HEP Computing
Part III
ROOT
Marcella Bona

Lectures 5-6

Queen Mary
University of London

m.bona@qmul.ac.uk

Aims of this part of the course

The aim of this section is to give you a crash course in using ROOT. By the time you've worked through this you should be able to:

- set up the ROOT environment on your own machine
- *start root and run a simple macro*
- *know how to use histograms, ntuples, files etc …*
- *know where to go for more information*
- *fit to histogram data*
- *compile a stand alone root application*
- Write a script to process a macro on several root files – i.e. learn how to automate *'chores'*
- Learn about more root based tools – functions, binned fits in root and automatic code generation

This set of lectures has been tested for ROOT >= 5.27

Lecture 5

- Getting started with using ROOT.
  - ntuples
  - histograms
  - macros
- TFiles → saving data
- Using histograms and trees
- simple ROOT macros

ROOT

- What is ROOT?

  C++ based, code on the web, actively developed by many people, don't need to learn another syntax to use it (root macros are C++)

  ➔ flexibility brings complexity ☹

  ➔ manual is large (over 300 pages)

  ➔ some good web based courses available as well

- Useful resources:
  - User Guide etc:

    http://root.cern.ch/ ← download root from here
  - HOW-TOs, Tutorials and class structure on web ☺

    (some tutorials listed in the references at end)

Some basic concepts

- Histograms
  - Plots of data as a function of 1, 2 or 3 variables
- NTuples
  - A more complicated data format
    - store information on an event or candidate basis
    - can cut on other variables in NTuple to do analysis on the fly
    - based on a tree-like data structure
- Files
  - The persistent data type
  - Persistent objects inherit from TObject
  - Can persist user defined objects if they inherit from TObject
- macros
  - Source file containing command to execute in the interpreter
- GUIs
  - Don't always need to know how to do things on the command line!

Histograms

1, 2 and 3D binned plots of the distribution of variables – good for visualising what analysis cuts do to data/complicated functions:

types

```
TH1F        TH1D
TH2F        TH2D
TH3F        TH3D
```

→ the F/D refers to the data type used either `Float_t` or `Double_t`

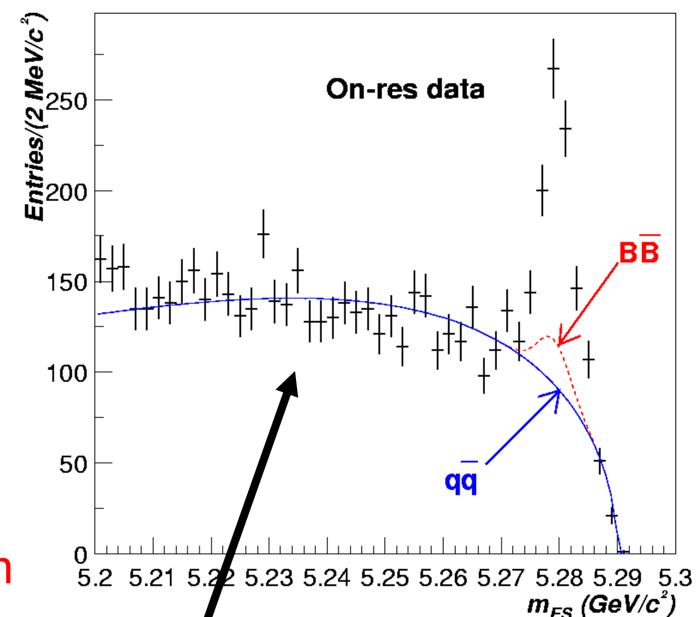
→ If you have a histogram called `myHist` and want to see what it looks like you `Draw()` it

root[10] `myHist.Draw()`

the root prompt

The histogram object with variable name myHist

Draw() member function is called to show the histogram



histogram shown as data points with background curves added on top of the histogram

Flat Files

'Flat File' is jargon for a formatted text files containing columns of numbers corresponding to variables.
e.g. a 3 column flat file would look something like:

```
5.285 0.02  5.43
5.273 -0.12 4.32
…
```

1 row corresponds to a candidate or an event

a column corresponds to a variable

where you have the advantage of being able to read the numbers by eye as well as using them in code/scripts etc. (most people can't read binary too well…)

You may see flat files knocking around from time to time

NTuples

- data structure on an entry by entry basis (e.g. candidate or event)
  TTree/TChain – the same kind of thing – both are NTuples

  You can
  - loop over the events one by one to analyse data
  - draw variables or combinations of variables
  - cut on variables as you draw them
  - fill histograms of anything that you can draw

- NTuples are a lot more flexible than histograms as you can optimize your analysis once you've made the ntuples → you don't have to do this before making them
  ➔ *i.e. you make a few root files by running on all of the experimental data you need, using loose cuts and then work on this subset of data/variables.*

Starting with ROOT

The first part of the ROOT tutorial uses Monte Carlo data from a BaBar analysis to introduce the basics of using histograms, files and ntuples in root. The examples lead to you developing a scaled down version of what you would do in a cut and count analysis.

This tutorial concentrates on using ROOT with LINUX. There are pointers to differences between LINUX and mac, <u>but the use of Windows is beyond the scope of these Lectures</u>.

What is ROOT?

ROOT is a data analysis toolkit that has one main application

root the main program that you run

- You need to append `$ROOTSYS/bin` to your `PATH` in order for your shell to know where to find the command `root` (see the next page).
- You also need to modify `LD_LIBRARY_PATH` so that the shell can find the shared libraries it needs at run time (see the next page) in case you decide to start compiling your ROOT analysis code at a later date.
- ROOT (sort of) uses C++ syntax:
    - If you compile your code you need to be precise with C++ syntax.
    - If you interpret your code (using CINT, it is CLING in root6), then you don't have to be as precise.
    - For the longer term, you should seriously consider compiling your code, however CINT is great for trying things out and learning!
    - There are a few limitations with CINT that you may encounter (e.g. templates)

N.B. mac users need to set the DYLD_LIBRARY_PATH

e.g. Setting up the ROOT environment

You (or your sys-admin) needs to have installed a version of root and to set the following environment variables:

- ROOTVER – the version number (not strictly necessary)
- ROOTSYS – The ROOT installation directory
- LD_LIBRARY_PATH – where the system looks for libraries
- you also need to append your path with the ROOT bin directory

If you use bash add the following to your .bash_profile.

```
export ROOTVER=5.27.00
# path to root install directory. This will depend on your sysadmin
export ROOTSYS=/users/bona/root/$ROOTVER
export PATH=$PATH:$ROOTSYS/bin:$MYPATHVAR
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
# on Mac OS X you'll want to comment out the previous line and
# uncomment the following.
#export DYLD_LIBRARY_PATH=$ROOTSYS/lib:$DYLD_LIBRARY_PATH
```

- run $ROOTSYS/bin/thisroot.sh to set-up a specific root version
- llog into a new terminal to see that your shell now knows about root.

Running ROOT

- **root** *start a root session*

  -l suppress the 'splash screen'

  The splash screen is the window that pops up for a few seconds when you start root. By suppressing this you start root a little faster.

  -b run in batch mode [no graphics displayed]

  This will speed things up a lot (especially if you are working from a remote machine).

  -q quit root when macro finished

  `root -l -b -q myMacro.cc("arguments")`

- Can open a ROOT file when starting a session:

  `root myfile.root`

m.bona@qmul.ac.uk

A few words on CINT/CLING

- Based on C++
- Is a C++ interpreter
- Can do things wrong sometimes (solution is to compile code)
    - you won't get warnings when it does ☹
    - prime example is if you forget '`;`' at the end of a line in a macro the whole line is just ignored!
- Some people say that 'ROOT needs to be restarted all the time'
    - this is probably true if your code is not bug free!
    - if you are bug free then it is not a problem…
- There are differences between CINT and C++ some are:
    - Sloppy use of "`->`" and "`.`"
        - *these can be replaced with each other, however one gets warnings in later versions of root if using the wrong syntax*
    - The "`;`" at the end of lines can be omitted in interactive use (not when running with macros!)
    - Can tab complete on an object in `cint` to see what it can do!

CINT/CLING commands

- CINT/CLING commands always start with a dot "`.`", e.g:

`.q`	*quit out of ROOT session*
`.! <shellcommand>`	*execute a shell command, e.g.*

```
.! ls
.! emacs myMacro.cc &
```

`.?`	*help; get list of CINT commands*

ROOT Data Types

- Similar to C++:
  - Basic types: first letter is capitalised and have suffix "_t":
    int → Int_t   float → Float_t   double → Double_t
  - Names of root classes start with "T" e.g.
    TDirectory, TFile, TTree, TH1F, TGraph, …
- Some ROOT types (classes):
  - TH1F - 1D Histogram filled using floating precision data
  - TH1D - 1D Histogram filled using double precision data
  - TFile – a file containing persistent data
  - TDirectory – a directory (useful to keep a TFile tidy/organised)
  - TTree – can store per-event info in branches and leaves
  - TF1 – 1-dimensional function, TF2, …
  - TString – a ROOT string object (better than a C/C++ string)
  - TObjString – a persistable root string

Why care about the difference between Float_t and float?

int	→	Int_t
float	→	Float_t
double	→	Double_t

- The ROOT data types are used in order to make user code and ROOT code more platform independent.

-You probably don't care or need to worry about the details of this

- However, in general you should try and use the ROOT defined types where possible

Tab Completion

Tab-completion of commands and filename calls can help in finding available commands, e.g.

```
TH1F h1("h1", "title", 50, 0.0, 10.0);
```

→ *define a histogram with 50 bins and an x axis range of 0.0-10.0*

```
h1.[tab]
```

→ *lists all available functions of a TH1F object*

```
TH1::[tab]
```

→ *list all available functions of a TH1 object*

```
TH1::SetName([tab]
```

→ *show the available function prototypes e.g.*

```
root [0] TH1::SetName([tab]
void SetName(const char* name)  // *MENU*
```

so the syntax to change the name of this histogram is just:

```
h1->SetName("myNewName")
```

Queen Mary University of London

Starting and exiting root

```
> root
root[0] .q
>
```

start up root by typing this at the shell prompt

quit root (remember this is CINT you are dealing with).

```
> root -l
root[0] TFile f1("somefile.root");
.
.
.
root[n] .q
>
```

start up root by typing this at the shell prompt & suppress the splash screen

do something

ROOT exercise 1: making sure you can use root

- log onto a machine with root installed on it
- download and untar the examples for part 3:

  `these are unpacked in ./Lectures/macros/`
- and `cd` into this dir.

- set up your root environment
- start a root session:
  - root –l

e.g. copy the lines like those below from the example on page 11.

```
export ROOTVER=5.24.00
# path to root install directory. This will depend on your sysadmin
export ROOTSYS=/Users/bevan/root/$ROOTVER

export PATH=$PATH:$ROOTSYS/bin:$MYPATHVAR
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH

# on Mac OS X you'll want to comment out the previous line and
# uncomment the following.
#export DYLD_LIBRARY_PATH=$ROOTSYS/lib:$DYLD_LIBRARY_PATH
```

ROOTVER → set as the root version installed
ROOTSYS → this is the full path to the install directory for root
LD_LIBRARY_PATH → like PATH, but for compiled libraries

- now you can play ….

```
root -l -b -q hello.cc'("Yourname")'
```

Files

persistence = save data (histogram, ntuple, object) in a file

A root file is known as a `TFile`. That is the class name for the object. From your root prompt you can open the `TFile` data/signal.root using (from Lecures/macros)

```
root[0] TFile f1("data/signal.root")
```

The content of the file can be seen by using the `ls()` member function:

```
root [1] f1.ls()
TFile**             signal.root
 TFile*             signal.root
  KEY: TH1D         cossphericity;1 cossphericity
  KEY: TH1D         photonlat;1     photonlat
  KEY: TH1D         pi0mass;1       pi0mass
  .
  .
 KEY: TTree      selectedtree;1  Final variables tree
```

Files

persistence = save data (histogram, ntuple, object) in a file

A root file is known as a `TFile`. That is the class name for the object. From your root prompt you can open the `TFile` data/signal.root using (from Lecures/macros)

```
root[0] TFile f1("data/signal.root")
```

The content of the file can be seen by using the `ls()` member function:

```
root [1] f1.ls()
TFile**          signal.root
 TFile*          signal.root
  KEY: TH1D      cossphericity;1 cossphericity
  KEY: TH1D      photonlat;1     photonlat
  KEY: TH1D      pi0mass;1       pi0mass
  .
  .
 KEY: TTree     selectedtree;1  Final variables tree
```

a 1D histogram

object version number

comment

key object type

key name

a TTree object (an NTuple)

you can also **Print()** and **Dump()** information about the content of a file

How do you get access to the persistent objects in a file? There are two ways:

```
root [4] selectedtree
(const class TTree*const)0x852ff08
```

loads tree into memory according to the key e.g. `selectedtree`

```
root [5] TTree * mySignalTree = (TTree*)f1.Get("selectedtree")
```

object type

pointer (you need the '*' prefix)

cast the returned pointer as the type you want (assumes you know it) *The default is a TObject **

use the key name to get the object

The second way is better as it will ALWAYS work for multiple open files – you keep track of the pointers yourself and can do anything you want with them! If you are only using a single file then you can use the first way to access the stored objects when working interactively.

ROOT exercise 2

1) Open the files found in the Lectures tarball:
Lectures/macros/data/signal.root
Lectures/macros/data/continuum.root
and look at the content of the file (use **`ls()`** member function of **`TFile`**).

2) get pointers to the **`TTrees`** in each file – **`Print()`** the content of one of them [they are the same structure – so there is no point in looking at both of them ☺]

3) draw some of the variables:
hint – you can cut on variables when you draw them

```
mySignalTree->Draw("aVar");
mySignalTree->Draw("aVar", "aCut", "same")
```

e.g.

```
mySignalTree->Draw("mes")
mySignalTree->Draw("mes", "abs(de)<0.2", "same")
```

- Do the same for a few histograms e.g.

```
root [4] pi0mass
(const class TH1D*)0x87a3df8
root [5] pi0mass->Draw()
```

← can't cut histograms

Now you can

- set yourself up to use a given root version
- open a file in root
- access its content
- draw from TTrees and histograms (same works for TGraphs etc)

Q) Don’t like the grey background on the plots?
A) `root [4] gROOT->SetStyle("Plain")`
will solve that problem for you.

→ should be default in newest versions

- The next part of the course is to write a macro that loops over the events in a TTree and makes some cuts – filling histograms. These histograms are then written out to a new file. Then you can compile the code stand-alone and see it run faster.

- For now however I’ll go into more detail on histograms and TTrees as we build towards this goal.

Histograms

Declare with:

```
TH1F h1(arguments …)
```

- Make your first 1D histogram:

```
TH1F h1("h_name", "h_title", 10, 0.0, 10.0);
```

n_{bins} → 10, x_{min} → 0.0, x_{max} → 10.0

`h_name` = key name of histo

`h_title` = name which appears on plotted histogram

- Now draw the (currently empty) histo:

```
h1.Draw();
```

- Fill with a few entries:

```
h1.Fill(1.);
h1.Fill(3,10.7);
```

the number to fill the histogram with *(default value is 1.0)* → 10.7

x value to fill histogram at → 3

- Try drawing the histogram when you have a few entries

```
h1.Draw();    //do this occasionally to update the histogram
```

Some useful commands to play with now that you've got a histogram

```
h1.SetFillColor(Color_t color = 1)
h1.SetFillStyle(Style_t styl = 0)
h1.SetLineColor(Color_t color = 1)
h1.SetLineStyle(Style_t styl = 0)
h1.SetLineWidth(Width_t width = 1)
```

`h1.SetFillColor(Color_t color = 1)`	Change the fill colour.
`h1.SetFillStyle(Style_t styl = 0)`	Change the fill style.
`h1.SetLineColor(Color_t color = 1)`	Change the line colour.
`h1.SetLineStyle(Style_t styl = 0)`	Change the line style.
`h1.SetLineWidth(Width_t width = 1)`	Change the line width.

Line colours and styles are described in the 'Graphical Objects Attributes' section of the ROOT user guide.


```
kRed
kOrange
kYellow
kSpring
kGreen
kTeal
kCyan
kAzure
kBlue
kViolet
kMagenta
kBlack
kPink
```

Make sure you use colours wisely! There is nothing more annoying than seeing a talk projected onto a screen with half a dozen invisible lines!

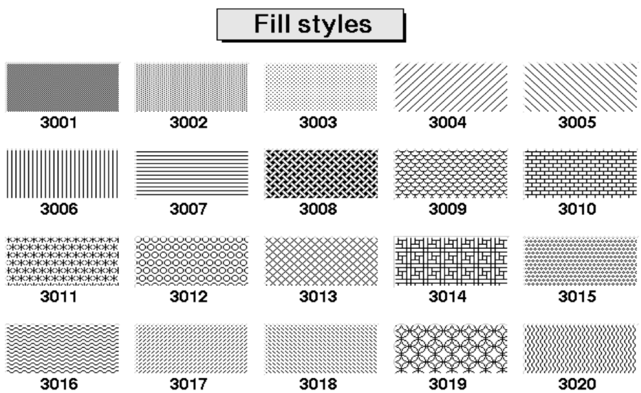
Try and stick to 'safe' colors like blue, red and black.

Can define new colours using the TColor class.

Some useful commands to play with now that you've got a histogram

`h1.SetFillColor(Color_t color = 1)`	Change the fill colour.
`h1.SetFillStype(Style_t styl = 0)`	Change the fill style.
`h1.SetLineColor(Color_t color = 1)`	Change the line colour.
`h1.SetLineStyle(Style_t styl = 0)`	Change the line style.
`h1.SetLineWidth(Width_t width = 1)`	Change the line width.

Line colours and styles are described in the 'Graphical Objects Attributes' section of the ROOT user guide.



Available fill styles shown left

Remember to give axis labels a sensible title:

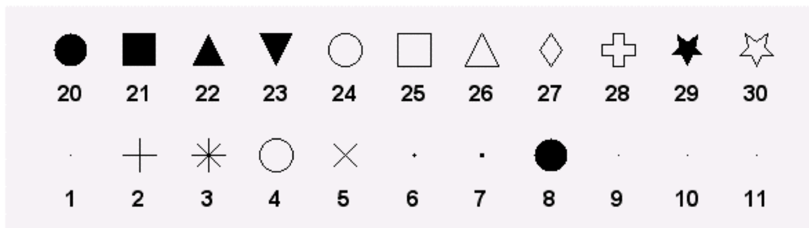
```
h1.SetXTitle("This is the x-axis")
h1.SetYTitle("This is this y-axis")
```

always label your axes!

The default line style is **kSolid**. There are times when you will want to change this to another value (either by integer or enum):

```
kDashed        - - - -
kDotted        ............
kDashedDotted  ·-·-·
```

Sometimes it can be useful to mark points on a histogram using a TMarker. There are various marker styles:



Which can be used as follows:

```
TMarker myMark(xCoord, yCoord, iStyle)
myMark.SetMarkerColor(kRed)
myMark.Draw()
```

Same graphical attributes modifiers as a histogram or line.

where xCoord and yCoord are the coordinates to plot the marker at (in terms of the histogram or graph), and iStyle is one of the marker styles above.

It is also possible to change the range of the x-axis that you want to plot a histogram for using

```
h1.SetAxisRange(xMin, xMax)
```

where xMin and xMax should be within the range defined in the constructor.

Why don't I see the changes I made to a histogram?

If you modify the settings of a histogram (or marker), you will need to redraw the object in order for it to be updated on the **TCanvas**.

Overlaying more than one histogram on a plot

More than one histogram can be drawn on top of each other using `h1.Draw("same")`. This only makes sense if the axes have matching ranges.

Errors on a histogram:

Bin entries on a histogram are an accumulation of events occurring with a probability according to a Poisson distribution.

If you use `h1.Draw("e")`, ROOT will draw error bars for you.

2D Histogram

```
TH2F  h2("h_name", "h_title", 10, 0.0, 10.0, 20, -10.0, 20.0);
```

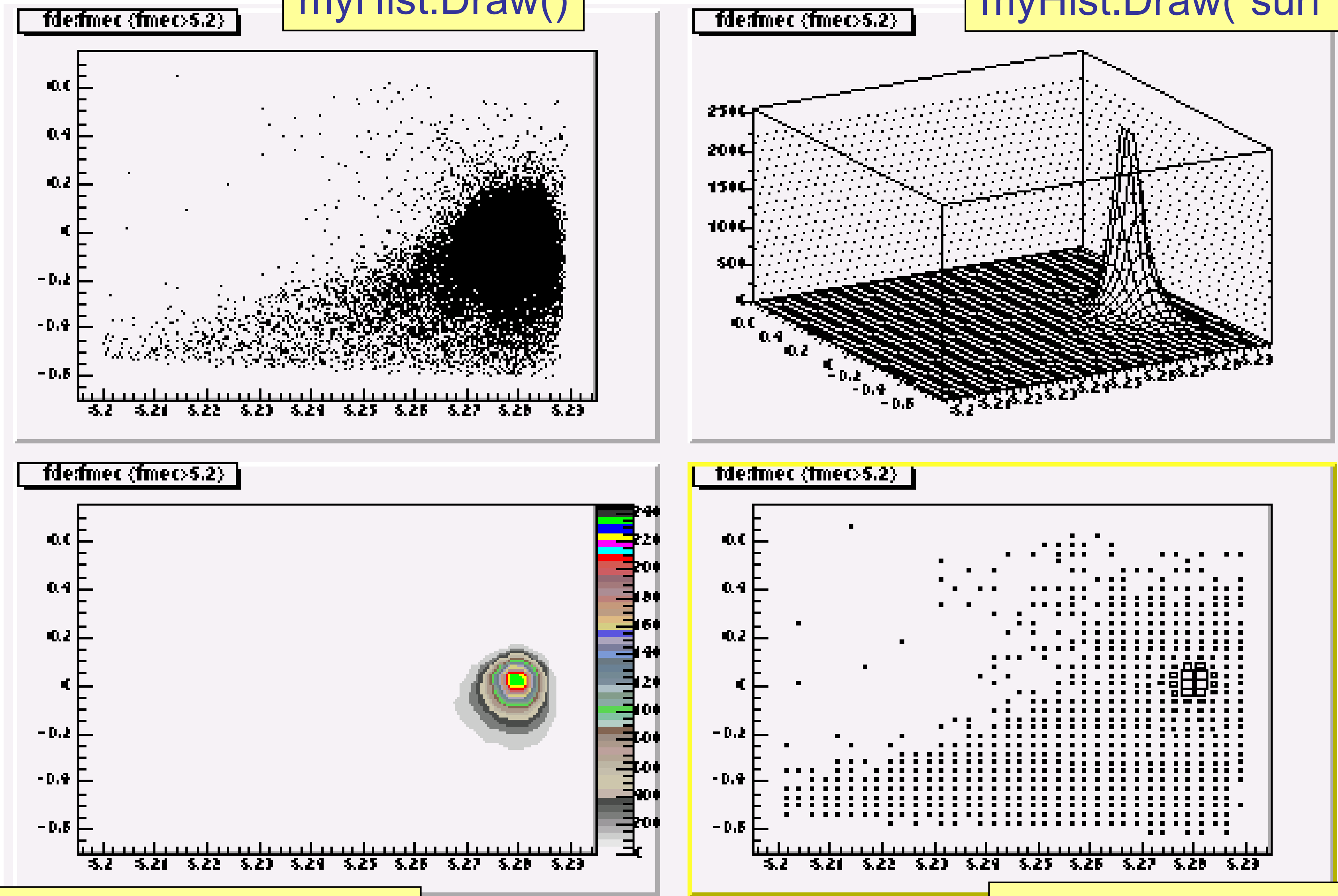
x axis co-ordinates (10, 0.0, 10.0) y axis co-ordinates (20, -10.0, 20.0)

- 2D histograms behave the same as 1D histograms
- have some interesting Draw() options

surf	*- draw a surface*
surf1	*- draw a surface with colour contors*
cont	*- draw a contour plot*
contz0	*- draw a contour plot with the y axis scale shown*
lego	*- draw a 2D histogram*
box	*- draw boxes (default is to spread points out according to the defined bins)*
text	*- draw 2D grid of number of entries per bin.*

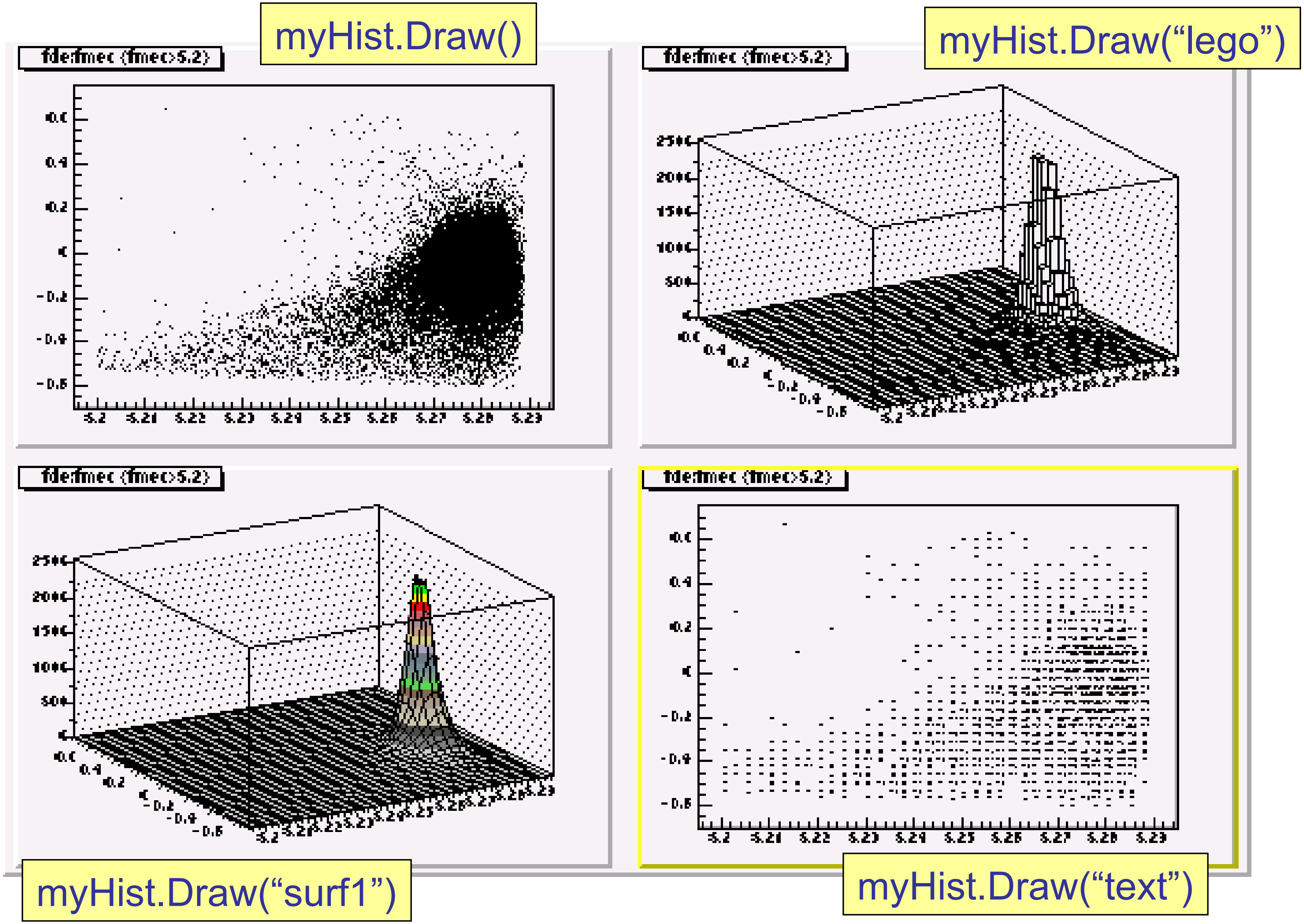
- These draw options also work for trees myHist.Draw()

myHist.Draw(“surf”)



myHist.Draw(“contz0”)

myHist.Draw(“box”)


myHist.Draw()
myHist.Draw("lego")
myHist.Draw("surf1")
myHist.Draw("text")

Making histograms from a TTree

- When you draw a variable from a TTree you can fill a histogram

variable name

means make hist

hist name

```
myTree.Draw("mes>>tmphist");
tmphist.Draw("e");
```

If you have already defined the histogram `tmphist`, then ROOT will fill this for you from the tree. If you have not defined `tmphist` ROOT will make a guess as to the axis ranges, and will create a 100 bin histogram for you.

Now root knows you have a histogram of name `tmphist`

`tmphist` is a histogram made to have the content corresponding to that of the tree

Making histograms from a Ttree (II)

```
myTree.Draw("mes>>tmphist");
myTree1.Draw("mes>>+tmphist");
myTree2.Draw("mes>>+tmphist");

tmphist.Draw("e");
```

`>>+` means add to existing histogram

By default you get a histogram with 100 bins. If you want to change this you'll have to specify a histogram yourself; e.g.:

```
TH1F tmphist("tmphist", "", 25, 5.2, 5.3);
myTree.Draw("mes>>tmphist");
```

Macros

- Lots of commands you'll want to repeat often just like scripts in terms of shell programming or source files in terms of programming.
  - save them in a "macro" file (same concept as a PAW kumac)
  - just a bunch of commands in file, enclosed in {…}
- The following is an example of an **un-named macro:**

```
{
  TFile f("data/signal.root");
  f.ls();
  TCanvas c1;
  pi0mass.Draw();
  c1.Print("pi0mass.eps");
}
```

- You save macros as a C file; e.g. `myMacro.cc` (actually the extension that you use can be anything).
- To execute an un-named macro:

```
root[0] .x myMacro.cc
```

On doing this ROOT will run all the commands in `myMacro.cc`.

- The following is an example of a **named macro**:

```
void myMacro(void)
{
  cout << "Hello World!" << endl;
}
```

contains normal C++ code, functions/classes etc.

- If the macro name is the same as a function then you can run the macro from the ROOT prompt with

```
root[0] .x myMacro.cc
```

or from the command line with

```
> root -l -b -q myMacro.cc
```

- named macros like this are `#include`able in other files.

m.bona@qmul.ac.uk

included file

macro entry point

the function call

e.g. save this as mainFunc.cc

```
#include "myMacro.cc"

void mainFunc(void)
{
   cout << "calling included function" << endl;
   myMacro();
   cout << "done" <<endl;
}
```

You can pass an argument to a named macro from the command line or ROOT.

Try running the following example:

```
root hello.cc'("Your name")'
```

- Combine named and un-named macros to build up an analysis.
- Macros can call and use other macros.
- Syntax to load a macro from a file:

  ```
  gROOT->LoadMacro("myFile.cc");
  ```

  formal version of the CINT command line `.L myFile.cc`
- If you will use the function frequently, better to have named macro or define the function in a header file you can `#include` from your macros.
- Scope works the same as in C++, anything defined in a macro or function exists only inside that macro or function.
- Complicated analyses should be compiled using gcc or another C++ compiler (to help you debug it and speed up the analysis).

Lecture 6

- More on TFiles – making a new file
- Reading data from a tree on an entry by entry basis
- Makefiles
- The 'main()' function
- Compiling a stand-alone executable
- Using scripts to run a ROOT analysis

TFiles

You've already met `TFiles` – a bit more on how to use them

- Files can contain directories, histograms and trees (ntuples) etc.
- These are 'persistent' objects
- In root you make an object persistent by inheriting from `TObject`

A few file commands/constructors that you've already met:

- Open an existing file (read only)

```
TFile myfile("myfile.root");
```

- Open a file to replace it

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

  or append to it:

```
TFile myfile("myfile.root", "UPDATE");
```

- Some useful member functions include

```
TFile::GetName();
TFile::GetTitle();
TObject * TFile::Get(const char *)
```

Tfiles (II)

- Open an existing file (read only)

```
TFile myfile("myfile.root");
```

- Open a file to replace it

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

or append to it:

```
TFile myfile("myfile.root", "UPDATE");
```

- Some useful member functions include

```
TFile::GetName();
TFile::GetTitle();
TObject * TFile::Get(const char *)
```

the object key name

`TObject *` - you have to "cast up" the returned object to the persistent type to be able to use it properly.
This is just what you did earlier with:

```
TTree * mySignalTree =
        (TTree*)myfile.Get("selectedtree")
```

Using TFile::Get()

```
root[0] TFile myfile("data/signal.root")
root [1] myfile.ls()
TFile**           signal.root
 TFile*           signal.root
  KEY: TH1D       cossphericity;1 cossphericity
  KEY: TH1D       photonlat;1     photonlat
  KEY: TH1D       pi0mass;1       pi0mass
  .
  .
  .
 KEY: TTree     selectedtree;1  Final variables tree
```

Open the file signal.root (This is $B^0 \to \pi^0 \pi^0$ Monte Carlo simulated data)

The key type is the root object type ☺

```
root [2] TH1D * cossph = (TH1D*)signal.Get("cossphericity");
root [3] TH1D * lat = (TH1D*)signal.Get("photonlat");
root [4] TH1D * mpi0 = (TH1D*)signal.Get("pi0mass");
```

"Get" the 3 histos in memory

```
root[5] mpi0->Draw();
root[6] lat->Draw();
root[7] cossph->Draw();
```

Try looking at the histograms

What if you want to make a new file?

```
TFile newfile("myNewFile.root", "RECREATE", "comment");
```

open a new file

any new objects are automatically put in this file (you can change this behaviour if you don't want it to happen)

```
//make some histograms
TH1F aHist("aHist", "some variable", 10, 0.0, 10.0);
TH2D a2DHist("a2DHist", "x vs y", 10, 0.0, 1.0, 100, -4.0, 4.0);

// make a new tree containing two scalar variables and an array
Float_t x,y;
Int_t n[10];
TTree mytree("mytree", "title");
TBranch * b_x = mytree.Branch("x", &x, "x/F");
TBranch * b_y = mytree.Branch("y", &y, "y/F");
TBranch * b_z = mytree.Branch("n", n, "n[10]/I");

// do stuff
newfile.Write();
newfile.Close();
```

you have to `Write()` a file to save what you have done
It will get closed when it goes out of scope (or is deleted).

Alternatively…

```
//make some histograms
TH1F aHist("aHist", "some variable", 10, 0.0, 10.0);
TH2D a2DHist("a2DHist", "x vs y", 10, 0.0, 1.0, 100, -4.0, 4.0);

// make a new tree containing two scalar variables and an array
Float_t x,y;
Int_t n[10];
TTree mytree("tree", "title");
TBranch * b_x = mytree.Branch("x", &x, "x/F");
TBranch * b_y = mytree.Branch("y", &y, "y/F");
TBranch * b_z = mytree.Branch("n", n, "n[10]/F");

// do stuff (e.g. your selection code)

// persist all objects to a file at the end of the macro
TFile newfile("myNewFile.root", "RECREATE", "comment");
aHist.Write();
a2DHist.Write();
mytree.Write();
newfile.Write();
newfile.Close();
```

you can also `Write()` objects to the file to save what you have done at the end of the macro, just before things go out of scope

Trees

- ROOT trees (`TTree`)
  - Trees can contain different types of data (e.g. Int_t, Bool_t, Float_t, Double_t). The trees have branches (subdirectories).
  - Trees also have leaves that represent variables and contain data.
  - Trees are optimized to enable fast access to data, and minimize disk space usage.
- Trees (with leaves but not branches) can be thought of like tables:
  - rows can represent individual events
  - columns (leaves) represent different event quantities
- Some useful function calls for a TTree:
  - To view the content (variables) in a tree: `myTree->Print()`
  - To inspect event iEvt (print out values of leaves): `myTree->Show(iEvt)`
  - To draw a distribution of a leaf `myTree->Draw("variable")`
  - To draw a 2D distribution of x vs. y `myTree->Draw("x:y")`
  - To draw x while cutting on y `myTree->Draw("x", "y>5")`

Reading data from a tree

```
TTree * mytree = (TTree*)myfile.Get("selectedtree");

Float_t mes, de, fisher, imass[3];

// set the tree up to fill local variables
mytree->SetBranchAddress("mes", &mes);
mytree->SetBranchAddress("de", &de);
mytree->SetBranchAddress("newfish", &fisher);
mytree->SetBranchAddress("imass", imass);

// loop over the candidates in the TTree
for(int iEvt = 0; iEvt < mytree->GetEntries(); iEvt++)
{
  mytree->GetEntry(iEvt);    // load the candidate #iEvt
  cout << "candidate iEvt = " << iEvt << "\t mes = "<<mes<<endl;
}
```

Set the Branch to fill local variable - you can update the value to that variable for any iEvt in the tree

Load the entry iEvt into the local variables mes, de, fisher & imass

The number of events or candidates in a tree (there is one per call to the tree->Fill() function).

eg1_loopOverTree.cc

Building a tree from scratch

```
// declare variables to use in the tree
Float_t   x,y;
Int_t     n[10];

// make the tree object
TTree mytree("tree", "title");

// set up the tree structure
TBranch * b_x = mytree.Branch("x", &x, "the variable x/F");
TBranch * b_y = mytree.Branch("y", &y, "the variable y/F");
TBranch * b_n = mytree.Branch("n", n,  "n[10]/I");


for(Int_t i = 0; i  < 100; i++)
{
        //do stuff to fill variables with a value
        [...]
        mytree.Fill();
}
```

Annotations on the Branch arguments: branch name, variable, comment/Type

An array used in this way is a pointer so you don't need the &

fill the tree with another entry you have to set the values of x, y and i before doing this

ROOT Exercise 3

1) Write a macro that takes the name of a file as an input, opens this and get the tree out of it to loop over (e.g. signal.root etc.)

2) Extend this macro so that you also make a second tree

– this should contain the variables:

`mes`
`de`
`newfish`

– do this while cutting on `mes` and `de` such that:

5.2 < `mes` < 5.29
-0.4 < `de` < 0.4

– loop over the events in the original tree writing those out that pass the cuts listed to the new tree and save to a new file.

There is an example solution saved as `myRootStuff.cc`

Some more advanced ROOT usage

• The last exercise made you write the essence of a simple analysis in root.

• As your analysis gets more complicated you'll probably introduce a few bugs and write some code that may well take a long time to run.

• When you start doing this – it is worth thinking about compiling your code to make sure it is robust and at the same time speed up its execution.

- use Makefiles to compile a stand alone application
  - faster run time execution
  - better error checking at compile time
  - get to debug output when things core dump
  - introduce you to (simple) Makefiles

The Makefile

use **`root-config`** to define libraries and include paths for you

this is a [tab]

The Content of a Makefile

```
LIBS=`root-config --libs`
CFLAGS=`root-config --cflags`
CC=g++

# set compiler options:
#    -g   = debugging
#    -O# = optimisation
COPT=-g

default:
	$(CC) $(COPT) main.cc -o main $(LIBS) $(CFLAGS)

clean:
	rm main
```

set compile options

file(s) to compile

output binary name

targets – e.g. `gmake` – compile the default target
`gmake clean` – run the clean target

Queen Mary University of London
m.bona@qmul.ac.uk

you will need to #include some files to make sure that the stand alone application finds the necessary declarations ….

Some useful files are:

- TNamed.h — knows about basic types such as Float_t
- TString.h
- TFile.h
- TTree.h
- TChain.h
- TH1F.h
- etc.

#include a file for each root class that you are using

If you use an object in root then you will need to #include the corresponding header file e.g.

```
#include "TNamed.h"
#include "TString.h"
etc.
```

A comprehensive list of classes can be found at:

http://root.cern.ch/root/Reference.html

ROOT Exercise 4

1) Write a file containing a main function – for example –put the following in a file called main.cc:

```
#include <iostream>
#include "myRootStuff.cc"

using namespace std;

int main(int argc, char * argv[]);

int main(int argc, char * argv[])
{
  // decode command line arguments
  char inputfile[256] = "";
  for(int iArg = 1; iArg < argc; iArg++)
  {
    if(!strcasecmp(argv[iArg],"-file")) strcpy( inputfile, argv[++iArg] );
  }

  // call root stuff in include file
  myRootStuff(inputfile);

  return 0;
}
```

include your root macro

prototype for main

main function that calls the macro entry point

2) Now you can **gmake** (or **make**), fix any errors and run the application – the application will be called **'main'** as specified after the **-o** in your **Makefile.**

ERRORS → will stop you being able to compile the program

Warnings → you might have a problem with the way you have written

it is good practice to make sure you don't have any warnings

3) run the application you have just compiled:

```
./main -file signal.root
```

→ If you got stuck with this at any point there are examples of Makefile, main.cc and myRootStuff.cc in Lectures/macros so you can take a look at these and play about with them…

- entry point:
  this is the thing that is called when the system runs a program. For a C/C++ program this is a function called main. For a ROOT macro, it is the function with the same name as the macro file.

Using scripts to run root

Now that you have a working executable you can run it using:

./main –file data/signal.root

An alternative way to do this is to pass the argument to the macro when you start root and remember to quit root when done:

```
root -l -b -q myRootStuff.cc'("root file name")'
```

You have two files to convert in this way – so you can run the program on one file and rename this. Then run the program on the second file. Imagine that you had 50 such files to do this to (i.e. many different possible backgrounds) … are you still going to do this by hand?

Exercise:

1) write a script to loop over the root files in **`data/`** and run your macro or binary program on these files.

→ *make sure that you're happy with doing this kind of thing as it will be useful*

usingScriptsToRunRoot.csh

More fun with ROOT
The following is additional material that will not be covered in Lectures.

The next section builds your knowledge of what you can do in root with more emphasis on presentation than the previous slides.

There is less formal structure in what follows

TCanvas and TPad

Canvas: a graphics window where histograms are displayed

- It is very easy to edit pictures on the canvas by clicking and dragging objects and right-clicking to get various menus.
- A ROOT canvas is a `TCanvas` object.
- the default canvas, c1, is created on first call to `Draw()`. This is equivalent to

```
TCanvas *c1=new TCanvas("c1","",800,600);
```

- Update canvas (if you make a change): `canvas->Update();`
- Tidy up canvas: `canvas->Clear();`
- Initially, the canvas has one pad which covers whole canvas → can use Divide

See FNAL tutorials and the ROOT User Guide for more on the use of canvases, pads and the ROOT GUI

- You can split canvas into several `TPads`, with

```
canvas->Divide(2,2);
canvas->Divide(nX, nY);
```

- You can plot different histograms on different pads
  - To change the pad you are working with use (where iPad ≤nX×nY)

```
canvas->cd(iPad)
```

- Save the contents of the canvas to a file

```
canvas->Write()
```

- Can save as ps, eps or gif using `SaveAs()` and `Print()`

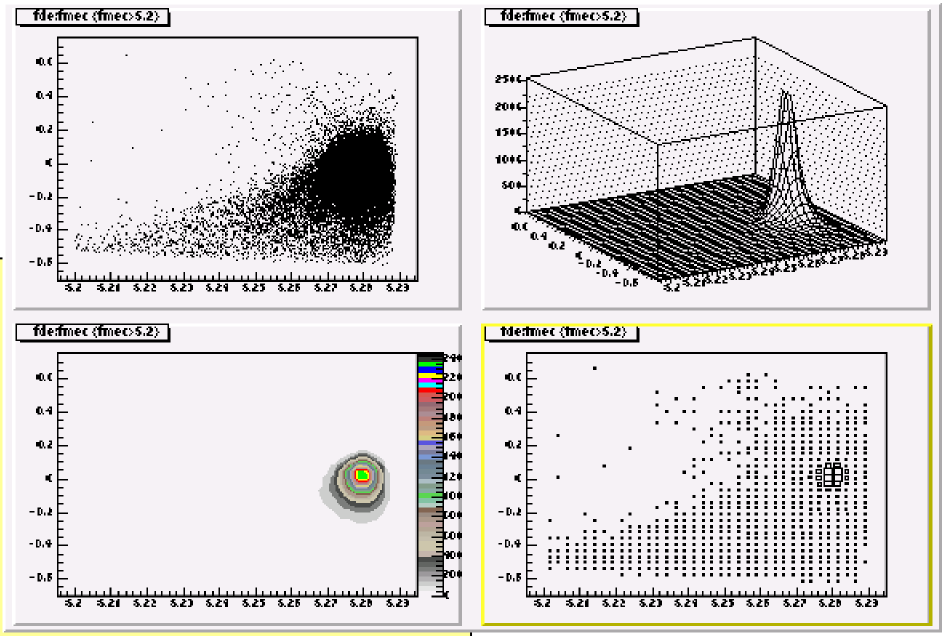
```
canvas->SaveAs("file.ps")
canvas->SaveAs("file.eps")   ← Can't run in batch mode
canvas->SaveAs("file.gif")

canvas->Print("file.ps")
etc..
```

- Also can make TPads by defining the co-ordinates by hand.

m.bona@qmul.ac.uk

Example use of a splitting up a TCanvas into 4 pads



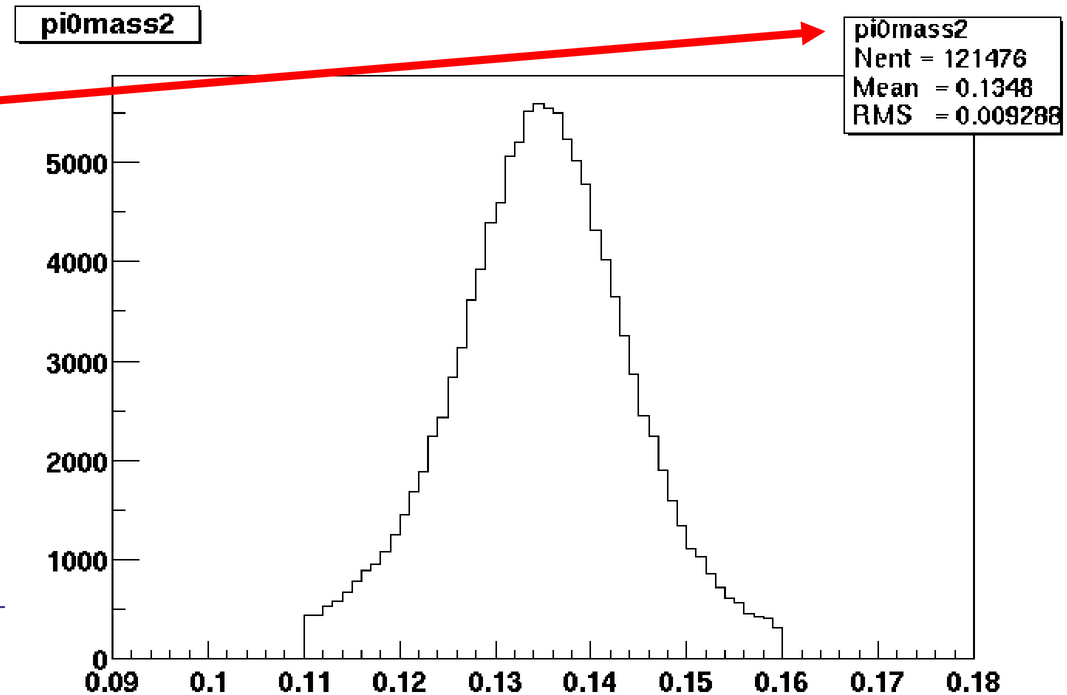
```
TFile f("data/signal.root")
TTree * mytree =
   (TTree*)f.Get("selectedtree")

TCanvas c1("c1")
c1.Divide(2,2);


c1.cd(1)
mytree->Draw("fde:fmec", "fmec>5.2")
c1.cd(2)
mytree->Draw("fde:fmec", "fmec>5.2", "surf")
c1.cd(3)
mytree->Draw("fde:fmec", "fmec>5.2", "contz0")
c1.cd(4)
mytree->Draw("fde:fmec", "fmec>5.2", "box")
```

Statistics Box

- Default placing – top right
- Various statistics can be displayed,
  - histogram name, mean, rms, number of entries, over- and under-flows [i.e. entries out of range]



To set up the stats box

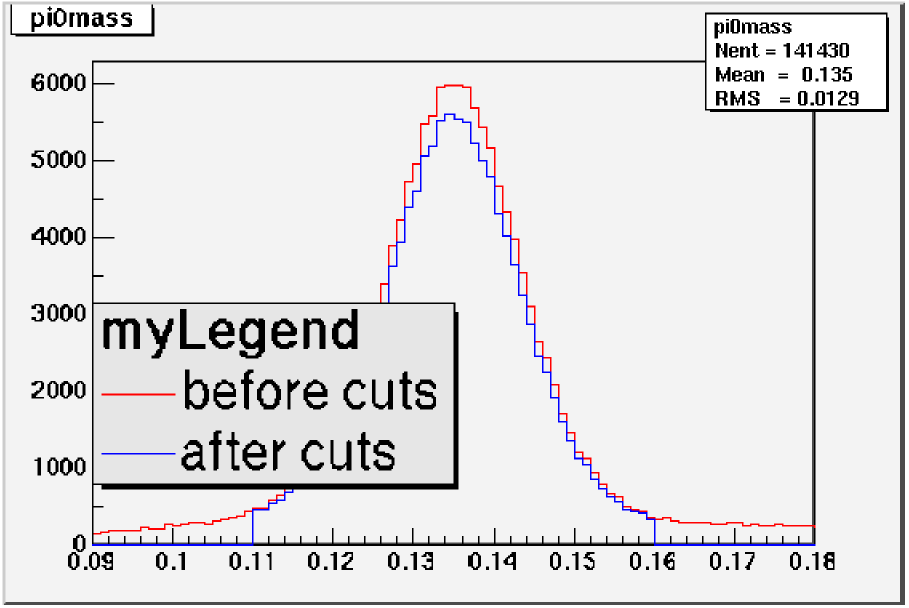
```
gStyle->SetOptStat();           //default setting
gStyle->SetOptStat(0);          //no stats box
h1->Draw();                     //update canvas
gStyle->SetOptStat(1111111);    //turn all options on
h1->Draw();
gStyle->SetOptStat(11);         //name & #events only
h1->Draw();
```

Legends

- `TLegend` - the key to the lines/histograms on a plot
- E.g. for a two-line histo (**h1** and **h2**):

```
// TLegend(x1,y1,x2,y2,header)
TLegend myLegend(0.1, 0.2, 0.5, 0.5, "myLegend")
myLegend.SetTextSize(0.04);
//AddEntry(): first arg must be pointer
myLegend.AddEntry(&h2, "after cuts", "l");
myLegend.AddEntry(&h1, "before cuts", "l");
myLegend.Draw();
```

- “l” (lowercase 'L') instructs ROOT to put a line in the legend.



Text Box

- Use text box (`TPaveText`) write on plots, e.g.:

```
TPaveText *myText = new TPaveText(0.2,0.7,0.4,0.85, "NDC");
                                        //NDC sets coords relative to pad
myText->SetTextSize(0.04);
myText->SetFillColor(0);          //white background
myText->SetTextAlign(12);
myTextEntry = myText->AddText("Here's some text.");
myText->Draw();
```

- Greek fonts and special characters:

```
h1->SetYTitle("B^{0} #bar{B^{0}}");    //must have brackets
h1->SetTitle("#tau^{+}#tau^{-}");      // to get super/subscript
```

The special characters that root knows are defined in the TLatex class. These are very similar to the use of latex maths commands but with '\' → '#'; e.g.

latex	→	root
\tau	→	#tau
\alpha	→	#alpha

not everything is available in TLatex

Symbols known to Tlatex.
N.B. these are all proceeded by a '#' symbol.

Lower case		Upper case		Variations	
alpha :	α	Alpha :	Α		
beta :	β	Beta :	Β		
gamma :	γ	Gamma :	Γ		
delta :	δ	Delta :	Δ		
epsilon :	∊	Epsilon :	Ε	varepsilon :	ε
zeta :	ζ	Zeta :	Ζ		
eta :	η	Eta :	Η		
theta :	θ	Theta :	Θ	vartheta :	ϑ
iota :	ι	Iota :	Ι		
kappa :	κ	Kappa :	Κ		
lambda :	λ	Lambda :	Λ		
mu :	μ	Mu :	Μ		
nu :	ν	Nu :	Ν		
xi :	ξ	Xi :	Ξ		
omicron :	ο	Omicron :	Ο		
pi :	π	Pi :	Π		
rho :	ρ	Rho :	Ρ		
sigma :	σ	Sigma :	Σ	varsigma :	ς
tau :	τ	Tau :	Τ		
upsilon :	υ	Upsilon :	Υ	varUpsilon :	ϒ
phi :	ϕ	Phi :	Φ	varphi :	φ
chi :	χ	Chi :	Χ		
psi :	ψ	Psi :	Ψ		
omega :	ω	Omega :	Ω	varomega :	ϖ

♣	#club	♦	#diamond	♥	#heart	♠	#spade
℘	#voidn	ℵ	#aleph	ℑ	#Jgothic	ℜ	#Rgothic
≤	#leq	≥	#geq	〈	#LT	〉	#GT
≈	#approx	≠	#neq	≡	#equiv	∝	#propto
∈	#in	∉	#notin	⊂	#subset	⊄	#notsubset
⊃	#supset	⊆	#subseteq	⊇	#supseteq	∅	#oslash
∩	#cap	∪	#cup	∧	#wedge	∨	#vee
©	#ocopyright	©	#copyright	®	#oright	®	#void1
™	#trademark	™	#void3	Å	#AA	å	#aa
×	#times	÷	#divide	±	#pm	/	#/
•	#bullet	°	#circ	···	#3dots	·	#upoint
ƒ	#voidb	∞	#infty	∇	#nabla	∂	#partial
″	#doublequote	∠	#angle	↵	#downleftarrow	¬	#corner
|	#lbar	|	#cbar	—	#topbar	{	#ltbar
⎩	#arcbottom	⎧	#arctop	⎡	#arcbar	⎣	#bottombar
↓	#downarrow	←	#leftarrow	↑	#uparrow	→	#rightarrow
↔	#leftrightarrow	⊗	#otimes	⊕	#oplus	√	#surd
⇓	#Downarrow	⇐	#Leftarrow	⇑	#Uparrow	⇒	#Rightarrow
⇔	#Leftrightarrow	∏	#prod	∑	#sum	∫	#int

m.bona@qmul.ac.uk

Fitting 1D Functions

- Fitting in ROOT based on Minuit (ROOT class: TMinuit)
- ROOT has 4 predefined fit functions, e.g.

  gaus: Gaussian function $\mathbf{f(x)=p_0exp\{-½[(x-p_1)/p_2]^2\}}$

  landau: Landau function (seethe literature for a full dfn).

  expo: exponential function $f(x) = p_0exp(p_1*x)$

  polyN: polynomial of order N, N=0, 1, 2, ... 9.

- Fitting a histogram with pre-defined functions, e.g.

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

- User-defined: 1-D function (TF1) with parameters:

```
TF1 *myFn =
   new TF1("myfn","[0]*sin(x) +[1]*exp(-[2]*x)",0,2);
```

- Set param names (optional) and start values (must do):

```
myFn->SetParName(0,"paramA");
myFn->SetParameter(0,0.75); //start value for param [0]
```

- Fit a histogram:

```
h1->Fit("myfn");
```

Fitting II

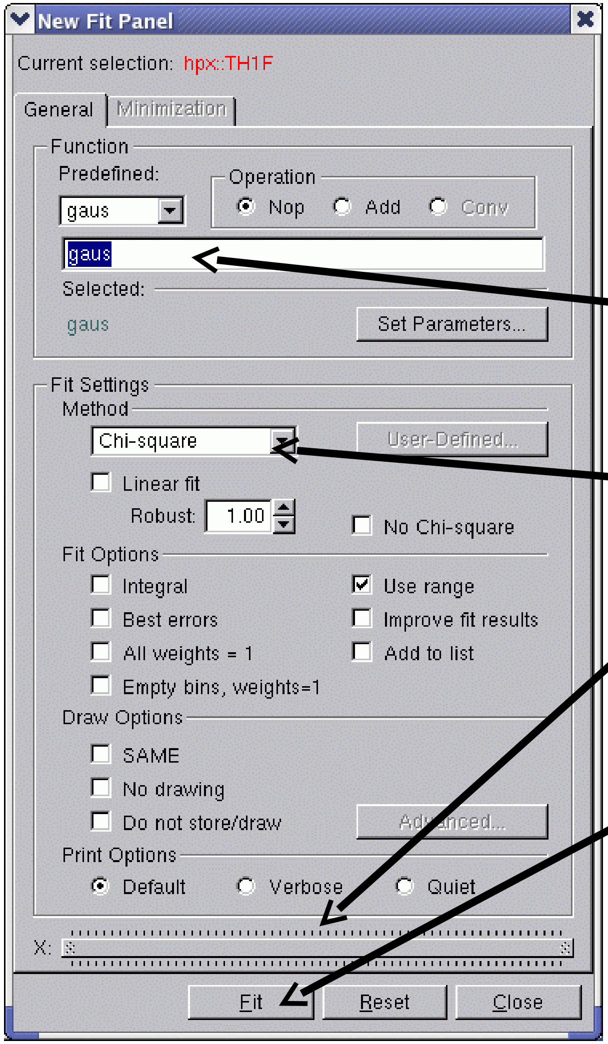
- Fitting with user-defined functions often requires solving a more complicated problem. Save the following as a macro called `myfunc.cc`

```
double myfunc(double *x, double *par)
{
   double arg=0;
   if (par[2]!=0) arg=(x[0]-par[1])/par[2];
   return par[0]*TMath::Exp(-0.5*arg*arg);
}
```

- `double *x` is a pointer to an array of variables
  - it should match the dimension of your histogram
- `double *par` is a pointer to an array of parameters
  - it holds the current values of the fit parameters
- now try and fit a histogram h1 with your function

```
.L myfunc.cc
TF1 *f1=new TF1("f1",myfunc,-1,1,3);
f1->SetParameters(10, h1->GetMean(), h1->GetRMS());
h1->Fit("f1");
```

Fitting III – The Fit Panel



- Open a fit panel for your histogram with:

```
myHistogram->FitPanel();
```

Specify the fitting function you want to use in the text box (has to be one known to ROOT).

Can switch between a χ^2 fit or a likelihood fit.

Can use the slide bar at the bottom to restrict the fit range to a sub-sample of your data.

To run or re-run a fit press the 'Fit' button.

Fitting IV

- If you have a complicated maximum-likelihood fit that you want to perform – don't do this by writing your own fit functions from scratch in ROOT.
- There is a package in ROOT called RooFit. This is a fitting package that is written by members of the HEP community to do complicated analyses (started on BaBar).
- There are tutorials on the web and the code is also available at the source forge: http://roofit.sourceforge.net/
- I would recommend that you think about using this if you have to do any unbinned maximum likelihood fit analysis as once you get started RooFit is a very powerful and flexible tool for easily building very complicated PDFs to fit to.

TBrowser – the ROOT GUI

- The `TBrowser` is the ROOT graphical interface
- It allows quick inspection of files, histograms and trees
- Make one with:

```
TBrowser tb;
```

- More formally:

```
TBrowser *tb = new TBrowser;
```

- Full details on how to manipulate the browse are in the ROOT user guide.

Using the TBrowser

- Start in ROOT with:

```
TBrowser tb;
```

- Any files already opened will be in the ***ROOT files*** directory
- Directory ROOT session started in will be shown too
- Otherwise click around your directories to find your files
- Click to go into chosen directory
- Double-click on any ROOT files you want to look at
  (you won't see an obvious response)
- Now go into the ***ROOT files*** directory
- Selected files now there
- Can click around files, directories, trees
- Can view histograms and leaves

Automatic code generation

You can simplify analysis of large ntuples by using built in automatic code generation methods available in ROOT. You have already learnt what you have to do to analyse NTuples in the examples – so you are now in a position to cheat to get the job done faster ☺

mytree->MakeCode()	- obsolete -> use MakeClass or MakeSelector instead
mytree->MakeClass()	- make a class with a loop() member function to run over the tree
mytree->MakeSelector()	- similar

Try these out to see what is auto generated. Toy should have a nameless macro in the first case and classes for the latter two. Usually I start from MakeCode(), but on occasion use MakeClass().

Summary

- You've now reviewed some of the basics UNIX, shell scripting & perl so that you can get these to do work for you – you'll probably need more practice
- had a crash course in root … and done some analysis
- seen histograms and ntuples close up.
- written a simple Makefile to compile your root code to make it faster
- used a script to run a job – automating work for you

The next step with this is to practice what you've learnt – this way you'll better recognise when to do certain things to make your life easier than it currently is

If you find yourself wasting time doing the same thing over and over again there is something out there to learn so that you can save time and get back to the real job at hand …. Physics!

Where to Get More Information ROOT

- The ROOT homepage:
  http://root.cern.ch/
  – examples, HOWTOs, tutorials, class information, ROOT source code
  – RootTalk mailing list – high traffic, great search facility
- Fermilab's three-day ROOT course
  http://patwww.fnal.gov/root
- SLAC's root web pages:
  http://www.slac.stanford.edu/BFROOT//www/Computing/Offline/ROOT/index.html
- Other students/post-docs in the group
- Email me: m.bona@qmul.ac.uk