsequent files are named "myfile_1.root", "myfile_2.root", etc.

Currently, the automatic change of file is restricted to the case where the tree is in the top level directory. The file should not contain sub-directories. Before switching to a new file, the tree header is written to the current file, then the current file is closed. To process the multiple files created by ChangeFile(), one must use a TChain.

The new file name has a suffix "_N"where N is equal to fFileNumber+1. By default a Root session starts with fFileNumber=0. One can set fFileNumber to a different value via TTree::SetFileNumber(). In case a file named "_N"already exists, the function will try a file named "___N", then "_____N", etc. The maximum tree size can be set via the static function TTree::SetMaxTreeSize(). The default value of fgMaxTreeSize is 1.9 GB. If the current file contains other objects (like TH1 and TTree), these objects are automatically moved to the new file.

7.6.6 User Info Attached to a TTree Object

The function TTree::GetUserInfo() allows adding any object defined by a user to the tree that is not depending on the entry number.

tree→GetUserInfo()→Add(myruninfo);

7.6.7 Indexing a Tree

Use TTree::BuildIndex() to build an index table using expressions depending on the value in the leaves.

tree→BuildIndex(majorname, minorname);

The index is built in the following way:

- a pass on all entries is made like in TTree::Draw()
- var1 = majorname

- var2 = minorname
- sel = $2^{31} \times$ majorname + minorname
- for each entry in the tree the sel expression is evaluated and the results array is sorted

into fIndexValues

Once the index is computed, using the TTree::GetEntryWithIndex(majornumber, minornumber), one entry can be retrieved.

// to create an index using leaves Run and Event

tree.BuildIndex(”Run”,”Event”);

// to read entry corresponding to Run=1234 and Event=56789

tree.GetEntryWithIndex(1234,56789);

Note that majorname and minorname may be expressions using original tree variables e.g.: “run-90000”, “event+3*xx””. In case an expression is specified, the equivalent expression must be computed when calling GetEntryWithIndex(). To build an index with only majorname, specify minorname=”0” (default).

Once the index is built, it can be saved with the TTree object with:

tree.Write(); //if the file has been open in “update” mode

The most convenient place to create the index is at the end of the filling process just before saving the tree header. If a previous index was computed, it is redefined by this new call.

This function can also be applied to a TChain. The return value is the number of entries in the Index (< 0 indicates failure).

7.7 Branches

The class for a branch is called TBranch. If two variables are independent, and the designer knows the variables will not be used together, they should be placed on separate branches. If, however, the variables are related, such as the coordinates of a point, it is most efficient to create one branch with both coordinates on it. A variable on a TBranch is called a leaf (TLeaf). Another point to keep in mind when designing trees is that branches of the same TTree can be written to separate files. To add a TBranch to a TTree we call the method TTree::Branch(). Note that we DO NOT use the TBranch constructor.

The TTree::Branch method has several signatures. The branch type differs by what is stored in it. A branch can hold an entire object, a list of simple variables, contents of a folder, contents of a TList, or an array of objects.

7.8 Adding a Branch to Hold a List of Variables

If in the staff.root, the data to save is a list of simple variables, such as integers or floats, we use the following TTree::Branch signature

tree→Branch("Ev_Branch",&event, "temp/F:ntrack/I:nseg:nvtex:flag/i");

The first parameter is the branch name. The second parameter is the address from which the first variable is to be read.

The leaf name is NOT used to pick the variable out of the structure, but is only used as the name for the leaf. The list of variables needs to be in a structure in the order described in the third parameter. This third parameter is a string describing the leaf list. Each leaf has a name and a type separated by a "/" and it is separated from the next leaf by a “:”.

<Variable> / <type>:<Variable> / <type>

The example has two leafs: a floating-point number called temp and an integer named ntrack.

“temp/F:ntrack/I:”

The type can be omitted and if no type is given, the same type as the previous variable is assumed. This leaf list has three integers called ntrack, nseg, and nvtex.

“ntrack/I:nseg:nvtex”

There is one more rule: when no type is given for the very first leaf, it becomes a float (F). This leaf list has three floats called temp, mass, and px.

“temp:mass:px”

The symbols used for the type are:

- C: a character string terminated by the 0 character
- B: an 8 bit signed integer
- b: an 8 bit unsigned integer
- S: a 16 bit signed integer
- s: a 16 bit unsigned integer
- I: a 32 bit signed integer
- i: a 32 bit unsigned integer
- L: a 64 bit signed integer
- l: a 64 bit unsigned integer
- F: a 32 bit floating point
- D: a 64 bit floating point

- O: [the letter ‘o’, not a zero] a boolean (Bool_t)

The type is used for a byte count to decide how much space to allocate. The variable written is simply the block of bytes starting at the starting address given in the second parameter. It may or may not match the leaf list depending on whether or not the programmer is being careful when choosing the leaf address, name, and type.

By default, a variable will be copied with the number of bytes specified in the type descriptor symbol. However, if the type consists of two characters, the number specifies the number of bytes to be used when copying the variable to the output buffer. The line below describes ntrack to be written as a 16-bit integer (rather than a 32-bit integer).

“ntrack/I2”

With this Branch method, you can also add a leaf that holds an entire array of variables. To add an array of floats use the f[n] notation when describing the leaf.

Float_t f[10];

tree→Branch(“fBranch”,f,“f[10]/F”);

You can also add an array of variable length:

{ TFile *f = new TFile(“peter.root”,“recreate”);

Int_t nPhot;

Float_t E[500];

TTree* nEmcPhotons = new TTree(“nEmcPhotons”,“EMC Photons”);

nEmcPhotons→Branch(“nPhot”,&nPhot,“nPhot/I”);

nEmcPhotons→Branch(“E”,E,“E[nPhot]/F”);

}

7.9 Adding a TBranch to Hold an Object

To write a branch to hold an event object, we need to load the definition of the Event class, which is in $ROOTSYS/test/libEvent.so (if it doesn't exist type make in $ROOTSYS/test). An object can be saved in a tree if a ROOT dictionary for its class has been generated and loaded.

root[] .L libEvent.so

open a file and create a tree.

root[] TFile *f = new TFile("AFile.root","RECREATE")

root[] TTree *tree = new TTree("T","A Root Tree")

We need to create a pointer to an Event object that will be used as a reference in the TTree::Branch method. Then we create a branch with the TTree::Branch method.

root[] Event *event = new Event()

root[] tree→Branch("EventBranch","Event",&event,32000,99)

To add a branch to hold an object we use the signature above. The first parameter is the name of the branch. The second parameter is the name of the class of the object to be stored. The third parameter is the address of a pointer to the object to be stored. Note that it is an address of a pointer to the object, not just a pointer to the object. The fourth parameter is the buffer size and is by default 32000 bytes. It is the number of bytes of data for that branch to save to a buffer until it is saved to the file. The last parameter is the split-level, which is the topic of the next section. Static class members are not part of an object and thus not written with the object. You could store them separately by collecting these values in a special "status"object and write it to the file outside of the tree. If it makes sense to store them for each object, make them a regular data member.

7.9.1 Setting the Split-level

To split a branch means to create a sub-branch for each data member in the object. The split-level can be set to 0 to disable splitting or it can be set to a number between 1 and 99 indicating the depth of splitting.

To split a branch means to create a sub-branch for each data member in the object. The split-level can be set to 0 to disable splitting or it can be set to a number between 1 and 99 indicating the depth of splitting.

When the split-level is 1, an object data member is assigned a branch. If the split-level is 2, the data member objects will be split also, and a split level of 3 its data members objects, will be split. As the split-level increases so does the splitting depth. The ROOT default for the split-level is 99. This means the object will be split to the maximum.

Splitting a branch can quickly generate many branches. Each branch has its own buffer in memory. In case of many branches (say more than 100), you should adjust the buffer size accordingly. A recommended buffer size is 32000 bytes if you have less than 50 branches. Around 16000 bytes if you have less than 100 branches and 4000 bytes if you have more than 500 branches. These numbers are recommended for computers with memory size ranging from 32MB to 256MB. If you have more memory, you should specify larger buffer sizes. However, in this case, do not forget that your file might be used on another machine with a smaller memory configuration.

A split branch is faster to read, but slightly slower to write. The reading is quicker because variables of the same type are stored consecutively and the type does not have to be read each time. It is slower to write because of the large number of buffers as described above. See “Performance Benchmarks” for performance impact of split and non-split mode.

When splitting a branch, variables of different types are handled differently. Here are the rules that apply when splitting a branch.

• If a data member is a basic type, it becomes one branch of class TBranchElement.

• A data member can be an array of basic types. In this case, one single branch is created for the array.

• A data member can be a pointer to an array of basic types. The length can vary, and must be specified in the comment field of the data member in the class definition. See “Input/Output”.

• Pointer data member are not split, except for pointers to a TClonesArray. The TClonesArray (pointed to) is split if the split level is greater than two. When the split level is one, the TClonesArray is not split.

• If a data member is a pointer to an object, a special branch is created. The branch will be filled by calling the class Streamer function to serialize the object into the branch buffer.

• If a data member is an object, the data members of this object are split into branches according to the split-level (i.e. split-level > 2).

• Base classes are split when the object is split.

• Abstract base classes are never split.

• All STL containers are supported.

• C-structure data members are not supported in split mode.

• An object that is not split may be slow to browse.

- A STL container that is not split will not be accessible in the browser

7.9.2 Exempt a Data Member from Splitting

If you are creating a branch with an object and in general you want the data members to be split, but you want to exempt a data member from the split. You can specify this in the comment field of the data member:

class Event : public TObject { private: EventHeader fEvtHdr; // Don't split the header

7.9.3 Adding a Branch to Hold a TClonesArray

ROOT has two classes to manage arrays of objects. The TObjArray can manage objects of different classes, and the TClonesArray that specializes in managing objects of the same class (hence the name Clones Array). TClonesArray takes advantage of the constant size of each element when adding the elements to the array. Instead of allocating memory for each new object as it is added, it reuses the memory. Here is an example of the time a TClonesArray can save over a TObjArray. We have 100,000 events, and each has 10,000 tracks, which gives 1,000,000,000 tracks. If we use a TObjArray for the tracks, we implicitly make a call to new and a corresponding call to delete for each track. The time it takes to make a pair of new/delete calls is about 7 s (10-6). If we multiply the number of tracks by 7 s, (1,000,000,000 * 7 * 10-6) we calculate that the time allocating and freeing memory is about 2 hours. This is the chunk of time saved when a TClonesArray is used rather than a TObjArray. If you do not want to wait 2 hours for your tracks (or equivalent objects), be sure to use a TClonesArray for same-class objects arrays. Branches with TClonesArrays use the same method (TTree::Branch) as any other object described above. If splitting is specified the objects in the TClonesArray are split, not the TClonesArray itself.

7.9.4 Identical Branch Names

When a top-level object (say event), has two data members of the same class the sub branches end up with identical names. To distinguish the sub branch we must associate them with the master branch by including a "."(a dot) at the end of the master branch name. This will force the name of the sub branch to be master.sub branch instead of simply sub branch. For example, a tree has two branches Trigger and MuonTrigger, each containing an object of the same class (Trigger). To identify uniquely the sub branches we add the dot:

tree→Branch("Trigger.","Trigger",&b1,8000,1);

tree→Branch("MuonTrigger.","Trigger",&b2,8000,1);

If Trigger has three members, T1, T2, T3, the two instructions above will generate sub branches called: Trigger.T1, Trigger.T2, Trigger.T3, MuonTrigger.T1, MuonTrigger.T2, and MuonTrigger.T3.

7.10 Adding a Branch with a Folder

add a branch from a folder

tree→Branch("/aFolder");

This method creates one branch for each element in the folder. The method returns the total number of branches created. "event"is a structure with one float and three integers and one unsigned integer. You should not assume that the compiler aligns the elements of a structure without gaps. To avoid alignment problems, you need to use structures with same length members. If your structure does not qualify, you need to create one branch for each element of the structure.

7.11 Adding a Branch with a Collection

This Branch method creates one branch for each element in the collection.

tree→Branch(*aCollection, 8000, 99);

// Int_t TTree::Branch(TCollection *list, Int_t bufsize,

// Int_t splitlevel, const char *name)

The method returns the total number of branches created. Each entry in the collection becomes a top level branch if the corresponding class is not a collection. If it is a collection, the entry in the collection becomes in turn top level branches, etc. The split level is decreased by 1 every time a new collection is found. For example if list is a TObjArray*

- If splitlevel = 1, one top level branch is created for each element of the TObjArray.
- If splitlevel = 2, one top level branch is created for each array element. If one of the array elements is a TCollection, one top level branch will be created for each element of this collection.

In case a collection element is a TClonesArray, the special Tree constructor for TClonesArray is called. The collection itself cannot be a TClonesArray. If name is given, all branch names will be prefixed with name_.

IMPORTANT NOTE1: This function should not be called if splitlevel< 1.

IMPORTANT NOTE2: The branches created by this function will have names corresponding to the collection or object names. It is important to give names to collections to avoid misleading branch names or identical branch names. By default collections have a name equal to the corresponding class name, e.g. the default name of TList is “TList”.

8 ROOT in Python

9 Concluding Remarks

packages named RooFit and RooStats providing an advanced framework for model building, fitting and statistical analysis. The ROOT namespace TMVA offers multi-variate analysis tools including an artificial neural network and many other advanced tools for classification problems. The remarkable ability of ROOT to handle large data volumes was already mentioned in this guide, implemented through the class TTree.