ROOT: introductory tutorial

Marcella Bona

for QMUL undergraduates and Particle Physics projects
October 4th 2017

Thanks for the help from
Rebecca Lane, Alex Owen, Tom Stevenson
Eddie Thorpe and Cozmin Timis

Based on material from
Adrian Bevan, Marcella Bona,
Alex Owen and Cozmin Timis

Aims of this tutorial

The aim of this tutorial is to give you a smooth start in using ROOT.
By the time you have worked through this you should be able to:

- *connect to our PPRC computer that has ROOT installed*
- *start ROOT and get out of it*
- *use ROOT interactively*
- *know how to use histograms, ntuples, files etc …*
- *write and run a simple macro*
- *fit to histogram data*
- *know where to go for more information*

This set of lectures has been tested most recently for ROOT 5 and 6

ROOT

- What is ROOT?

  C++ based analysis toolkit developed at CERN
  and used in most (if not all) particle physics/high energy analyses
  code available and downloadable on the web
  actively developed by many people
  you do not need to learn another syntax to use it:
  you can write small programs called *root macros*
  and they are just C++ programs that can run
  within the ROOT environment or standalone
  using the ROOT library

  ➔ manual is large (over 300 pages)
  ➔ lots of material online, but it could be quite dispersive
  ➔ some example macros are available in the ROOT package itself
  ➔ some good web based courses available as well

ROOT

- What is ROOT?

  C++ based analysis toolkit developed at CERN

- Useful resources
  - User Guide etc:

    http://root.cern.ch/

    download root from here

    HOW-TOs, Tutorials and class structure on web

One step back: getting to ROOT

Let's try to give you the possibility to use ROOT on QMUL machines:

ROOT is installed on a linux computer made available to you by Alex and Cozmin. The name of the computer is:

students2.ph.qmul.ac.uk

to get there you need to use a program that connects the computer you are using to this linux machine over the QMUL network:

MobaXterm

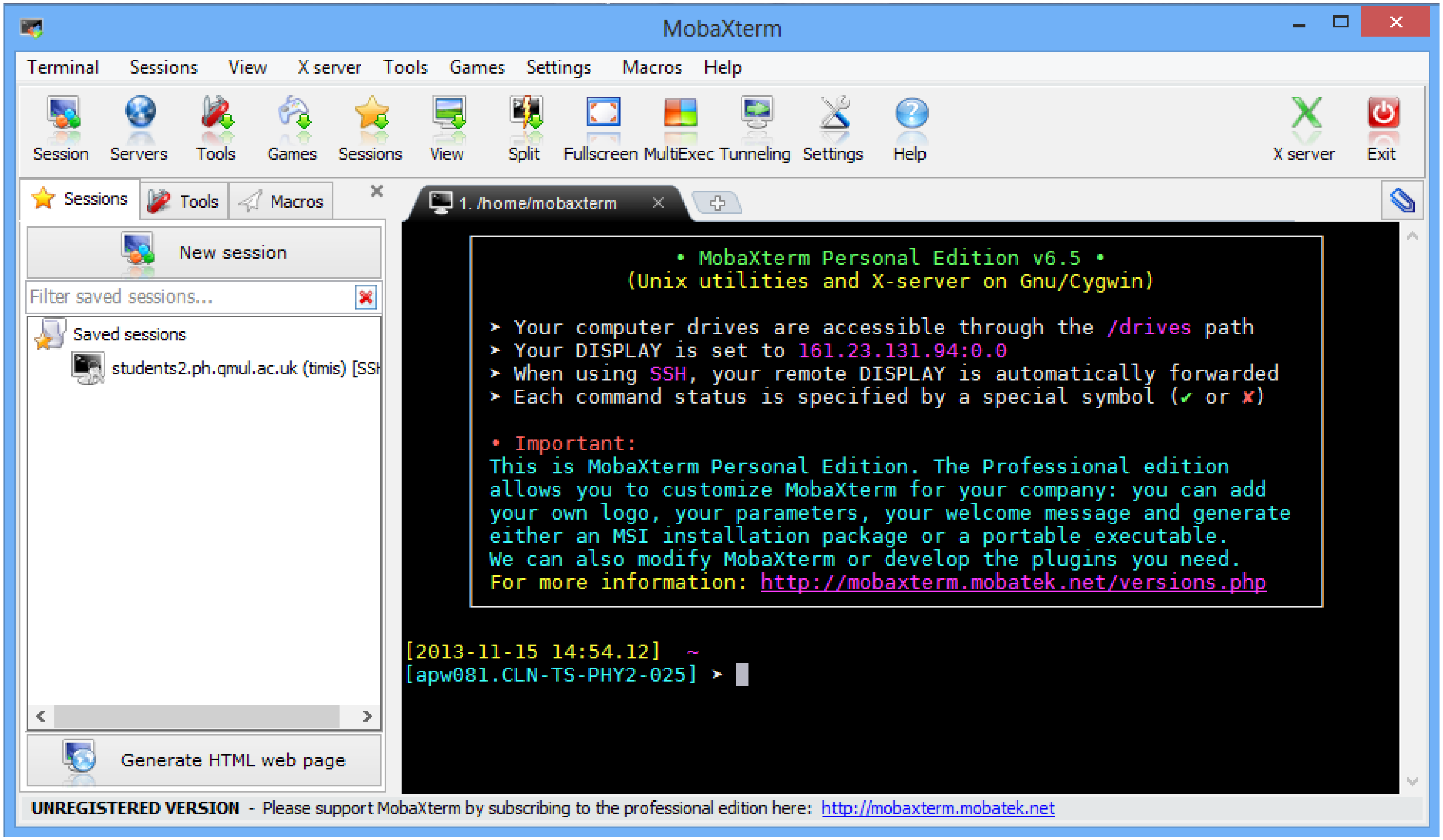
So now start mobaXterm on the computer you are at.

If you need to connect from a linux machine you need to do:

```
ssh -X username@students2.ph.qmul.ac.uk
```

One step back: getting to ROOT

This is how the screen looks like once started:

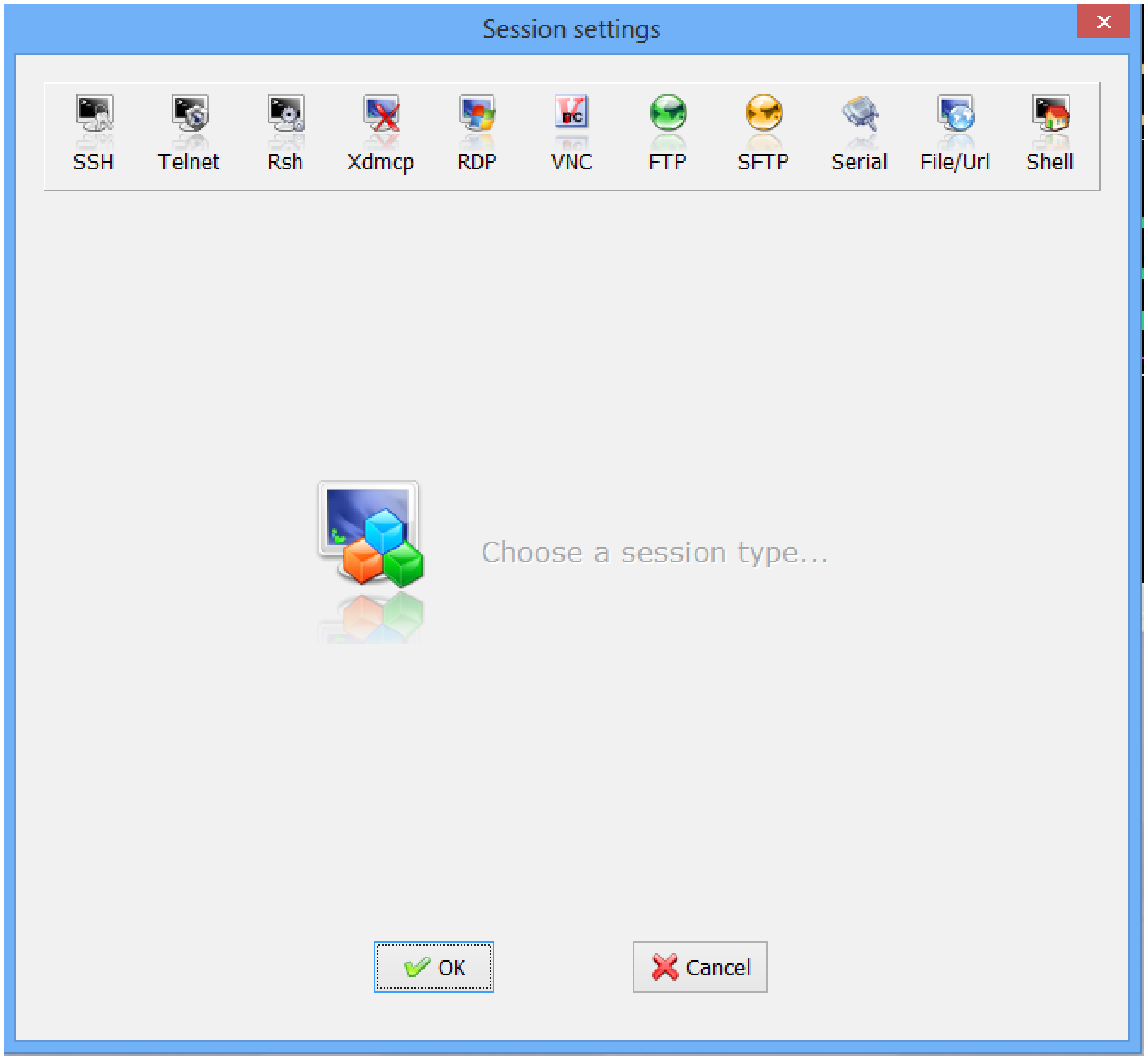

MobaXterm
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split Fullscreen MultiExec Tunneling Settings Help X server Exit
Sessions Tools Macros
1. /home/mobaxterm
New session
Filter saved sessions...
Saved sessions
students2.ph.qmul.ac.uk (timis) [SSH
Generate HTML web page

```
                • MobaXterm Personal Edition v6.5 •
             (Unix utilities and X-server on Gnu/Cygwin)

 ➤ Your computer drives are accessible through the /drives path
 ➤ Your DISPLAY is set to 161.23.131.94:0.0
 ➤ When using SSH, your remote DISPLAY is automatically forwarded
 ➤ Each command status is specified by a special symbol (✓ or ✗)

 • Important:
 This is MobaXterm Personal Edition. The Professional edition
 allows you to customize MobaXterm for your company: you can add
 your own logo, your parameters, your welcome message and generate
 either an MSI installation package or a portable executable.
 We can also modify MobaXterm or develop the plugins you need.
 For more information: http://mobaxterm.mobatek.net/versions.php

[2013-11-15 14:54.12]  ~
[apw081.CLN-TS-PHY2-025] ➤
```

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: http://mobaxterm.mobatek.net

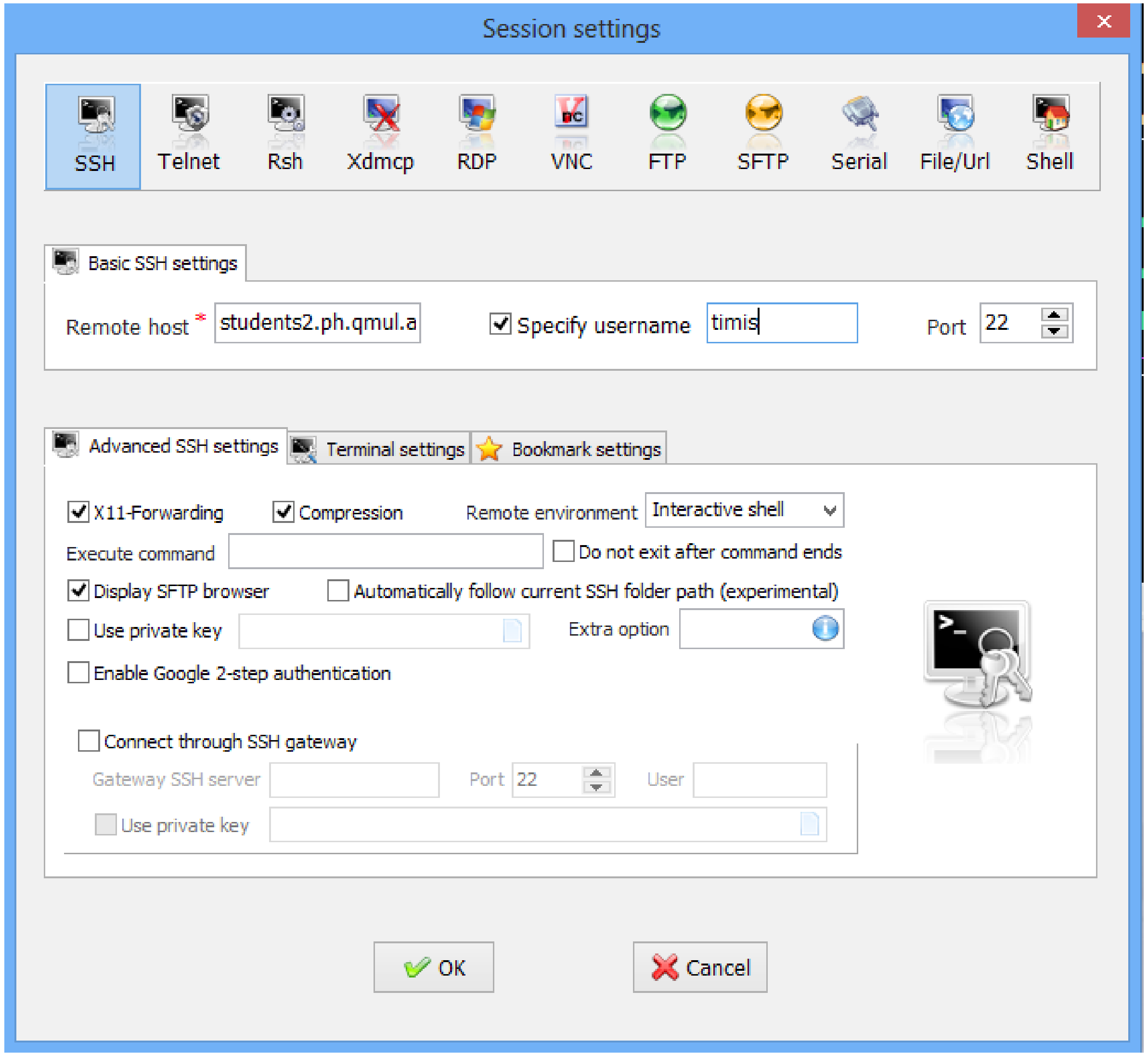
One step back: getting to ROOT

Click on “Session” so you'll get:



One step back: getting to ROOT

Select ssh as session type and input the computer name in the remote host box and the username in the username box


Session settings

Basic SSH settings

Remote host * students2.ph.qmul.a ☑ Specify username timis Port 22

☑ X11-Forwarding ☑ Compression Remote environment Interactive shell

Execute command ☐ Do not exit after command ends

☑ Display SFTP browser ☐ Automatically follow current SSH folder path (experimental)

☐ Use private key Extra option

☐ Enable Google 2-step authentication

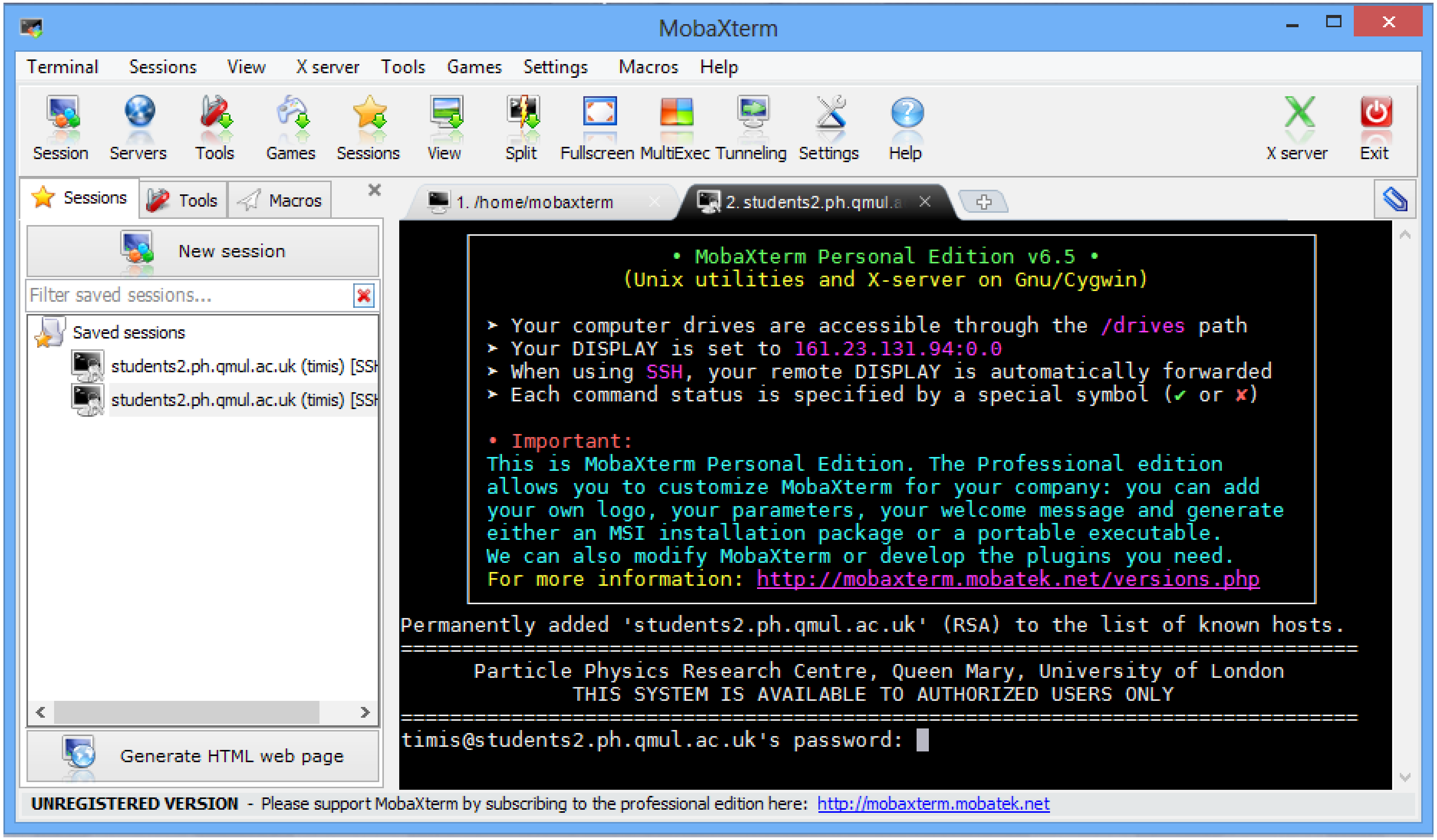
☐ Connect through SSH gateway

Gateway SSH server Port 22 User

☐ Use private key

One step back: getting to ROOT

Click OK and type your password when prompted


MobaXterm
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split Fullscreen MultiExec Tunneling Settings Help X server Exit
Sessions Tools Macros
New session
Filter saved sessions...
Saved sessions
students2.ph.qmul.ac.uk (timis) [SSH
students2.ph.qmul.ac.uk (timis) [SSH
Generate HTML web page
1. /home/mobaxterm
2. students2.ph.qmul.a
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: http://mobaxterm.mobatek.net

```
          • MobaXterm Personal Edition v6.5 •
        (Unix utilities and X-server on Gnu/Cygwin)

 ➤ Your computer drives are accessible through the /drives path
 ➤ Your DISPLAY is set to 161.23.131.94:0.0
 ➤ When using SSH, your remote DISPLAY is automatically forwarded
 ➤ Each command status is specified by a special symbol (✔ or ✘)

 • Important:
 This is MobaXterm Personal Edition. The Professional edition
 allows you to customize MobaXterm for your company: you can add
 your own logo, your parameters, your welcome message and generate
 either an MSI installation package or a portable executable.
 We can also modify MobaXterm or develop the plugins you need.
 For more information: http://mobaxterm.mobatek.net/versions.php

Permanently added 'students2.ph.qmul.ac.uk' (RSA) to the list of known hosts.
=============================================================================
      Particle Physics Research Centre, Queen Mary, University of London
              THIS SYSTEM IS AVAILABLE TO AUTHORIZED USERS ONLY
=============================================================================
timis@students2.ph.qmul.ac.uk's password: █
```

Few linux/unix commands

At this point you are on a linux shell.
Basic commands to survive a linux shell:
- "folders" in linux are "directories"

`ls` <dir>	list the content of a directory
`cd` <dir>	change directory to the specified one
`mkdir` <dir>	make a new directory
`cp` <file> <newfile>	make a copy of a file
`mv` <file name> <new file name>	rename a file
`more` <file>	file viewer
`less` <file>	file viewer (more versatile than more)
`g(un)zip` <file>	(un)zip a file up
`tar -cvf` somefile.tar <dir>	make a tar file (archive files or directory trees)
`tar -xvf` somefile.tar	unpack a tar file
`tar -cvzf` somefile.tgz <dir>	make a tar file and zip it up in one go
`tar -xvzf` somefile.tgz	unpack a zipped tar file

Getting to ROOT...

In the linux shell, we need to set up things in order to use root:

enter the commands:

```
> scl enable devtoolset-2 python27 bash
> module load root
```

this allows root to be available to use and also to sychronise the compiler with the root version.

This allows you to use **root 6**

Getting to ROOT...

Root 6 is relatively new and includes a number of changes with respect to the previous versions. Hence there are less tutorials and help online.

You can use the previous version **root 5.34** and in that case you can do:

```
> exit
> module unload root
> module load root_5.34.00
```

or you can close the mobaXterm and restart again and do directly:

```
> module load root_5.34.00
```

Back to ROOT..

ROOT is a data analysis toolkit that has one main application

`root`	the main program that you run

- once the environment is set like detailed in the previous slides
- just launch the executable writing:

```
> root
```

then you will be in the ROOT environment
An useful argument to be added to the executable name is:

```
> root -l
```

this avoids the splash screen
and makes the start of root a little quicker
especially if the connection is slow

Back to ROOT..

ROOT is a data analysis toolkit that has one main application

`root`	the main program that you run

- once the environment is set like detailed in the previous slides
- just launch the executable writing:

```
> root
```

then you will be in the ROOT environment

- ROOT (sort of) uses C++ syntax:
  - you can give C++-style commands within the ROOT environment
  - when you give interactive commands
    you are using an interactive C++ interpreter (CINT or Cling)
  - similarly we will see that we can run your program (root macro)
    without compiling, hence using the interpreter

CINT/Cling commands

- CINT/Cling commands always start with a dot "."
- e.g:

`.q`	*quit, get out of the interactive ROOT session*
`.?`	*help; get list of available commands*

CINT/Cling commands

- CINT/Cling commands always start with a dot "."
- e.g:

`.x hi.cc`	*run a macro*
`.L hello.cc`	*load a macro in order to use the functions in the macro*
`hello("marcella")`	*run the function with the required argument*

```
wget https://www.dropbox.com/s/rvvn70ib9vlcj0g/hi.cc
wget https://www.dropbox.com/s/pcpq1in63cje32d/hello.cc
```

CINT/Cling commands

- interactively you can also use C++ syntax:
  - declarations of objects and use of functions
  - for example:
    - open a root file:

```
TFile *myfile = TFile::Open("signal.root")
```

```
wget https://www.dropbox.com/s/ycryf6khdgic8ge/signal.root
wget https://www.dropbox.com/s/02amcswqg4ts7ik/continuum.root
```

CINT/Cling commands

- interactively you can also use C++ syntax:
  - declarations of objects and use of functions
  - for example:
    - open a root file:

```
TFile *myfile = TFile::Open("signal.root")
```

this is declaring an object of type TFile
  that is a file with an extension .root
root files are ROOT specific files and they are use to contain data
  in different ROOT formats,
  i.e. as different ROOT objects.

CINT/Cling commands

- interactively you can also use C++ syntax:
  - declarations of objects and use of functions
  - for example:
    - open a root file:

```
TFile *myfile = TFile::Open("signal.root")
myfile->ls()
```

the TFile object I declared is a pointer
hence I need to call the functions implemented in the TFile class with the arrow →
ls() just lists everything is in the root file

https://root.cern.ch/doc/master/classTFile.html

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 -l signal.root
  ```

  the pointer will be called `_file0`

Some basic concepts

- Histograms:
  - Plots of data as a function of 1, 2 or 3 variables
  - Show distributions of relevant variables and correlation between the variables
  - ROOT objects: TH1

  https://root.cern.ch/doc/master/classTH1.html

Some basic concepts

- Histograms:
  - Plots of data as a function of 1, 2 or 3 variables
  - Show distributions of relevant variables and correlation between the variables
  - ROOT objects: TH1

  https://root.cern.ch/doc/master/classTH1.html

- Ntuples:
  - A more complicated data format that stores information on an event or candidate basis:
    - creates a matrix-like structure where:
      - a row contains the values of the variables in a single “collision event” or “composite particle candidate”
      - a column contains the values of a single variable describing a specific characteristic of the event/particle
      - ROOT object: TTree

  https://root.cern.ch/doc/master/classTTree.html

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)

ROOT Data Types II

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")
```

Histograms

1, 2 and 3D binned plots of the distribution of variables
– good for visualising how the data are behaving:

types

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

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

→ these are the ROOT classes so in the code we need to declare a TH1 object and then we can access all the functions or methods implemented in the TH1 class

```
TFile *myfile = TFile::Open("signal.root")
myfile->ls()
TH1D* Bmass = (TH1D*)myfile->Get("mecdis")
Bmass->Draw()
```

Histograms

```
TFile *myfile = TFile::Open("signal.root")
myfile->ls()
TH1D* Bmass = (TH1D*)myfile->Get("mecdis")
Bmass->Draw()
```

Get() function is in TFile class
and returns a pointer to a generic TObject

hence we can check what kind of object was stored in
the file with ls() and then we can istantiate it explicitly:

in this case, the object is a TH1D so I declare a pointer
to a TH1D and I assign to it the object from the file

Histograms

1, 2 and 3D binned plots of the distribution of variables

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

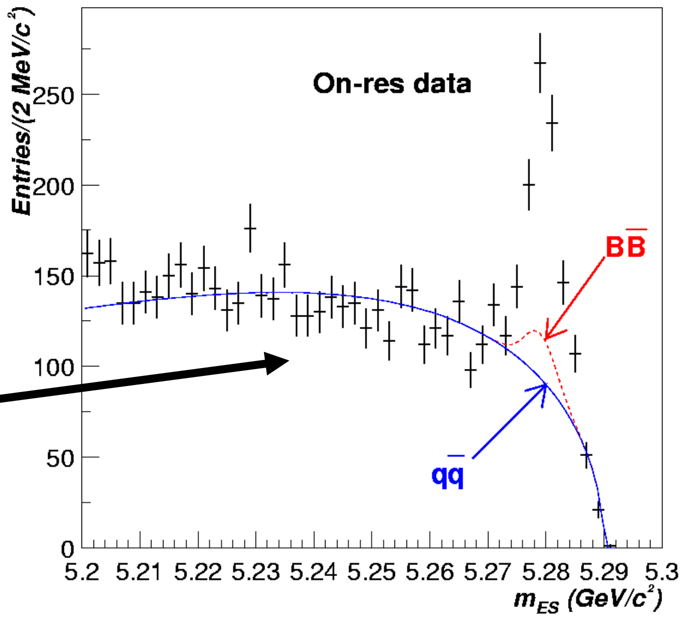
```
root[10] myHist->Draw()
```

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



marcella bona

NTuples

- data structure on an entry by entry basis (e.g. candidate or event)
  Ttree (or 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
  - apply selection cuts on variables as you draw them
  - create and fill histograms with the values of the variables

- NTuples are a lot more flexible than histograms as you can retain the correlations between the variables and the view of the complete characteristics of the event/candidate

- again you need to declare an object of the class TTree

```
TTree* tree = (TTree*)myfile->Get("selectedtree");
```

NTuples

- you need to declare an object of the class TTree

```
TTree* tree = (Ttree*)myfile->Get("selectedtree")
tree->Print()
```

- the Print function of the TTree class allows you to see all the variables contained in the TTree/NTuple.

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` signal.root using

```
root[0] TFile signal("signal.root")
```

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

```
root [1] signal.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` signal.root using

```
root[0] TFile signal("signal.root")
```

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

```
root [1] signal.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

comment

object version number

key object type

key name

a TTree object (an NTuple)

Objects in a file: e.g. TTree

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*)signal.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.

Histograms

Declare with:

```
TH1F h1(arguments …)
```

- Make your first 1D histogram:

  n_{bins} x_{min} x_{max}

  ```
  TH1F h1("h_name", "h_title", 10, 0.0, 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)*

  x value to fill histogram at

- Try drawing the histogram when you have a few entries

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

ROOT exercise 1

1) Open the files found at these links

    signal.root

    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

Histograms from a TTree

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

variable name

means make hist

hist name

```
mySignalTree->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 pointer to a histogram made to have the content corresponding to that of the tree

Histograms from a TTree (II)

```
mySignalTree->Draw("mes>>tmphist");
mySignalTree1->Draw("mes>>+tmphist");
mySignalTree2->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.:

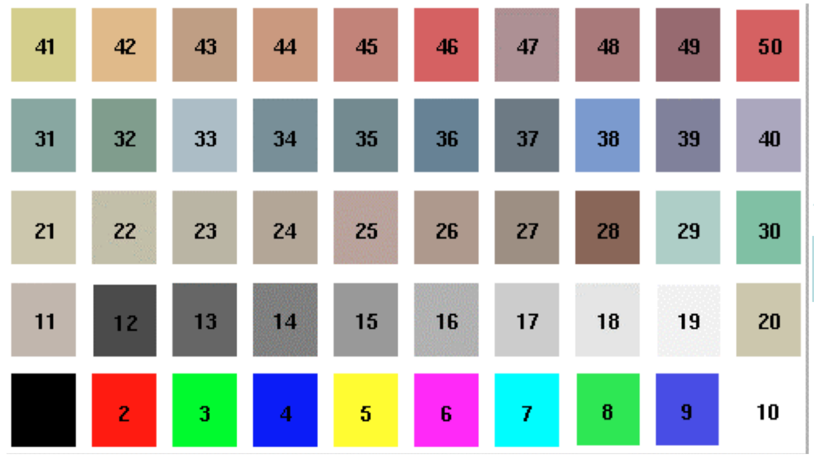
```
TH1F tmphist("tmphist", "", 25, 5.2, 5.3);
mySignalTree->Draw("mes>>tmphist");
tmphist.Draw("e");
```

Histograms made nicer

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

`myHist.SetFillColor(1)`	Change the fill colour.
`myHist.SetFillStyle(0)`	Change the fill style.
`myHist.SetLineColor(1)`	Change the line colour.
`myHist.SetLineStyle(0)`	Change the line style.
`myHist.SetLineWidth(1)`	Change the line width.

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


41 42 43 44 45 46 47 48 49 50
31 32 33 34 35 36 37 38 39 40
21 22 23 24 25 26 27 28 29 30
11 12 13 14 15 16 17 18 19 20
2 3 4 5 6 7 8 9 10

```
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!

```
myHist.SetFillColor(2)
```
or
```
myHist.SetFillColor(kRed)
```

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

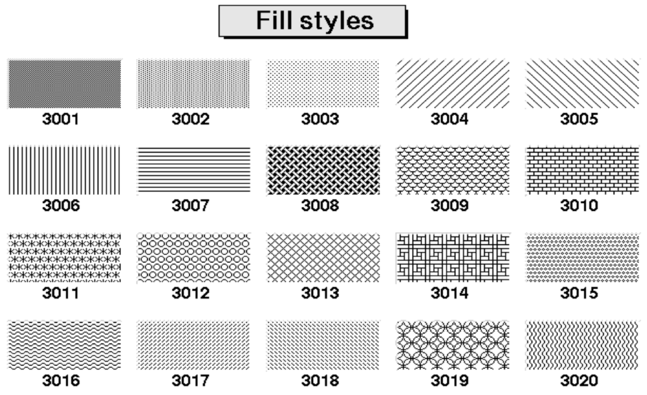
marcella bona

Histograms made nicer (II)

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

`myHist.SetFillColor(1)`	Change the fill colour.
`myHist.SetFillStyle(0)`	Change the fill style.
`myHist.SetLineColor(1)`	Change the line colour.
`myHist.SetLineStyle(0)`	Change the line style.
`myHist.SetLineWidth(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: always label your axes!

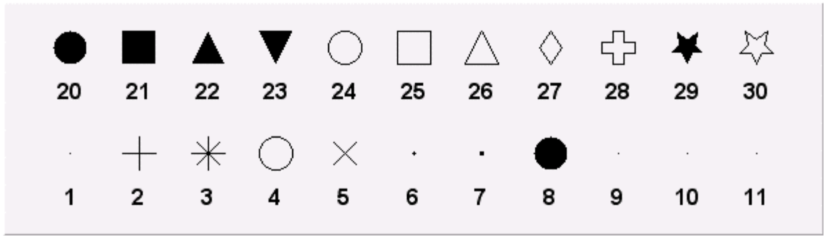
```
myHist.SetXTitle("This is the x-axis")
myHist.SetYTitle("This is this y-axis")
```

Histograms made nicer (III)

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`	....
`kDashDotted`	· - · - ·

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



2D Histograms

$$\texttt{TH2F myHist("h\_name", "h\_title", }\underbrace{\texttt{10, 0.0, 10.0}}_{\text{x axis co-ordinates}}\texttt{, }\underbrace{\texttt{20, -10.0, 20.0}}_{\text{y axis co-ordinates}}\texttt{);}$$

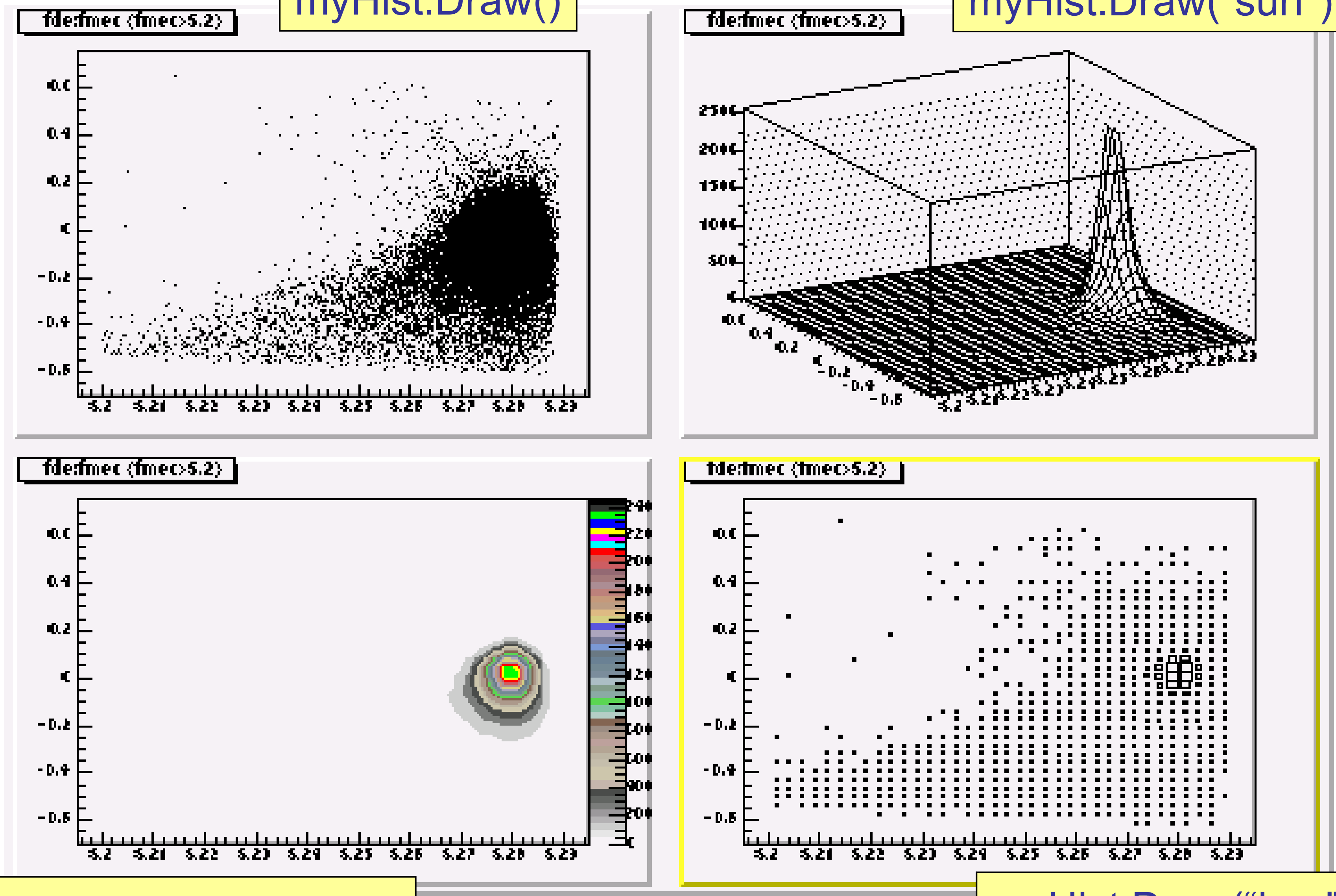
- 2D histograms behave the same as 1D histograms

- 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.*

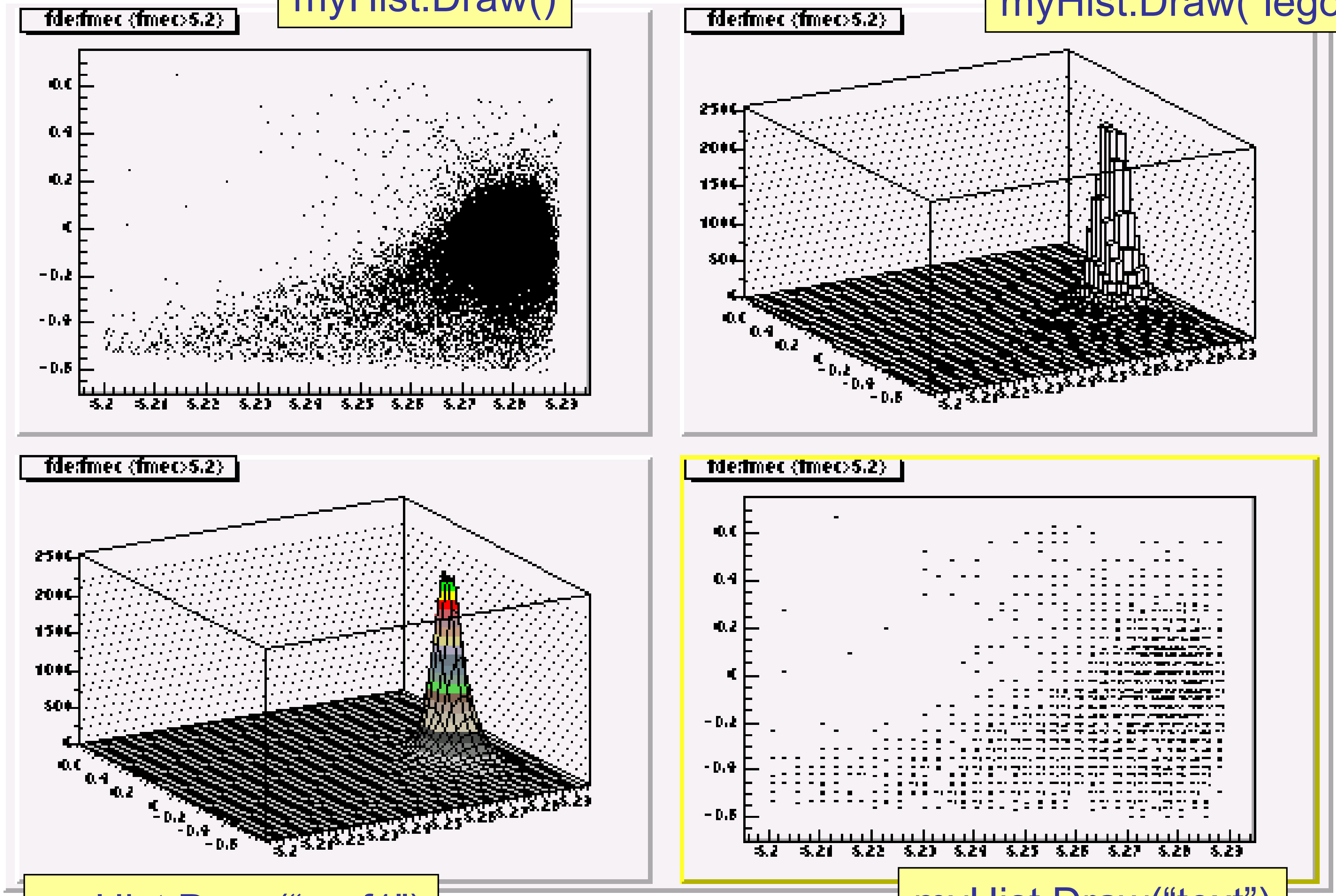
- These draw options also work for trees myHist.Draw()

myHist.Draw("surf")



myHist.Draw("contz0")

myHist.Draw("box")



myHist.Draw()

myHist.Draw(“lego”)

myHist.Draw(“surf1”)

myHist.Draw(“text”)

More on histograms

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

```
        myHist.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 `myHist.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 `myHist.Draw("e")`, ROOT will draw error bars for you, where $\sigma=\sqrt{N}$.

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)

- 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.

Macros

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

```
{
  TFile *myfile = TFile::Open("signal.root");
  myfile->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 but the convention is to use .cc or .cpp or .cxx
- To execute an un-named macro:

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

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

Text editor

- it is really up to you, you can choose whatever text editor you like

- however if in doubt, here is some thoughts and a suggestion:
  - it is useful if your text editor knows about C++

    i.e. it recognises that you are writing C++ code
  - it is useful to have a text editor that you can use “in shell”

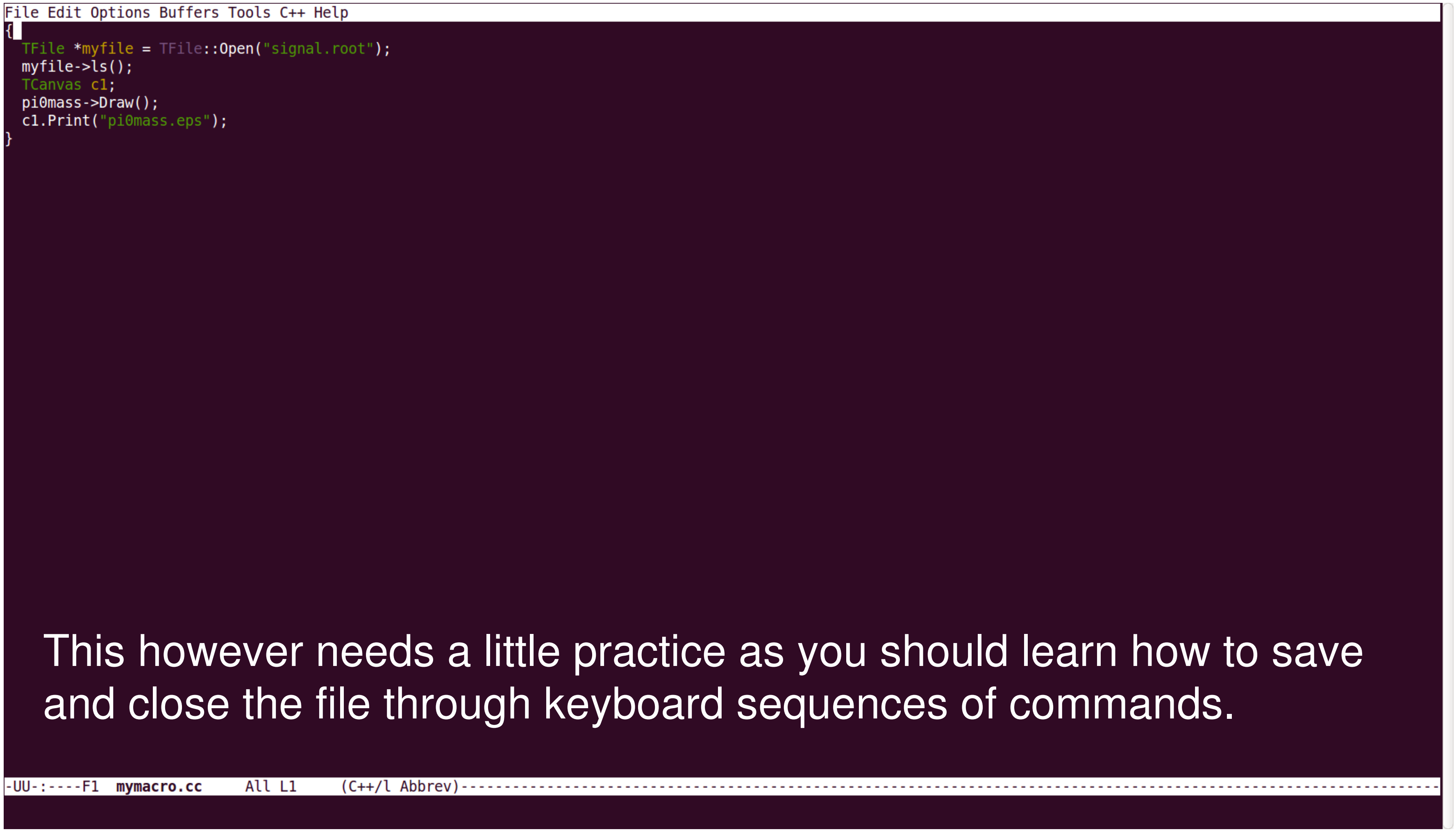
    to edit directly on the computer you are accessing online
  - one such an editor is called emacs (or xemacs)
    - it is very powerful and real geeks can do amazing things with it
    - however it is usable in a plain and simple way

  – try to write in the students2 shell

    » **`emacs -nw myMacro.cc`**

Text editor

This opens the file within the shell you are already using: **nw**==no window



```
{
  TFile *myfile = TFile::Open("signal.root");
  myfile->ls();
  TCanvas c1;
  pi0mass->Draw();
  c1.Print("pi0mass.eps");
}
```

```
-UU-:----F1  mymacro.cc      All L1     (C++/l Abbrev)-----------------------------------------------------------
```

This however needs a little practice as you should learn how to save and close the file through keyboard sequences of commands.

[ctrl-x]+[ctrl-s]	save a file
[ctrl-x]+[ctrl-c]	close emacs

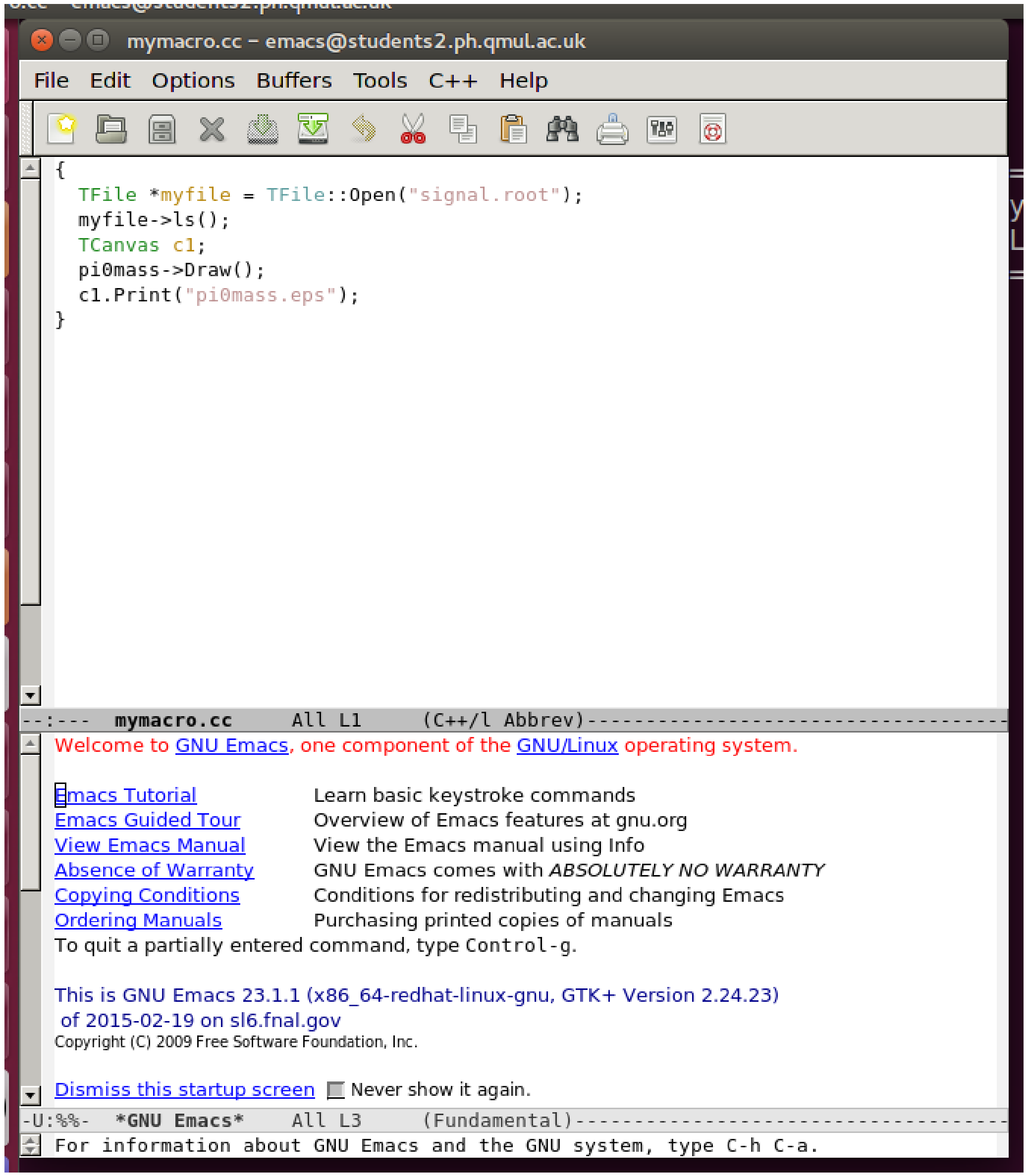
Text editor

if you just type

> **`emacs myMacro.cc`**

you will open another window with buttons and all and that makes easier to edit

it might however be a little slower as you are opening over the network.


mymacro.cc – emacs@students2.ph.qmul.ac.uk
File Edit Options Buffers Tools C++ Help

```
{
  TFile *myfile = TFile::Open("signal.root");
  myfile->ls();
  TCanvas c1;
  pi0mass->Draw();
  c1.Print("pi0mass.eps");
}
```

```
--:---  mymacro.cc      All L1     (C++/l Abbrev)-----------------------------------
```

Welcome to GNU Emacs, one component of the GNU/Linux operating system.

Emacs Tutorial	Learn basic keystroke commands
Emacs Guided Tour	Overview of Emacs features at gnu.org
View Emacs Manual	View the Emacs manual using Info
Absence of Warranty	GNU Emacs comes with *ABSOLUTELY NO WARRANTY*
Copying Conditions	Conditions for redistributing and changing Emacs
Ordering Manuals	Purchasing printed copies of manuals

To quit a partially entered command, type Control-g.

This is GNU Emacs 23.1.1 (x86_64-redhat-linux-gnu, GTK+ Version 2.24.23)
of 2015-02-19 on sl6.fnal.gov
Copyright (C) 2009 Free Software Foundation, Inc.

Dismiss this startup screen ☐ Never show it again.

```
-U:%%-  *GNU Emacs*     All L3     (Fundamental)------------------------------------
For information about GNU Emacs and the GNU system, type C-h C-a.
```

Macros (II)

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

```
void hi(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 hi.cc
```

  or from the command line with

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

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

```
#include "hi.cc"
```

Macros (III)

included file

macro entry point

the function call

e.g. save this as mainFunc.cc

```
#include "hi.cc"

void mainFunc(void)
{
   cout << "calling included function" << endl;
   hi();
   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")'
```

Macros (IV)

To build an analysis, you can build a sequence of macros:

- Macros can call and use other macros.

- Syntax to load a macro from a file:
  `gROOT->LoadMacro("myFile.cc");`
  formal version of the CINT/Cling command line `.L myFile.cc`

- If you 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).
  – see later for compiling

More on 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*)signal.Get("selectedtree")
```

Using TFile::Get()

```
root[0] TFile signal("data/signal.root")
root [1] signal.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 $B^0 \rightarrow \pi^0\pi^0$ 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

How to create a new file

```
TFile file("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 tree("tree", "title");
TBranch * b_x = tree.Branch("x", &x, "x/F");
TBranch * b_y = tree.Branch("y", &y, "y/F");
TBranch * b_z = tree.Branch("n", n, "n[10]/I");

// do stuff

// dump and close
file.Write();
file.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).

How to create a new file (II)

```
//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 tree("tree", "title");
TBranch * b_x = tree.Branch("x", &x, "x/F");
TBranch * b_y = tree.Branch("y", &y, "y/F");
TBranch * b_z = tree.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 file("myNewFile.root", "RECREATE", "comment");
aHist.Write();
a2DHist.Write();
tree.Write();

file.Write();
file.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 * tree = (TTree*)file.Get("selectedtree");

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

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

// loop over the candidates in the TTree
for(int iEvt = 0; iEvt < tree->GetEntries(); iEvt++)
{
  tree->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).

see myRootStuff.cc

Building a tree from scratch

branch name

variable

comment/Type

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

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

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


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

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 2

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.

```
wget https://www.dropbox.com/s/3ciw4ooocts4gif/myRootStuff.cc
wget https://www.dropbox.com/s/f7xj5kmapdepirr/compiledStuff.cc
```

ROOT Exercise 2

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

– 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.

3/ you can then run your macro from within ROOT:

```
.x myRootStuff.cc("signal.root")
```

or you can compile and run:

```
.x compiledStuff.cc++("signal.root")
```

```
wget https://www.dropbox.com/s/3ciw4ooocts4gif/myRootStuff.cc
wget https://www.dropbox.com/s/f7xj5kmapdepirr/compiledStuff.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. `make` – compile the default target
`make clean` – run the clean target

In the code to be compiled

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

Some useful files are:

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

A comprehensive list of classes can be found at:

https://root.cern.ch/root/html/ClassIndex.html

ROOT Exercise 3

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

ROOT Exercise 3

2) Now you can **`make`** `(or g`**`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.

ROOT Exercise 3

in case you are having trouble finishing this:
have a look at these two files:

```
wget https://www.dropbox.com/s/u721qjhqqce4b3p/Makefile
wget https://www.dropbox.com/s/566dtggxzrbx5fz/main.cc
```

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): `c1->Update();`
- Tidy up canvas: `c1->Clear();`
- Initially, the canvas has one pad which covers the whole canvas $\rightarrow$ can use the Divide function to create more than one pad.

TCanvas and Tpad (II)

- 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()
```

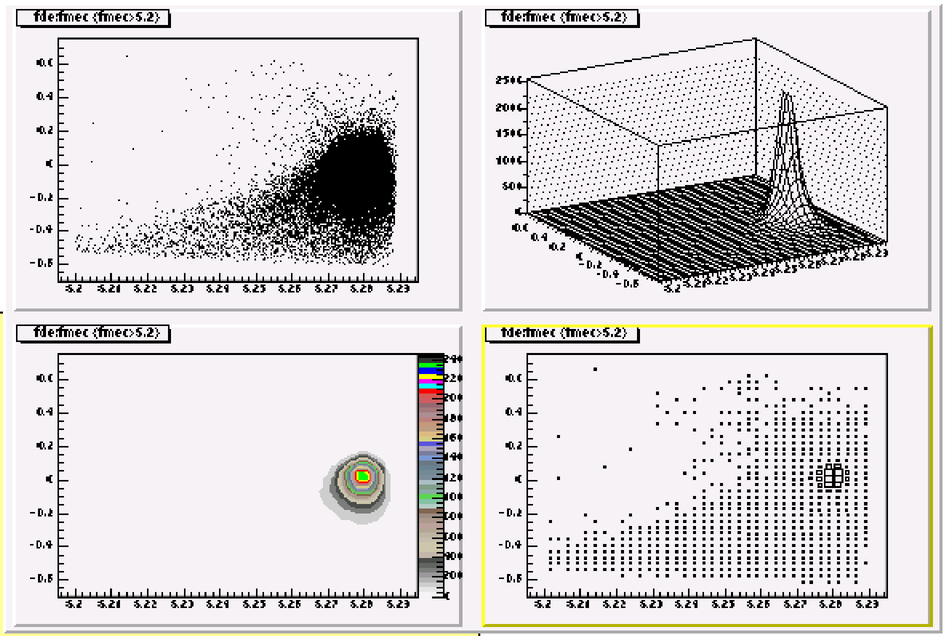
- Can save as eps, pdf or png using `SaveAs()` and `Print()`

```
canvas->SaveAs("file.eps")
canvas->SaveAs("file.pdf")
canvas->SaveAs("file.png")
...etc...
```

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

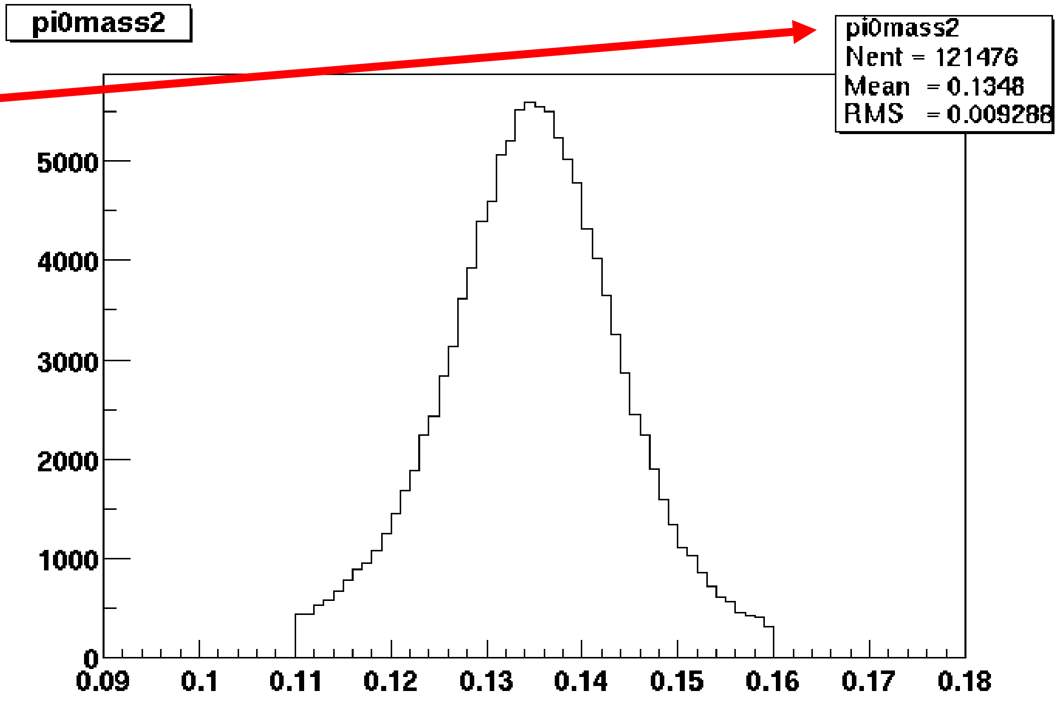
Example use of a splitting up a TCanvas into 4 pads

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

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

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

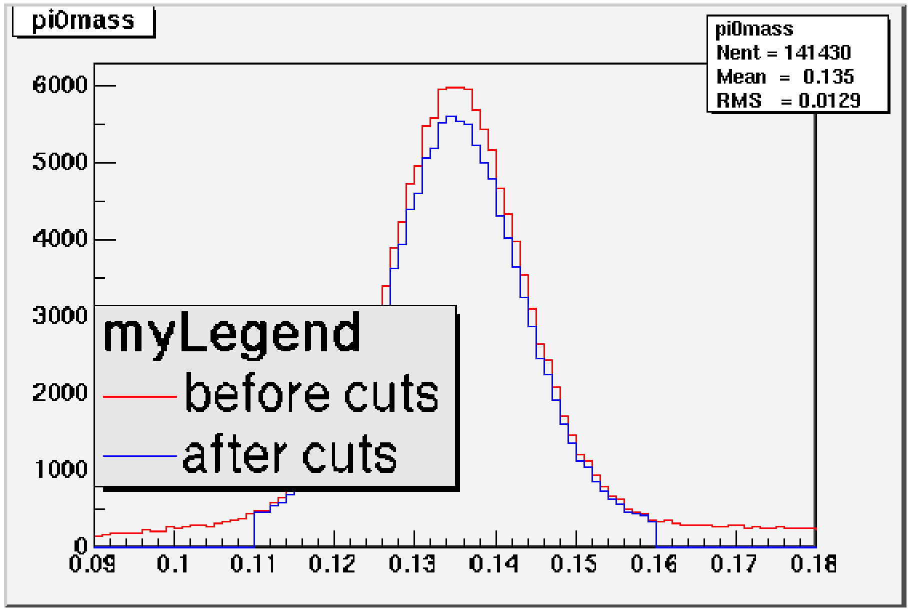
```
gStyle->SetOptStat();          //default setting
gStyle->SetOptStat(0);         //no stats box
h1->Draw();                    //update canvas
gStyle->SetOptStat(1111111);   //turn all options on
h1->Draw();
gStyle->SetOptStat(11);        //name & #events only
h1->Draw();
```

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:

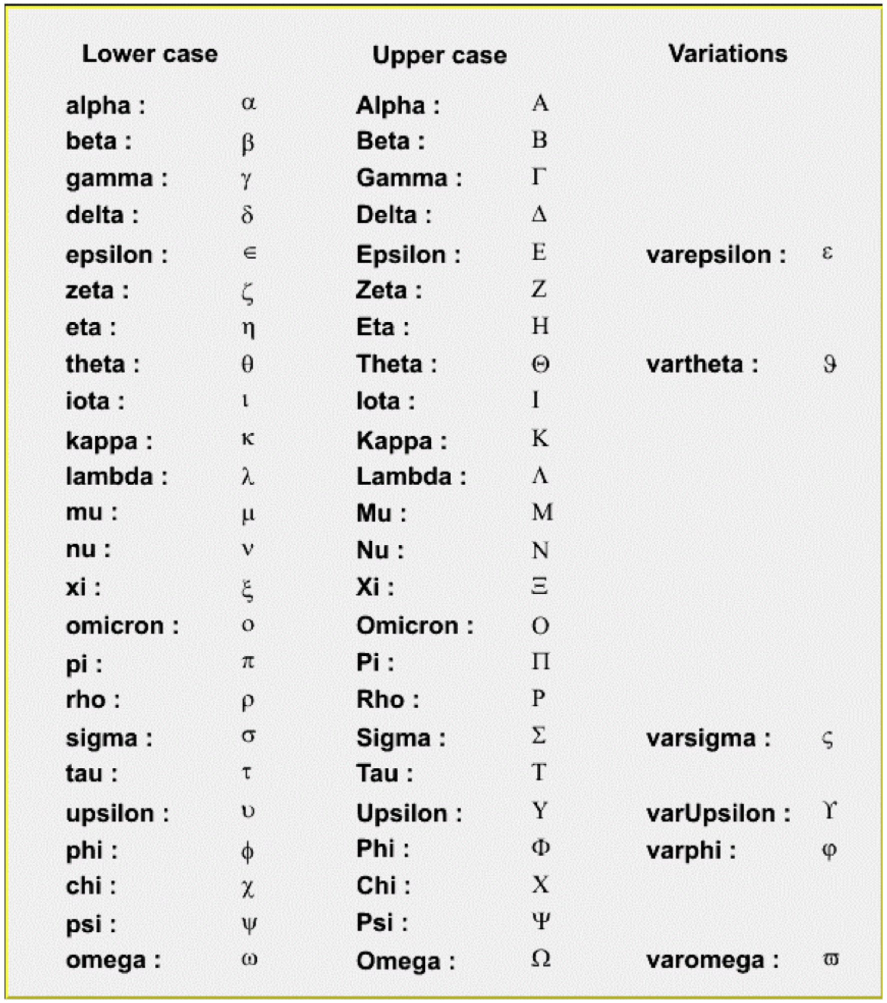
```
h1->SetYTitle("B^{0} #bar{B^{0}}");    // must have brackets
h1->SetTitle("#tau^{+}#tau^{-}");      // to get super/subscript
```

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.

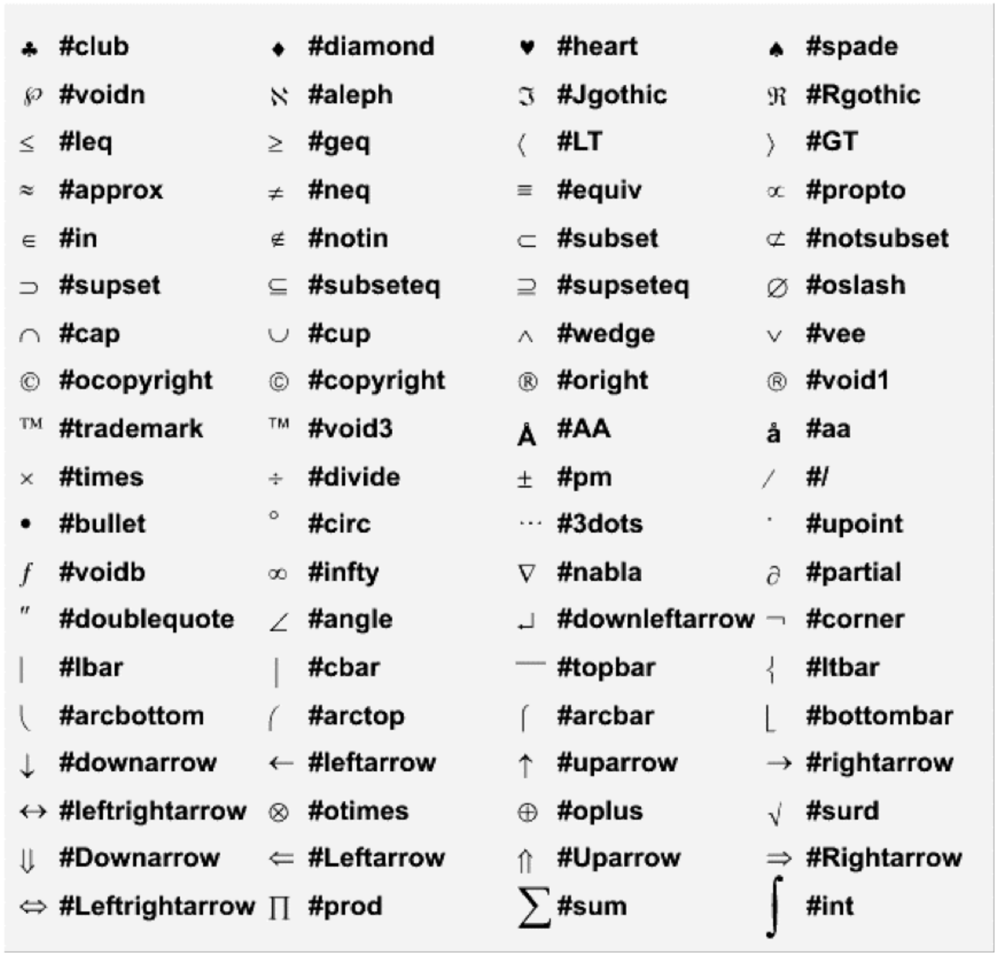
latex		root
\tau	→	#tau
\alpha	→	#alpha

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 :	ϵ	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

Fitting 1D Functions

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

gaus: Gaussian function **$f(x)=p_0\exp\{-½[(x-p_1)/p_2]^2\}$**

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

expo: exponential function $f(x) = p_0\exp(p_1 * x)$

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

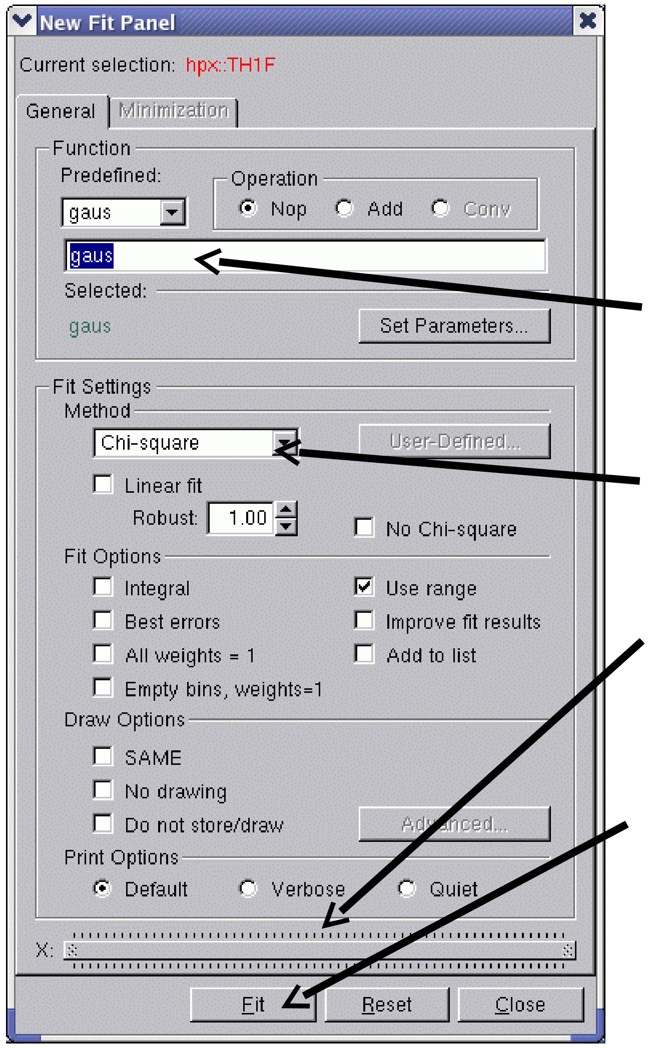
- Fitting with user-defined functions often requires solving a more complicated problem. Save the following as a macro called `myfunc.cc`

```
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 *p` 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);
h1.SetParameters(10, h1.GetMean(), h1.GetRMS());
h1.Fit("f1");
```

Fitting III – The Fit Panel



New Fit Panel

Current selection: hpx::TH1F

General | Minimization

Function

Predefined: gaus

Operation: [✓ Nop] Add Conv

gaus

Selected: gaus

Set Parameters...

Fit Settings

Method: Chi-square

User-Defined...

Linear fit

Robust: 1.00

No Chi-square

Fit Options

Integral

Best errors

All weights = 1

Empty bins, weights=1

[✓ Use range]

Improve fit results

Add to list

Draw Options

SAME

No drawing

Do not store/draw

Advanced...

Print Options: [✓ Default] Verbose Quiet

X:

Fit | Reset | Close

- 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 (now bundled with ROOT) called RooFit. This is a fitting package that is written by members of the HEP community to do complicated analyses.
- There are tutorials on the web.
- 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.
- Not addressing this here as it would need another two good hours..

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

Summary

- You’ve now 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
- you will still need to go through all this as there are lots of information here and you need a little time to assimilate them
- if you want to practice, ask us for data and a project and we have plenty to give you.. also to allow you to do something actually interesting :)

back-up slides

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=5.34.00
# path to root install directory. This will depend on your sysadmin
export ROOTSYS=/users/bona/root/$ROOTVER
export PATH=$PATH:$ROOTSYS/bin:$PATH
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
```

- Log into a new terminal to see that your shell now knows about root.

ROOT exercise 0: making sure you can use root

- 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=5.34.00
# path to root install directory. This will depend on your sysadmin
export ROOTSYS=/Users/bevan/root/$ROOTVER

export PATH=$PATH:$ROOTSYS/bin:$PATH
export LD_LIBRARY_PATH=$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")'
```