A `root` Cheat Sheet

A. Stephen Beach

June 9, 1998

Abstract

This is a quick guide to `root` in a question and answer format. Its target audience is a physicist who knows a little bit of computer programming, but has no experience with `root` or C++. Its goal is to get the user up and running quickly.

Contents

1 Introduction

1.1 Who wrote this, and how can I contact him to say that the !/#$%@ answer wasn't correct?

This cheat sheet was written by Steve Beach, a graduate student working in experimental particle astrophysics at Penn State University. He has, like any graduate student, got far too much stuff to do, so be gentle in the corrections, please. He can be reached best by e-mail at `asb4@psu.edu` , and also has a web site (mostly with pictures of his darling daughter!) at `http://landau.phys.psu.edu/people/beach/Untitled.html`.

But seriously, any and all additions to this document will be appreciated. If you have answers to my unanswered questions, please send them to me. Ditto for more questions, with or without answers.

Note that this document has been neither edited, proofed, nor blessed by the `root` development team. I have personally run every code example and suggestion in this document, and they have worked for me. However, it still may have mistakes in it, and its distribution and use is entirely at the risk of the user. In other words, you get what you pay for, and if you blow up your computer or delete your data set, since you didn't pay anything, you don't get to sue me.

1.2 What is the intended purpose of this cheat sheet?

Remember when you were in school and the teacher asked you a question you didn't know the answer? Didn't you just really want a sheet with all the quick answers to things you really felt stupid not to know? Well, that's what this is for. Not as an encyclopedic dictionary of minutia, but as a compendium of those answers you really feel stupid not knowing. It can be used as a study guide, helping the newbie get his/her feet wet, but do not use as a reference for getting that graph just right.

Also, this is not an FAQ. I did not comb the `root` newsgroups, although I may do that for a later version if I get enough response.

1.3 Can I distribute this cheat sheet?

Why of course! Why else would I have written it up? This document is hereby copyright 1998 by A. Stephen Beach, and is licensed to the world under terms of the Gnu Public License. As a courtesy to the author, it is requested that I be notified when this document is included in any compilation or CD-ROM. Also, please attempt to get the latest version from my web site listed above, in order to save me from error reports on items that I've already fixed!

2 Basic Questions

2.1 What is `root` ?

`root` is an data analysis and graphics program designed for use in high-energy physics, but adaptable to use in other fields. The source code as well as pre-compiled versions of `root` are available at no cost. Improvements are being made continuously by the hard working, talented, `root` development team of René Brun, Nenad Buncic, Valery Fine, and Fons Rademakers. At the time of this writing, the latest version was 2.00/04, and that is what this cheat sheet is based on.

2.2 Where do I get `root` ?

The best place to get `root` as well as the official documentation is the main web site, http://root.cern.ch.

2.3 What other documentation is available for `root` ?

See the bibliography. Reference [1] is an overview of the system, on a somewhat technical level, appropriate for pitching it to your boss and showing him/her all the nifty things it can do. Reference [2] is a tutorial, or a collection of macros that helps to guide you through the intricacies of `root` . However, it's pretty much a self-guided tour. Reference [3] is the authoritative guide to `root` . If you have a question about what a function does or what arguements a function takes, look there. However, do not print it out – it's over 600 pages long! And, it will change as `root` changes, so it will probably be out of date before you've managed to look at half the pages!

In a slightly different vein is reference [4]. It is an outline of a course in `root` , aiming to be a comprehensive course, but assuming from the students a fair understanding of object oriented programming and C++. It should also be understood as a work in progress, as it ends fairly abruptly.

Finally, there are the `root` howto's, on a wide variety of topics, for example, how to read a tree or how to debug a macro. They are available on the main `root` web site, under "how to's". They are quite comprehensive, but I found them relatively difficult to follow, in part because of their comprehensive scope.

So what does this have to offer instead? First, it is aimed at a lower audience. Second, I believe that the format is better suited for a harried grad student told "Do this!". Third, since I am not a part of the `root` team, I believe I may have a little bit more insight as to what the `root` novice needs to know or is confused about.

2.4 How do I start `root` ?

You start `root` by typing `root` at the system prompt. If everything is working correctly, you will see a logo screen with a lightning bolt on an orange background and the word "`root` " in large type. That will disappear in a couple of seconds, leaving you with a "Welcome to `root` " message in your original screen and a "root prompt". The `root` prompt looks like

```
root [0]
```

This is where you enter your commands. The number inside the brackets [] reflects the number of commands you've entered so far. As a help, if you want to repeat a command, you can use the up arrow key to scroll through the prior commands.

If this doesn't work, you need to check two things:

1. Is the system variable ROOTSYS set, and is it pointing to the correct directory?
2. Is your path correct?

If either or both of these is gibberish, may I suggest either finding a friendly co-worker or learning about UNIX? For help with either, see reference [5].

2.5 How do I end `root` ?

At the `root` prompt, type `.q` or `.qqq`.

2.6 What is a macro? How do I run a macro?

A macro is a set of C or C++ commands enclosed in braces ({ }), stored in a file. These are commands that you could just type into `root` yourself, but would rather not have the typos that come from typing! The braces at the start and end are important, in that they tell `root` the start and finish of the macro.

Running a macro is simple: at the `root` prompt, type `.x macro-name`, where `macro-name` is the file name of the macro, exactly as it is typed on your system. So, on Linux, if you wanted to run the macro “test”, in your home directory, you would type `.x test` if you started `root` from your home directory, or `.x  /test` if you started it somewhere else.

As an aside, you can use “meta-macros”, at least on macros. For example, say you wanted to run the macro “test”, and then exit, saving all the output to the file “test.output”. First, make up a text file named “runtest” with the following contents:

```
.x test
.q
```

Then, at the system prompt (i.e., before starting `root`), type

```
root < runtest > test.output
```

`root` will then execute all the commands of the “test” macro, save all the results to the file “test.output”, and quit.

2.7 How can I run `root` in batch mode?

There are at least three ways to do this. The first is the “meta-macro” method of question 2.6.

The second way is the “official” way, using command line options. There are four command line options on version 2.00 of `root` :

- -b: run in batch mode without graphics
- -q: exit after running all of the macros listed on the command line
- -l: do not show the “splash” screen (i.e., the lightning bold logo)
- -n: do not run the logon and logon macros in `.rootrc`
- -?: list these options

For example, say you wanted to run the macro `test`, save the results and exit. At the system command prompt, you would enter

```
root -b -q test > testout
```

The third method is to compile your macros. While this is the most difficult method, the macros do run faster. Much faster. Perhaps by a factor of 100 sometimes.

2.8 How can I debug a macro?

By using the `root` environment wisely. There are breakpoint and step-by-step commands, but I've never figured out how to use them. I'm leaning towards doing everything complicated enough to need debugging as a compiled program, so that the compiler and linker can check for bugs.

2.9 How do I convert a `root` macro into a compiled application?

If you've adhered to all C++ syntax (and not used any of the `root` interpreter extensions mentioned in question 3.1), it should be pretty easy. Basically, all you've got to do is put in the header files, a `main` function to call the rest, make two small additions to the macro, and then compile it. There are just enough subtleties that I'll expound further on each.

The top matter: You've got to tell the compiler a couple of things. First, where to find the declarations for all the classes and libraries that you use. So, include the following lines at the top:

```
#ifndef __CINT__
#include <stdio.h>
#include <math.h>
#include ``TROOT.h''
#include ``TClass.h''
#include ``TClassTable.h''
#include ``TCollection.h''
#include ``TObject.h''
#endif
```

The `#ifndef` and `#endif` lines are to leave your macro executable by as a macro. (I.e., the lines in between will be ignored if you are running `__CINT__`, the C interpreter that runs macros.) You may have to add more lines here. Compile your code, see what classes the compiler doesn't understand, and then add `#include ``T`, the class name, then `.h''`.

Next in the top matter, you've got to include a global object `TROOT`, as well as some other type declarations, perhaps as

```
#ifndef __CINT__
extern void InitGui();
VoidFuncPtr_t initfuncs[] = {InitGui, 0};
TROOT root("hello","Hello World",initfuncs);
#endif
```

The next thing to add at the top is a call to the function `main` that every C++ program must have. Remember that your macro is surrounded by braces? Perfect for adding the `main` function name. Right before the start of your macro, insert

```
#ifndef __CINT__
int main(int argc, char **argv)
#endif
```

Now, there is only two changes you need to make in your macro. If you want any sort of GUI in your program, add the following as the first executable statement:[1]

```
#ifndef __CINT__
TApplication theApp("App",&argc,argv);
#endif
```

If you want the user to be able to interact with the GUI, put the following lines in at the appropriate location. Note that all other processing will cease until the user chooses to exit `root` . If you want to continue processing after the user has exitted the GUI, change it to `theApp.Run(kTRUE);` .

```
#ifndef __CINT__
theApp.Run();
#endif
```

Last and probably least, you've got to insert a return at the end of the code. As the last executable statement, add

```
#ifndef __CINT__
return 0;
#endif
```

Finally, the compiling. There are so many flags and include stuff here that you really shouldn't even try to do it without using make. So, when you write up your Makefile, all you have to do is steal from the one they so nicely gave you, located in $(ROOTSYS)/test/Makefile. But if you can't find that, here's the flags you need.

For compiling, `-fPIC -Wall -I$ROOTSYS/include`

For linking, `-L$(ROOTSYS)/lib -lNew -lBase -lCint -lClib -lCont -lFunc -lGraf -lGraf3d -lHist -lHtml -lMatrix -lMeta -lMinuit -lNet -lPostscript -lProof -lRint -lTree -lUnix -lZip -lGpad -lGui -lGX11 -lX3d -L/usr/X11R6/lib -lXpm -lX11 -lg++ -lm -ldl -rdynamic`
Note that I'm running Linux, so if you're not, some of the X11 libaries etc. might differ.

2.10 Is there any slick front end for `root` , like PAW++?

Not at the moment, but give it a chance! `root` is only a couple of years old, and the development team is wisely making it functional first, idiot-proof second.

[1]If you want the root command line to be available to the user (handy for playing with derived objects), substitute `TRint` for `TApplication` .

3 The `root` Language

3.1 What is all of this funny syntax in the tutorial macros? I know C, but I've never seen this stuff before!

You've never seen it before because you haven't seen C++ before. C++ is an object oriented programming language, and the concept of objects is at the heart of C++. You do *not* have to be a C++ wizard to run `root` effectively, but you should know enough to realize what the syntax is doing. Towards that end, here are some code examples with some explanation of the funny capitalization, parenthesis, and arrows.

```
hpx->Fill(px);
```

If you are a C programmer, you might recognize this as a reference to a member of a structure pointed to by the pointer `hpx`. This is C++, however, and this is a class, not a structure. So, functions are allowed to be members of a class, and this statement means that you are calling the function `Fill`, with no arguments, as part of the object pointed to by the pointer `hpx` (loosely, the object `hpx`).

```
TH1F *hpx = new TH1F("hpx","This is the px distribution",100,-4,4);
```

This is another instance of classes making your life easier, even if it doesn't look like it. The first part the command, `TH1F *hpx`, is a type declaration, declaring `hpx` to be a pointer to an object of type `TH1F`. In C, you would have been forced to put this at the top of the program, but C++ gives you the freedom to put the declaration right when you need it.

The `new` keyword is a C++ innovation, replacing `malloc()`. It not only allocates the space for a `TH1F` object, but it checks for running out of space, as well as allowing the programmer to do intelligent things when a new object is set up. Those intelligent things are governed by the parameters listed inside the parenthesis. The meanings of these arguments are given in the `root` class documentation [3].

Finally, one notation you'll see in the documentation (not so much in the code) is the "class scope reslution operator" `::` , as in `TF1::SetParLimits` . What that means is that "SetParLimits" is a member of the class "TF1". Kind of handy, actually.

3.2 Are there any syntax shortcuts?

Yes, `root` does provide shortcuts, but I prefer not to use them, for two reasons. First, I'm still learning C++, and I think using the long versions will help me get the hang of the syntax and the thought process. Second, I want to write code that can be easily compiled, if necessary. The syntax easings are purely for the `root` prompt interpreter, not for the compiler.

With those caveats, I now present the syntax shortcuts. First, you don't have to use the full syntax when creating an object. The following code fragments are equivalent:

```
TFile fill(''fillrandom.root'');
TFile *fill = new TFile(''fillrandom.root'');
```

Second, when referring to a member function, you don't have to use the redirection operator: `fill.ls();` is allowed, instead of `fill->ls();`.

Third, when typing in a single line statement, you don't need the ; at the end. Say, for example, you are at the root prompt, and you've got your ntuple all set up. To draw it, you can leave the semicolon off of `ntuple->Draw(``x'')` But, obviously, if you're doing it on in a macro or a compiled program, put the semicolon in. Myself, I always put it in, since I don't want to get into bad habits!

Fourth, when referring to an object in a file that has been loaded but not yet used, you don't have to find it first. For example, if the ntuple `ntuple` is in the file `file.root` , you can draw it by typing

```
f = new TFile(``file.root'');
ntuple->Draw(``x'');
```

Without the interpreter shortcuts, you would have to type

```
TFile *f = new TFile(``file.root'');
TNtuple *ntuple = (TNtuple *) gROOT.FindObject(``ntuple'');
ntuple->Draw(``x'');
```

3.3 What naming conventions are used in `root` ?

First, a general rule: in any name made from more than one word, the second and later words are always capitalized, without underscores. For example, use `THashTable`, not `T_hash_table`.

Second, different types have different standard identifiers. The following table is taken from reference [4]:

Identifier	Convention	Example
Classes	Begin with T	THashTable
Non-class types	End with _t	Simple_t
Enumeration types	Begin with E	EColorLevel
Data members	Begins with f for field	fViewList
Member functions	Begin with a capital	Draw()
Static variables	Begin with g	gSystem
Static data members	Begin with fg	fgTokenClient
Locals and parameters	Begin with lower case	seed
Constants	Begin with k	kInitialSize
Template arguments	Begin with A	Atype
Getters and setters	Begin with Get, Set, or Is (boolean)	SetLast(), GetFirst(), IsDone()

3.4 What are the data types `Float_t`, `Int_t` etc., and how are they different from the usual C data types?

Since `root` has been ported to many different systems, with many different compilers and processors, there is no way of knowing, a priori, how many bytes are in an `int` , a `float` , or whatever. In order to get the same results no matter what machine it's run on, the creators of `root` defined machine independent basic types. Here is a table of what exactly they are (taken from [4])

Type	Meaning
Char_t	Signed, 1 byte
UChar_t	Unsigned, 1 byte
Short_t	Signed integer, 2 bytes
UShort_t	Unsigned integer, 2 bytes
Int_t	Signed integer, 4 bytes
UInt_t	Unigned integer, 4 bytes
Long_t	Signed integer, 8 bytes
ULong_t	Unsigned integer, 8 byte
Float_t	Floating point, 4 bytes
Double_t	Floating point, 8 bytes
Bool_t	Boolean, taking values kTRUE, kFALSE

4 Histograms and Ntuples

4.1 What is a histogram?

A histogram is a graph showing how many events fell in a certain range. For example, 1 event had an angle between 0 and 1 degree, 3 events were between 1 and 2 degrees, etc. Very, very handy graph for experimental physics.

4.2 What is an ntuple?

An ntuple, in mathematical usage, is a group of n numbers. For example, a point on a plane corresponds to a ntuple of two numbers. For experimental physics, this concept has been broadened and specialized, so now "ntuple" refers to a collection of data about an experiment, where each event is a collection of numbers that describes the event (for example, time, temperature, amount of energy seen), and there are many events in the collection. Specifically for `root` , an ntuple is a data set with a specific format, an object of the class `TNtuple` .

Since an ntuple is such a basic type of data for experimental physics analysis, the creaters of `root` have made it easy to go from the data in an ntuple to the analysis of that data. Histograms, fitting, etc. are all relatively easy to accomplish in `root` . If you don't believe it, try doing it on your own!

4.3 How do I draw a histogram from an ntuple? How do I set cuts for an ntuple?

The easiest way, by far, is to just draw it directly. `root` will set the bounds, make a decent choice for the style, and then do it very quickly. The point is, you don't have to create a histogram and fill it.

So, say you have an ntuple called (without originality) `ntuple` , with the variables x and y. To draw a 1-D histogram showing the distribution of x, type

```
ntuple->Draw(``x'');
```

Pretty simple, huh? To draw a 2-D histogram scatter plot, you've got to do just a little more, in order for you to see the markers.[2]

```
ntuple->SetMarkerColor(1);
ntuple->Draw(``x:y'');
```

Note that the above graphs "x" along the vertical axis. I guess whatever is in front of the colon gets graphed as a function of whatever's after.

The extension to 3-D is obvious, although you should note that the default is to *not* show axes.

As an aside, note that you don't have to use just the variables in the ntuple. You can use any functions of them, as in `ntuple->Draw(``x*y + 4:y*y'');` You can also save your histogram to a named histogram by ending the argument with `>>h1` , as for example `ntuple->Draw(``x*y + 4:y*y>>h1'');` . If you want to add the data from this ntuple into another histogram (maybe you want a histogram based on five files), but a "+" before the name of the existing histogram, as in `ntuple->Draw(``x*y + 4:y*y>>+h1'');`

Now what about cuts? Cuts are handled about the most obvious way imaginable, as another argument to the `Draw()` member function. If the second argument is a boolean operator (i.e., returns either 0 or 1), then if the value is 1 for an event, that event is included. If the value is 0, the event is cut. This can be any expression, as long as it evaluates to either 0 or 1. For example, `ntuple->Draw(``x'', ``y>.5'');` will draw a 1-D histogram of x, including only those events for which y is greater than 0.5. Cuts can be chained together, using standard C syntax: `ntuple->Draw(``x'', ``(y>.5) && (x>.5)'');` will draw a

[2]Defaults aren't root's strong point.

1-D histogram of x, taking only events for which y is greater than .5 and x is greater than .5. Use a lot of parenthesis for clarity!

Options to the histogram can be put into a third argument to `Draw()` . The same options are supported as for histograms drawn directly (see question 5.1.

Finally, there are optional fourth and fifth arguments. The fourth is the number of events to use, and the fifth is the starting event. The defaults are "all" and "first", so usually, you won't need these.

4.4 Somebody just gave me a `root` file containing an ntuple. How do I find out what variables are in the ntuple?

First, load the file, as shown in question 7.1. Let's say the name of the file is `fts` . Second, find the name of the ntuple or tree in the file using `fts->ls();` . That will give you a list of all the histograms and the name of the tree/ntuple. Let's say that the tree is named `h80` . To find the names of the variables, type `h80->Print();` .

Another option for exploring the ntuple is to use a browser. Very GUI, but usable nonetheless. To do that, type in at the `root` prompt

```
TBrowser *browse = new TBrowser(``browse'', ``This is a browser'');
```

Shortly, a window will pop up, looking very much like a Windows95 selection window[3]. In the box on the left, way down at the bottom, will be a folder labeled "ROOT Files". Double click on it, and you'll get a tree of all the `.root` files in your current directory. Double click on any of those files, and you'll get a list of the objects contained in the file. Double click on an object that's an ntuple or a tree, and you'll get (in the right hand window) a list of the variables in that ntuple. A long procedure to explain, but when you do it, it will all be obvious.

4.5 How can I manipulate an ntuple?

Another way of asking this question is *"How do I traverse an ntuple, looking at one event at a time?"* Once you've got a loop set up to traverse the ntuple, you can use ordinary C statements to do what you want – cut it, save only some variables to another ntuple, create some new variables, etc.

Traversing a tree is such an ordinary thing to do that the creators of `root` wrote a function to make it easier. Say you've got an ntuple named `fred` , and you want to look at each event in turn. Rather than writing the code yourself, execute `fred->MakeCode(``fred.cc'');` at the `root` prompt. That will put a skeleton macro in the file `fred.cc` , that you can then edit to do what you want.

[3]Convergent evolution?

As an example, I had an ntuple named `ntuple` with the variables x, y, and z. When I ran `ntuple->MakeCode(‘‘ntuple.cc’’);` , this is what I got:

```
{
//////////////////////////////////////////////////////////
//   This file has been automatically generated
//     (Fri Apr 10 11:50:05 1998 by ROOT version 2.00/04)
//   from TTree ntuple/data from ascii file
//   found on file: basic.root
//////////////////////////////////////////////////////////


//Reset ROOT and connect tree file
   gROOT->Reset();
   TFile *f = new TFile("basic.root");
   TTree *ntuple = (TTree*)f->Get("ntuple");

//Declaration of leaves types
   Float_t         x;
   Float_t         y;
   Float_t         z;

//Set branch addresses
   ntuple->SetBranchAddress("x",&x);
   ntuple->SetBranchAddress("y",&y);
   ntuple->SetBranchAddress("z",&z);

//     This is the loop skeleton
//       To read only selected branches, Insert statements like:
// ntuple->SetBranchStatus("*",0);  // disable all branches
// ntuple->SetBranchStatus("branchname",1);  // activate branchname

   Int_t nentries = ntuple->GetEntries();

   Int_t nbytes = 0;
//   for (Int_t i=0; i<nentries;i++) {
//      nbytes += ntuple->GetEvent(i);
//   }
}
```

Now, if I want to do something simple, like print all the variables to the screen, all I have to do is change the last `for` loop, uncommenting it, and replacing it with

```
   for (Int_t i=0; i<nentries;i++)
      {
         nbytes += ntuple->GetEvent(i);
         printf(``x,y,z= %f, %f, %f \n'',x,y,z);
      }
```

Notice that after I've gotten the event, I can refer to the variables just by their name, nothing fancy required. Execute this like any other macro, as `.x ntuple.cc` at the `root` prompt.

So, manipulating one ntuple into another is easy. Traverse it as above, but declare another ntuple, and instead of `printf` , calculate the variables you want, and then use a `Fill` into the new ntuple.

4.6 How do I create a new ntuple?

This is a three step process. First, you create it, by the C++ command `new` , which calls the "constructor" of the class:

```
TNtuple *ntuple = new TNtuple(``ntuple'', ``This is an ntuple'', ``x:y:z'');
```

The first argument is the name of the ntuple. The second argument is a short description of the ntuple – makes debugging a file a lot easier! The last argument is a list of the data in the ntuple – names separated by colons. Please give descriptive names, as these names are stored with the ntuple, and will greatly ease the life of the next user.

The second step is to fill the ntuple. This is absurdly easy. First, get the data you want, and then say

```
ntuple->Fill(x,y,z);
```

The last step is to save it to a file. For that, see question 7.2.

4.7 How do I create a new histogram?

Pretty much the same way you create an ntuple, by using the constructor of the class, and then filling in the data. The only hard part is that there is an incredible selection of histogram types. First, histograms can be 1, 2 or 3 dimensional. Second, the values stored can be character (one byte integers), short (2 byte integers), floats, or doubles. The names of the histogram classes reflect the different types available. They all start `TH` , and then add on the dimension (1, 2 or 3) and the data type (C, S, F, or D). So a 1 dimensional floating point histogram is in the class `TH1F` .

Actually creating a histogram is done by calling 'new' and the name of the class, with arguments to specify the data range, name, header, etc.

5 Formatting Questions

5.1 How do I tell `root` what style to use when drawing a histogram?

There are two ways. First, the user can play with the GUI canvas after the drawing has been done. She can change the style, the colors, the markers, add text, just about anything. Second, the options can be specified when drawing the histogram. Naturally, there are different options for 1-D, 2-D, etc., but to an amazing extent, they overlap. To use any of these options, make them an argument to the `Draw()` class member, as in `histo->Draw(''box'');` . (Yes, the quotes are necessary.

Some major options for all histograms are

- `same` – superimpose on the previous picture
- `cyl, pol, sph, psr` – use cylindrical, polar, spherical or pseudorapidity/phi coordinates
- `lego` – draw a lego plot (variations include `lego1, lego2`)
- `surf` – draw a surface plot (variations include `surf1` through `surf4`)

Some of the options for 1-D histograms are

- `A` – do not draw axis labels and tick marks
- `B` – draw a bar chart, which shows in a contrasting color under the graph line, rather than just outlining it
- `C` – draw a smooth curve, rather than a bar chart outline
- `E` – draw error bars (variations include `E1` through `E4`)
- `L` – draw a line through the bin contents
- `P` – put a marker at each bin, rather than a line
- `*` – put a '*' marker at each bin, rather than a line. (Note that I couldn't get this one to work.)

Some of the options for 2-D histograms are

- `scat` – draw a scatter plot (default). This has a funny interpretation that may take some getting used to. Recall that what you're doing is drawing the histogram, and the histogram only stores processed data, not the individual points. So, when it does a scatter plot, it can't plot each point as it was input, it only puts a point randomly in the interval of the bin.
- `box` – draw a box whose size is proportional to the number of points in the bin
- `col` – draw a box for each bin, with different colors for different numbers of points
- `cont` – draw a contour plot (variations include `cont0` through `cont3`)

As an example of what these do, I've done a generic 1-D and 2-D histogram, and drawn four sample styles in figures 1 and 2. Enjoy!

5.2 What about other options, like axis titles, overall titles, and statistics?

Overall titles are automatically put in. The text is given when the histogram is created. If the histogram is drawn from an ntuple, the title is the name of the ntuple. Any other comments you want to add can be put in with the `TPaveLabel` class. See the file $(ROOTSYS)/tutorial/h1draw.C for some examples.

You can put an x grid on the background with the command (assuming you're working on pad1) `pad1->SetGridx();` (obvious changes for a y grid). This can also be done with the entire canvas: `c1->SetGridx();` .

To put a title on the x axis, use the command (for histogram h1) `h1->SetXTitle(''x axis'');` . Obvious changes for y and z axes. Sometimes, there is a problem with the title running into the labels. You can move the title out of the way with the command `h1->SetTitleOffset(1.1,''X'')` . (Obvious change for the Y and Z axes.) The default offset is 1.0, and increasing it moves the title away from the axis. Going over about 1.7 will move the title off the graph entirely, so be careful. Another way to get the proper clearance is move the labels. The command `h1->SetLabelOffset(-0.1,''X'')` will move the labels up (closer to the axis) slightly.

Simple statistics (name, number of entries, mean and RMS) are the default for 1-D histograms. To change which statistics are shown, use the command `gStyle->SetOptStat(x)` , where x is a string of 1's and 0's that specify which statistics to print. The choices, in order are

- Number of underflows
- Number of overflows

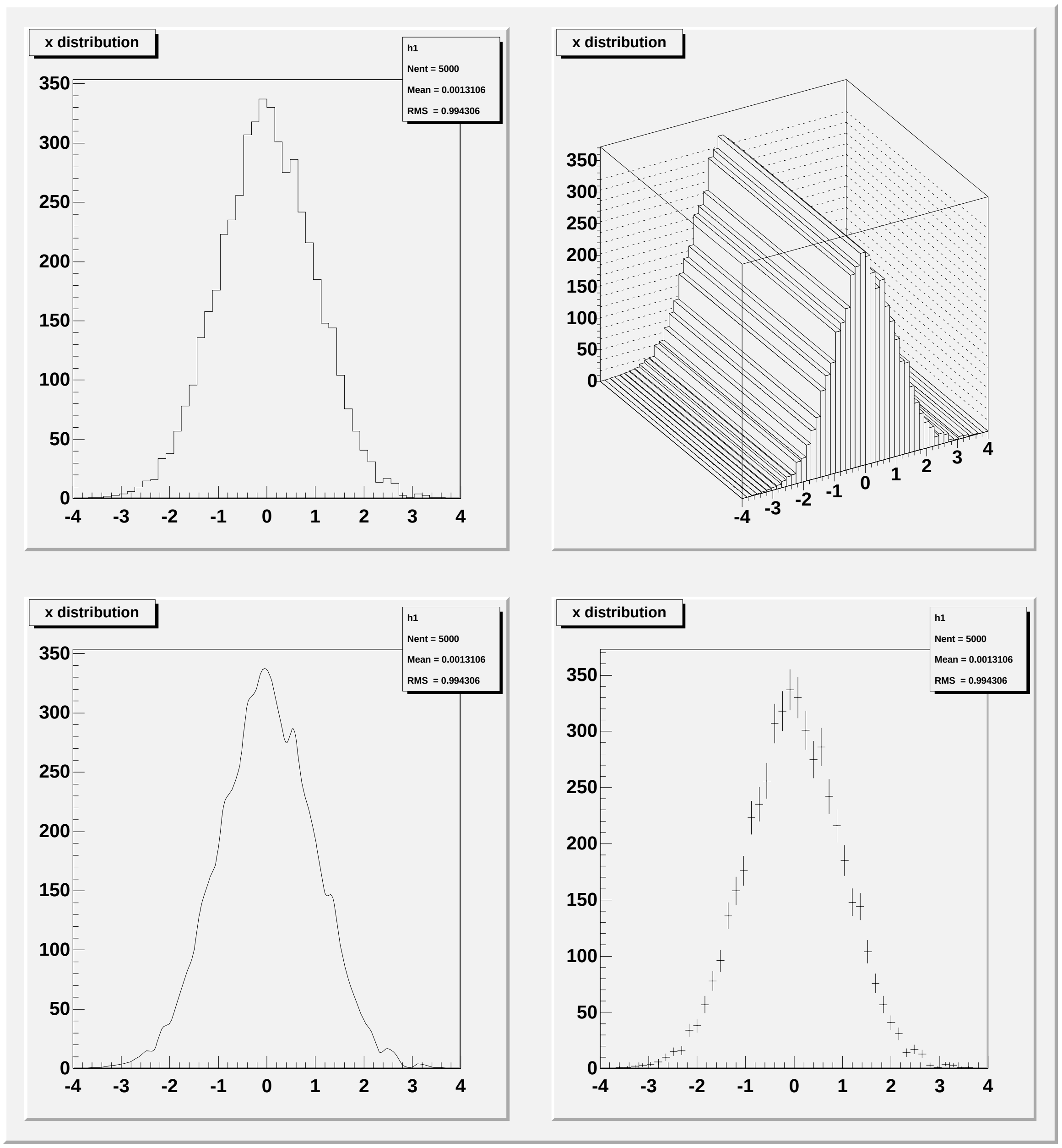

Figure 1: 1-D sample histogram styles. Clockwise from upper left: default, lego, e, c.

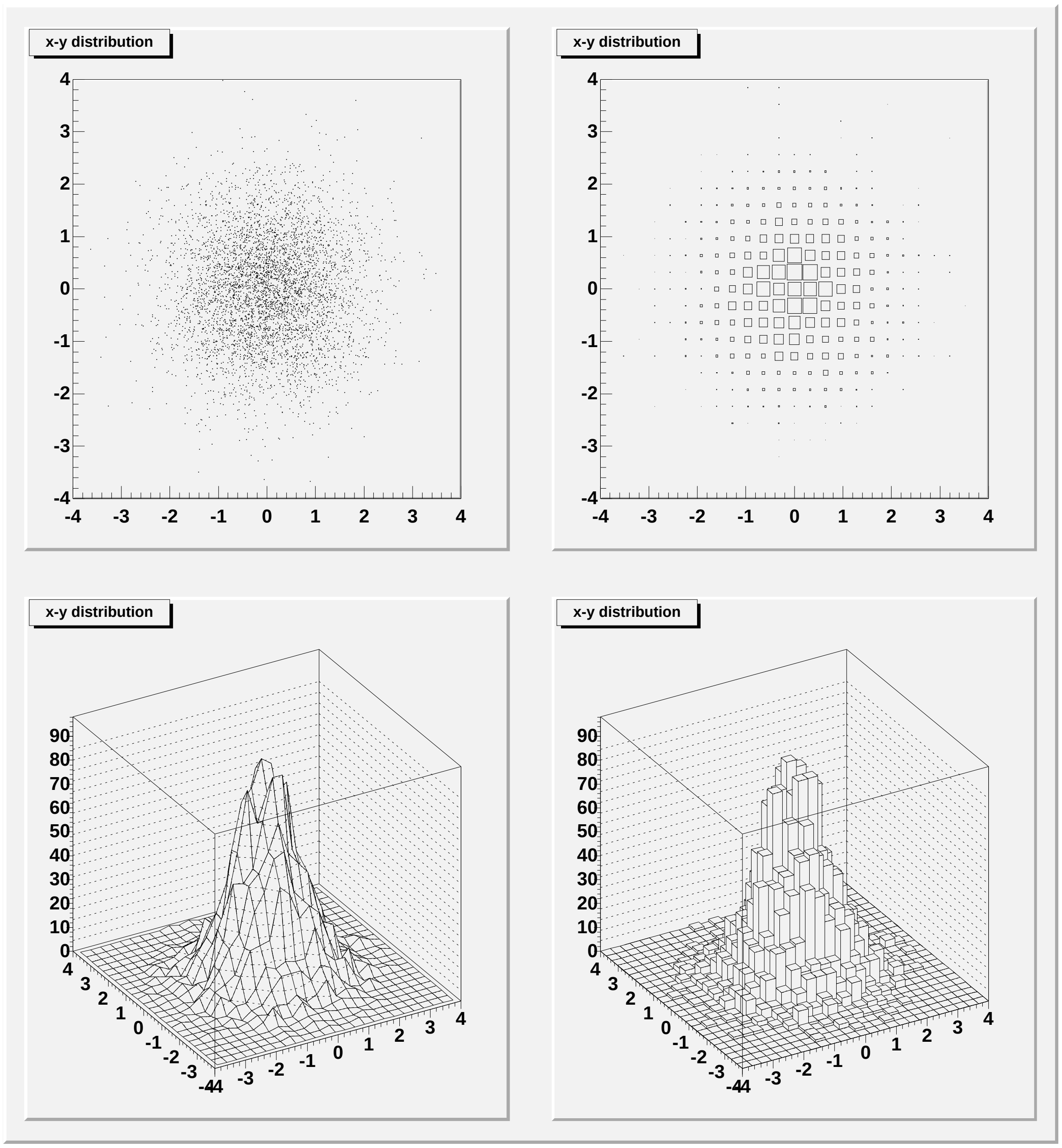

Figure 2: 2-D sample histogram styles. Clockwise from upper left: scat, box, lego, surf.

- Root mean square deviation
- Arithmetic average
- Number of entries
- Name of object

So, if you wanted all of these, do `gStyle->SetOptStat(111111)` . If you only want the RMS deviation and average, do `gStyle->SetOptStat(1100)` . Do not enter any leading zeros, as they just screw everything up.

Other attributes of the statistics box can also be set with `gStyle` . `SetStatColor` will set the background color of the box. `SetStatX(a)` and `SetStatY(b)` will put the top right corner of the statistics block at (a,b), where a is a number between 0 (far left) and 1 (far right). `SetStatH(a)` and `SetStatW(b)` set the height and width as a fraction of the total height.

Other drawing options are the marker style and color, the fill color and style, and the line color and style. All colors are referred to as integers from 1 to 50. To see what these colors mean, go to a canvas, and from the menu, click on "view" then "colors". To set a color, type

```
h1->SetFillColor(2);
h1->SetMarkerColor(2);
```

etc. Marker styles (what you get if you have a 2-D scatter plot or a 1-D plot with error bars) are numbered from 1 to 30. To see them, go to a canvas, and from the menu, click on "view" then "markers". To set a style, type

```
h1->SetMarkerColor(2);
h1->SetMarkerSize(0.7);
```

The size is relative to the default. Play until it looks right.

If you want to set the background color of a canvas and any pads it has, you can do so with

```
c1->SetFillColor(42);
c1->GetFrame()->SetFillColor(21);
c1->GetFrame()->SetBorderSize(6);
c1->GetFrame()->SetBorderMode(-1);
```

The last two lines make the border look pretty. I have no idea what other options look like.

5.3 Is it possible to display only a part of a histogram? For example, only the part of the histogram with x between 1 and 2?

Yes and no. You can set the range of a histogram to just the "interesting part", but doing so will require resetting the histogram and filling it up again. To do so for histogram `h1` , type

```
h1->Reset();
h1->SetBins(25,1,2);
```

This will reset the x axis so it has 25 bins, going from 1 to 2. If you add three more arguments, you can reset the y axis bins.

Now, say you've got an ntuple that you're going to draw a histogram from. Can you set the bins for that? Yes and no. The simple way is to just do a cut that preserves only the interesting region. However, you still get the default number of bins. If you want to go to all the work, you could traverse the ntuple (see question 4.5) and fill in a histogram manually.

5.4 How do I get a logarithmic histogram?

The property of having an axis with logarithmic scaling is a function of the pad that the graph is on. The way of setting this (assuming pad1) is with `pad1->SetOptLogy(1);` . To turn off logarithmic scaling, change the 1 to a 0. To set it for other axes, change Logy to Logx or Logz.

5.5 How do I set the options on histograms drawn from an ntuple?

Pretty much the same way you set them on a direct histogram. Say you've got an ntuple named `ntuple` , and you want to draw a killer histogram from it.

```
ntuple->SetFillColor(2);
ntuple->SetMarkerStyle(21);
ntuple->Draw(``x'', ``x>-5000'', ``e1p'');
```

The second argument is a "false cut", since you've got to have a place holder there. The point is that you can set the options just as if it were a histogram.

6 Doing Stuff With `root`

6.1 How can I define a formula? How do I define a function?

A little bit of `root` terminology here: a "formula" is a string of characters that when parsed as a C expression, with inputs of "x", "y" and/or "z", will return a number (of type double).[4] A "function" is a formula, with the difference that it has parameters, whose values can be varied, as well as having a range of values for x, y, and/or z. So, the real difference is that a formula will give a number when evaluated, but a function can be drawn and fitted to data.

You can define and evaluate a formula pretty simply:

```
TFormula *form1 = new TFormula(``form1'', ``abs(sin(x)/x)'');
form1->eval(10.0);
```

Functions are also pretty easy. The classes are `TF1, TF2` and `TF3` , corresponding to 1, 2 and 3 dimensional functions. A simple function without any parameters would be defined and drawn by

```
TF1 *func1 = new TF1(``func1'', ``abs(sin(x)/x)'',0,10);
func1->Eval(0.56);
func1->Draw();
```

Note that you have to provide upper and lower limits for the function. For a 2-D function, you've gotta provide all the limits:

```
TF2 *func2 = new TF2(``func2'', ``abs(sin(x)/x)*y'',0,10,-2,2);
func2->Draw(``lego'');
```

But functions are a lot more exciting than that, since you can put in adjustable parameters, allowing for fitting. For example,

```
TF1 *f3=new TF2(``func3'',``[0]*sin(x)+[1]*sin(2*x)+[2]*sin(3*x)'',0,6.28);
f3->SetParameter(0,1.1);
f3->SetParameter(1,2.1);
f3->SetParameter(2,3.1);
f3->Draw(``lego'');
```

Note that before you draw it, you have to give values to the parameters. If you like, you could also give names to the parameters via `f3->SetParName(0,''a0'');` .

`root` has also defined a bunch of pre-defined functions, all the functions that physicists normally use and abuse, including gaussian, exponential, and polynomial. The gaussian function is called `gaus` , and is called something like

```
TF1 *func4 = new TF1(``func4'', ``gaus(0)+[3]*sin(3*x)'',0,6.28);
func3->SetParameter(0,1);
func3->SetParameter(1,0);
```

[4] A round of applause here, please, for the coder of the TFormula class, Nicolas Brun, age 18!!!. Heck, at 18, I was just wasting time!

```
func3->SetParameter(2,1);
func3->SetParameter(3,3.1);
func3->Draw();
```

The '0' in "gaus(0)" means to take the three parameters of the gaussian function starting at spot zero. The three parameters are amplitude, location, and width.

The exponential function is named `expo` , and has two parameters, used as `expo(x) =` $e^{p_0+p_1*x}$.

Polynomial functions are named `pol0, pol1, pol2` , etc, for polynomials of degree 0, 1, 2, etc. For degree N, there are N+1 parameters to specify.

6.2 How do I do a fit with `root` ?

This is pretty easy. If you want to use one of the pre-defined functions, and you are willing to accept `root` 's defaults, it's a one liner:

```
h1->Fit(``gaus'');
```

This will fit a gaussian to the histogram `h1` , by minimizing the sum of the squared errors, weighted by the inverse of the square root of the number of counts in the bin, using the function value at the midpoint of the bin, and using a host of other options. Now, if you don't want to use the basic options, here's what you do. You use the call

```
h1->Fit(``gaus'', ``fit_option'', ``graph_option'',lower_bound,upper_bound);
```

Here are the fit options (input as a quoted character string):

- V – verbose fitting information – USE THIS! How else are you going to know that the fit didn't work at all!
- I – use integral of function over bin, rather than value of function at midpoint – better, but slower.
- W – use constant weights
- E – "perform better error estimation using the Minos technique", whatever that means.
- B – "some parameters are bound" – see below.
- R – fit only over the range of x, y, or z specified in the function.
- + – fit a new function, but don't throw away the old function.
- H – do not draw or store the graphics object resulting from the fit.
- O – store, but do not plot, the graphics object from the fit.

The graph options are the same as for drawing any histogram (see question 5.1).

Rather than fitting from the keyboard, you can fit directly from the GUI. Move the mouse over the pad of the drawn histogram until you see an arrow (not a hand, crosshairs, or anything else), and then right click. Then pick out "FitPanel" to see another GUI with all of the above options on it.

You can also fit your own user defined functions, but if you do so, you have to give `root` the starting parameter values. `root` will guess at the starting parameter values for its own functions.

To fit a function whose parameters fall only in a certain range, you have to tell the function that the parameter is bound, and then tell `Fit` that the function has a bound parameter. You do the latter with the "B" option. You do the former (parameter 0 between ± 1, for example) with the call

```
func->SetParLimits(0,-1,1);
```

Once you've gotten the fit, you can get the parameter values out as follows (assuming that your histogram `h1` was fitted with a `gaus` function):

```
TF1 *func = h1->GetFunction(‘‘gaus’’);
func->GetParameter(0);
func->GetParameter(1);
func->GetParameter(2);
```

You can get the errors of the fit parameters the same way, using `GetParError(0)` , etc. Errors of the fit can be had by finding the histogram value via `h1->GetBin(x);` , and then subtracting the function value at x.

6.3 How can I smooth the data with `root` ?

6.4 What kinds of random deviates does `root` have? How do I generate them?

Normal deviates: generate them with calls to `gRandom->Gaus(0.0,1.0)` , with the first argument being the mean and the second the standard deviation.

Uniform deviates: generate them by `gRandom->Rndm();` .

For either of the above, you can set the seed by the call `gRandom->SetSeed(x);` . The default seed is 12345.

You can get deviates for an arbitrary distribution, but you have to do a couple more steps. First, you have to define the PDF (probability distribution function, which gives the probability of landing between x and $x+dx$) as a function (see question 6.1) with the proper range. Then, you call the `GetRandom()` member function. For example, to generate random deviates from a gamma distribution with $\alpha = 3$ and $\beta = 1$, you'd do the following:

```
TF1 gamma = new TF1(``gamma'', ``0.5 * exp(-x) * x * x'',0,100);
gamma->GetRandom();
```

The one thing you have to be careful about here is the upper and lower limits, due to the way `root` calculates the random deviate. What `root` does is integrate the function between the limits, at a given set of points. After normalizing (i.e., multiplying the function by a value so that the total integral is 1.0), it then generates a uniform deviate, figures out in which bin it falls, and then linearly interpolates. If the limits of your function are too wide, that integration will be very grainy, and you'll be relying too much on the linear interpolation. So, my advice is, always take a look at a 1-D histogram of the random deviates before using it. The graininess will be readily apparent as a "flat spot" on the graph where it doesn't belong.

6.5 How can I get a graph of some input data?

Here's the situation: you want to make a killer graph of your data, suitable for publication, so you decide to use `root` . You've come up with the data points and the errors on each side, and now you need to come up with a way of drawing your error bars.

`root` makes this pretty simple, but there it is a subtle process. The first thing to do is to get the x and y data into separate one dimensional `Float_t` matrices. Also, put the number of data points into a integer variable. Then, create the graph with a statement like

```
TGraph *tt1 = new TGraph(n,x,y);
```

'n' is the integer number of points, 'x' is the name of the matrix containing the x values, and 'y' is the name of the matrix containing the y values.

Drawing the graph is easy: `tt1->Draw(``AL'')` . The stuff inside the quotes gives the style for the graph, with the following options:

- A – draw axes for the graph
- P – use the current marker (see next question) for each point
- * – use a "*" for each point
- C – draw a smooth curve through the points
- L – draw a simple line between the points
- F – fill in the area below the points
- B – draw a bar chart

One subtlety is the drawing of multiple plots on the same graph. Note that there is no way to specify more than one set of data points in `TGraph` , so you've got to be creative. The way to do it is to create and draw the first graph with axes (option 'A'), and then put the create and draw the second graph without axes. (Use the same pad/canvas.) For example,

```
TGraph *tt1 = new TGraph(n,x1,y1);
tt1->Draw(``ALP'');
TGraph *tt2 = new TGraph(n,x2,y2);
tt2->Draw(``L*'');
```

6.6 How do I set graphics options for a graph?

The overall title and most of the color/style options are easy. Use the same commands as for histograms, before drawing the graph. For example, to set the title and background color, do something like

```
// Create the canvas, set options
c1 = new TCanvas("c1","",200,10,700,500);
c1->SetGridx();
c1->SetGridy();

// Create the graphs
TGraph *tt1 = new TGraph(nlines,x1,y1);

tt1->SetTitle("");
tt1->SetMarkerStyle(24);
tt1->Draw("ALP");
```

Setting the axis titles is a little tricky, since `root` piggybacks TGraph onto the histogram classes. The basic method is to draw the graph, then set the axes titles:

```
// Create the canvas, set options
c1 = new TCanvas("c1","",200,10,700,500);

// Create the graph
TGraph *tt1 = new TGraph(nlines,x1,y1);

tt1->SetTitle("");
tt1->Draw("AL");
c1->Update();
tt2->GetHistogram()->SetXTitle("Energy Lost (GeV)");
```

```
tt2->GetHistogram()->SetYTitle("Relative Frequency");
c1->Update();
```

Note that you have to do the `c1->Update` first to actually draw the graph and create the histogram. The second update is to redraw the canvas to show the titles.

6.7 How do I get error bars on a graph?

By using the class `TGraphErrors` , naturally. This can only do symmetric error bars, but that's a start. This works very similarly to `TGraph` , but with just a slightly different input. Use

```
TGraphErrors *tt1 = new TGraphErrors(n,x,y,dx,dy);
tt1->Draw(``ALP'');
```

'n', 'x', and 'y' are as in TGraph. 'dx' and 'dy' are one dimensional matrices of the errors in x and y. Perhaps soon there will be an extension to asymmetric error bars. It shouldn't be that hard, but they might just be a little busy.

6.8 How do I get a legend on a graph, showing what each line on a plot is?

Got me. I'm searching. Any help would be appreciated.

7 Input & Output Questions

7.1 How do I read in a .root file? How do I find out what's in a .root file?

This is easy, the way the gods of computing really wanted it to be. It's just a one liner to open up the file:

```
TFile *file = new TFile(``file.root'');
```

Finding out what's in the file is pretty easy too:

```
file->ls();
```

Now that you've read in the file, you can start using the objects in the file straight away, without having to declare them or anything silly like that, since the interpreter automatically looks for them in reasonable places:

```
sqroot->Print();
histo1->Draw();
```

If you're writing code to compile, you've got to do it the long way:

```
TH1 *histo1 = (TH1*) gROOT.FindObject(``histo1'');
histo1->Draw();
```

7.2 How do I create a .root file?

The first step is to create the file.

```
TFile myfile(``myfile.root'', ``RECREATE'');
```

The second step is to put stuff into the file. Say you've got an ntuple `ntuple` and a histogram `h1` you want to write to the file. There are two ways to do this. If you want some, but not all of the objects laying around to be saved, you can save them explicitly:

```
ntuple->Write();
h1->Write();
```

If you want to save all of your objects to the file, try this:

```
myfile->Write();
```

Finally, the file has to be closed. This will happen automatically at the end of the run, but if you want to do it explicitly, you can type `myfile->Close();`

7.3 How do I convert an .hbook file into a .root file? Will all the data be preserved, including the names?

Since the developers of `root` knew that physicists love legacy code, there is included in each distribution a program to convert old "`.hbook`" files created with PAW into new "`.root`" files usable by `root` . To use it, just type at the system command line

```
h2root file-name.hbook file-name.root
```

If you're lazy, you can omit the second file name, and `h2root` will use the same file name, except with .root instead of .hbook

As for using the new file, just load it up into `root` as you usually would. Any histograms have the same name, and can be used just like a normal `root` histogram. Any ntuples have been turned into a `TTree` object, and can be treated like a `root` tree. Since PAW does not recognize the difference between uppercase and lowercase in names, `root` has a choice about capitalization. The choice made was to capitalize the initial letter, and make the rest lower case.

7.4 How do I read data from an ASCII file into an ntuple?

The basic method is to steal a macro from the `root` Tutorial [2], and modify it for your purposes. For those of you who can't find the tutorial, here is my mangling of the macro:

```
{
// Reset -- this appears to be done with every macro -- might as well do it
gROOT->Reset();
// Open a file, just like on c -- no need for stream output here, yeah!
FILE *fp = fopen("/home/tebe/cern/ascii1.txt","r");
// Declare some variables -- use their types, not the usual int, float, etc.
Float_t x,y,z;
Int_t ncols,nlines=0;
// Here comes the C++ stuff
// Allocate the file, giving the name and an option
// I'm not sure what "RECREATE" means
TFile *f = new TFile("basic.root","RECREATE");
// This creates an ntuple
TNtuple *ntuple = new TNtuple("ntuple","data from ascii file","x:y:z:");
// Now loop through the file using typical C syntax
while (1)
{
  ncols = fscanf(fp,"%f %f %f",&x, &y, &z);
  if (ncols < 0) break;
  if (nlines < 5) printf("x=%g, y=%g, z=%g\n",x,y,z); // print first 5 lines
  // Put the data into the ntuple
  ntuple->Fill(x,y,z);
  ++nlines;
}
printf("found %d points\n",nlines);
// Close the ascii file
fclose(fp);
// Write the root file, containing the ntuple
f->Write();
}
```

7.5 How do I take data from an ntuple and write it to an ASCII file?

See the answer to question 4.5 on how to traverse an ntuple. Once you've gotten everything set up to traverse the tree, just put in the usual C statements to open, write to, and close a file. Here's an example of how to do that, working with the same ntuple as question 4.5:

```
    FILE *out = fopen(``ascii.txt'', ``w'');
    if (out == NULL)
       printf(``Sorry, but the ascii output file did not open.'');

    for (Int_t i=0; i<nentries;i++)
       {
          nbytes += ntuple->GetEvent(i);
          fprintf(``x,y,z= %f, %f, %f \n'',x,y,z);
       }

    fclose(out);
```

7.6 How do I get data from a fitted function and write it to an ASCII file?

This is pretty simple. if you want the values, the best way to do it is to write a loop that goes through the x values you want, and for each one, calculate the value of the function using Eval. As an example, say you've got a histogram called `h1` that you want to fit and get the results out to another program. Then this would fit a degree 6 polynomial to it and write the data:

```
h1->Fit(``pol6'', ``V'');
TF1 func = h1->GetFunction(``pol6'');
Double_t h_min = h1->GetMinimum();
Double_t h_max = h1->GetMaximum();
Double_t h_step =  (h_max+h_min)/100.0;
for(Double x = h\_min; x <= h_max; x += h_step)
fprintf(out, ``%f %f \n'',x,func->Eval(x));
```

If you want to output the paramters of the function, see question 6.2 for how to get the parameters, and then use `fprintf` to get them put out.

7.7 How can I get `root` to output a histogram directly to an encapsulated postscript file?

By using the `Print()` member function, with an argument of the name of the file you want to create. For example,

```
c1->Print(''test.eps'');
```

will create an encapsulated postscript file named `test.eps` .

Note that you can also create the files manually by clicking on the canvas.

7.8 How can I get `root` to read a file and histogram it at the same time another process is writing the file?

This really isn't that hard. Just set up `root` as if you were reading an ASCII file as shown above, but with a little more circumspection on the input. Read the input a character at a time, gradually building up an input string. When the string is complete, process the observation. If you get to an end of file (EOF) condition, make `root` wait and read the file again.

There are two keys. First, make sure the data is coming in. If it's coming from another program, make sure that the entire output isn't being buffered. Second, make sure you have some way of stopping execution of `root` , either by a time out, another input file that passes commands, or a count of the input.

If anybody is interested in this, I've got a program I wrote to do this, as a real time monitor of some DAQ. Write me, and I'll send it to you.

References

[1] Brun, René, et al., *ROOT, Overview*, published as the gzipped postscript file ROOTMain.ps.gz, which may be downloaded freely from the main root web site, http://root.cern.ch, under "status and availability".

[2] Brun, René, et al., *ROOT, Tutorial*, published as the gzipped postscript file ROOTTutorials.ps.gz, which may be downloaded from the main root web site, http://root.cern.ch, under "status and availability".

[3] Brun, René, et al., *ROOT, Classes Reference Manual*, published as the gzipped postscript file ROOTClasses.ps.gz, which may be downloaded from the main root web site, http://root.cern.ch, under "status and availability".

[4] Rademakers, Fons, *ROOT Course*, published in three gzipped postscript files course1.ps.gz, course2.ps.gz, and course3.ps.gz, which may be downloaded freely from the main root web site, http://root.cern.ch, under "publication list".

[5] Levine, John R. and Margaret Levine Young, *UNIX for Dummies*, second edition, IDG Books Worldwide, 1995. *I am not implying that my readers are dummies!* I learned everything I know about UNIX from this book, and found it quite useful.

[6] Oualline, Steve, *Practical C Programming*, third edition, 1997, published by O'Reilly & Associates.

[7] Kochan, Stephen G., *Programming in ANSI C*, first edition, 1988, Hayden Books.

A second edition exists, published ?. One big advantage of this over the Oualline book is that it has an appendix on the standard C library.

[8] Pohl, Ira, *C++ For C Programmers*, second edition, 1994, published by Addison-Wesley.