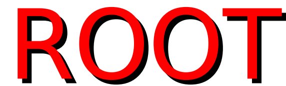
I will assume, in this talk, that everybody knows: what is an histogram, what is an ntuple, basics of C++ (stdin, stdout, file in, file out, if statements, etc etc)..... If you are confused, at any time, by assumptions: rise your hand and I'll stop



ROOT

This talk is (as the presenter) linux based/oriented.
A useful and complete guide about using ROOT under windows can be found in:
http://d0.phys.washington.edu/~haas/windows%20introduction.html

A. Sarti

How to compile/run it

- Download
  - You can get root from http://root.cern.ch/twiki/bin/view/ROOT/Download
- Configure/make
  - You should untar the root-XXX.tgz file, go inside the root dir, issue the command *./configure* and then the commands *make* and *make install*. **To enable specific libraries building** (mathmore, mathcore, roofit) you should use the --enable-yyy option when using ./configure command [*E.g. ./configure --enable-roofit*]
- Setting up variables
  - ROOTSYS : should point to your root installation
  - LD_LIBRARY_PATH : should point to $ROOTSYS/lib + any other lib that you'll need (more on that later, when discussing makefiles)
  - PATH : should point to $ROOTSYS/bin [use '*set path=($path $ROOTSYS/bin)*' command on tcsh]
- Running ROOT
  - *root* (start the interactive session with graphics enabled)
  - *root -b* (start root without graphics) or *root -l* (without graphics box welcome)

Useful files

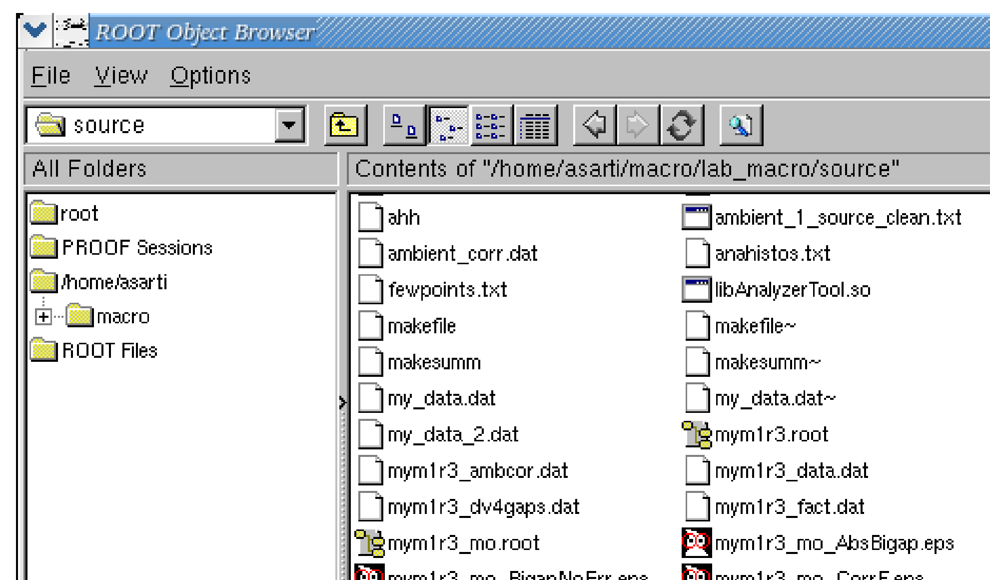
- There are some directories or files you should know about!
  - $ROOTSYS/etc/system.rootrc : contains the default values that customize your root session. E.g.:
    - # Default statistics parameters names.
    - Hist.Stats.Entries   Entries
    - Hist.Stats.Mean   Mean
  - $home/.rootrc : This file can be used to override default values to customize your root session. Default values can be displayed using the *gEnv→Print()* command.
  - $home/.root_hist : contains all the commands executed in the past sessions (that you can recall in your interactive session with the up and down arrows)
  - $ROOTSYS/tutorials : this directory contains almost everything you'd like to do with ROOT. You can search it to find your favourite use case and learn about graphics, fitting, functions, etc etc

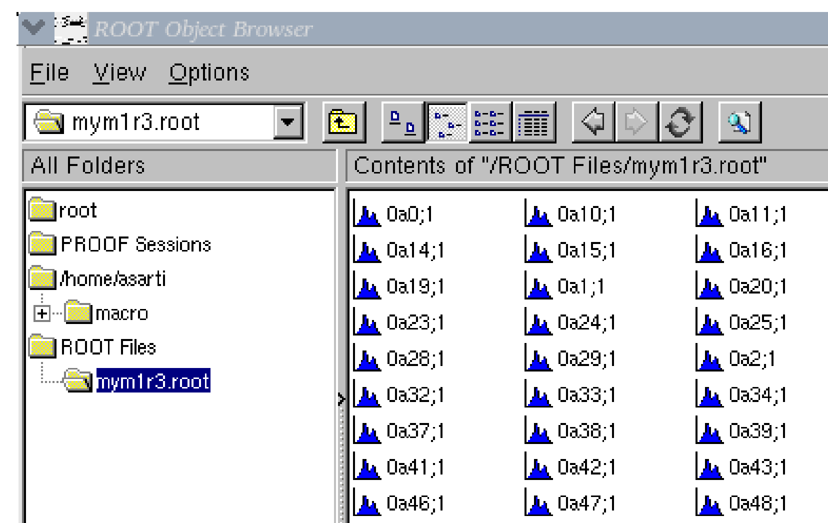
Interactive session

- Type *root* (*.q* to quit)
- You can:
  - Start a GUI (browser) to display files : *TBrowser b;*
  - Execute a macro : *.x nomemacro*
  - Load a macro : *.L nomemacro*
  - Execute a shell command : *.! ls*
  - Excute c++ commands:
    - *for(int i=0; i<10; i++) {cout<<sqrt(pow(2,i))<<endl;}*
  - Use root objects, libraries
    - TTree * ntp = new TTree();
  - Use top,down arrows to navigate command history
  - Use tab key to 'complete' commands
- You also have access to a lot of pointers to default objects (style, system, root session, pad, directory..): gStyle, gSystem, gROOT, gPad, gDirectory....
  - *gDirectory->ls(); //list the content of current directory*

Interactive analysis

- Typical user case:
  - Browsing a root file:
    - In your interactive session start the GUI typing *TBrowser b;*
    - Then you browse your filesystem and find the root file you want to open. You double click on it and then you can find it in the folder 'ROOT files'.
    - You go to the folder ROOT files and you play with the GUI: double clicking on the various items will open/browse them. You can than use the mouse to edit, fit, save the plots.....
  - Displaying histograms and ntuple information from command lines:
    - Following slides





Histogram displaying (inter.)

LHCb

- To display an histogram trough command lines you should:
  - Load the root file: *TFile *f = new TFile(“myrootfile.root”);*
  - Display its content using gDirectory: *f->ls();* or *gDirectory->ls();* give the same result in the MAIN directory. gDirectory is a pointer to the current directory (changes if you change dir).
  - Navigate the file until you reach the desired directory: *f->cd(“mydir”);*
  - Get the pointer to the histogram using Get() method: *TH1D * myh = ((TH1D*)gDirectory->Get(“histoname”));* To use this method you need to know the name of the histogram you want to display!
  - Draw the histogram: *myh ->Draw(“options”);*
    - More common graphic options are: c (draws a smooth curve); l (draws a line); p (draws a marker that can be selected using *myh->SetMarkerStyle (stylenum);)*; same (draws the histogram on top of preceeding one)
  - At this point you can use the pointer to the histogram to make every action you want: fit it, change its axis-color-legend, display statistical information, save it into a plot..... [more on that later]

Ntuple Analysis (inter.)

LHCb

- To analyze an ntuple via command lines you should:
  - Open the root file and go to the directory in which the ntuple is kept
  - Get access to the ntuple pointer using Get method: *TTree *ntp = ((TTree*)gDirectory->Get("ntpname"));*
  - Use the TTree methods to Draw, Print, Scan the ntuple:
    - *ntp->Draw("nomevar"); //draw the variable content in the current canvas*
    - *ntp->Scan("nomevar"); //displays the variable content*
    - *ntp->Print(); //Displays the content of the ntuple*
  - Draw method has many capabilities:
    - Draw one var vs the other (up to 3 var): *ntp->Draw("var1:var2");*
    - Apply cuts: *ntp->Draw("var1","var2>0");* See also the TCut class documentation to learn about Drawing with cuts.
    - Draw the variable information directly into an histograms: *ntp->Draw("var1>>histo")* [more on that later]
  - Graphics options can be supplied using syntax: *ntp->Draw("nomevar", "cuts", "graph options");*
    - e.g. *ntp->Draw("var1","","l");*
    - *ntp->Draw("var1:var2","","box");*

Ntuple Analysis II (inter.)

- In order to retrieve the histogram used to display the variable content of an ntuple it is possible to use many ways:
  - The _ntp->Draw("var1");_ command creates a TH1F object name htemp in the current directory. Access to it can be gained by using the command: _TH1F *htemp = (TH1F*)gPad->GetPrimitive("htemp");_ Once you have got the pointer you can do anything you wanto to the histogram (ask for the entries, change labels, rebin it, etc etc…)
  - The _ntp->Draw("var1:var2")_ command created a TGraph object that can be accessed in the same way as before: _TGraph *graph = (TGraph*)gPad->GetPrimitive("Graph");_
  - The variable can be drawn inside an histo of any name by using: _myNtp->Draw("[varname]>>myhisto");_ The pointer to myhisto can be retrieved using: _TH1F *myhisto = (TH1F*)gDirectory->Get("myhisto");_
  - The variable can be drawn inside a specified histogram that has fixed binning and range using: _myNtp->Draw("[varname]>>histo(500,10,20)");_

ROOT macros.

- What is a ROOT macro:
  - A plain text file (named with .C exetnsion) that contains ROOT/C++ commands that are intepreted and executed subsequently [ex. in the following slides]
- Why/When use ROOT macros:
  - Macros can be used when the root job foresee lots of sequential commands (like opening a file, find a TTree, plot some quantities, customize the graphics, etc. etc.) or when you need to perform some complex analysis (loop on ntuple events, etc etc).
  - Macros are also useful when you need to redo the same analysis @ different times with small changes (producing plots for a paper is a good example!)
- How to use ROOT macros:
  - From the root session: *.x nomemacro.C(opts)* or *.L nomemacro.C* followed by *nomemacro(opts)*.
  - From the command prompt (outside root): *root -b -q .x nomemacro.C\(opts\)*
- ROOT macros should not be abused! Intepreter is not as safe as a compiler/debugger! For large/complex analyses I strongly recommend to go for an executable!

Histogram analysis (macro)

- First example of ROOT macro: how to (dis)play with an histogram:
  - Create a file named: DisplayHisto.C
  - Define the void DisplayHisto method (method name should match the macro name) and insert inside the ROOT command you want to execute
  - Run the macro using the command *root -b -q DisplayHisto.C*
  - Run the macro using the root command, followed by *.x DisplayHisto.C*
  - Run the macro using the root command followed by *.L DisplayHisto.C* and by *DisplayHisto();*
  - An example is given here:

```
void DisplayHisto() {

  //Create a new file
  TFile *file = new TFile("mym1r3_mo.root");
  file->cd();
  file->ls();
  TH1D *myh = ((TH1D*)gDirectory->Get("hSpr"));
  myh->SetLineColor(2);
  myh->SetTitle("My plot");
  myh->SetXTitle("X title");
  myh->SetYTitle("Y title");
  myh->Draw("l");
  c1->Print("myDumb.eps");
  return;
}
```

- Histogram creation
  - The command *new TH1D("name","title",xbins,xmin, max)* can be used to create histos with a given name, title, range and binning.

Ntuple analysis (macro)

- First example of ROOT macro: how to (dis)play with an ntuple:
  - Create a file named: DisplayNtp.C and follow the same steps as before.
  - An example is given here:
    - File is open and read
    - Ntuple is acquired using Get
    - The ntple is Printed
    - p_lab1 variable is drawn in default canvas (c1), Pad (gPad) and histo (htemp)
    - p_lab1 is drawn inside myhisto histogram (retrieved with Get)
    - p_lab2 is drawn in default Pad. Is retrieved with GetPrimitive()
    - p_lab1 is drawn vs p_lab2, requiring p_lab1 > 0 and using the 'box' 2D style

```
void DisplayNtp() {

  //Create a new file
  TFile *file = new TFile("Bd2KPi.root");
  file->cd();
  file->ls();
  TTree *myNtp = ((TTree*)gDirectory->Get("B2hh"));
  myNtp->Print();
  //Draw and print the momentum of particle 1
  myNtp->Draw("p_lab1"); c1->Print("plab1.eps");
  //Draw the momentum of particle 1 in myhisto histogram
  myNtp->Draw("p_lab1>>myhisto");
  TH1F *myhisto = ((TH1F*)gDirectory->Get("myhisto"));
  myhisto->SetMarkerStyle(21);
  myhisto->SetMarkerColor(2);
  myhisto->Draw("p");
  c1->Print("plab1_2.eps");
  myNtp->Draw("p_lab2");
  TH1F *myhist2 = ((TH1F*)gPad->GetPrimitive("htemp"));
  myhist2->SetLineColor(4);
  myhist2->Draw("c");
  c1->Print("plab1_3.eps");
  myNtp->Draw("p_lab1:p_lab2","p_lab1>0","box");
  c1->Print("plab1_lab2.eps");
  return;
}
```

This won't compile! c1 declaration is needed!

Looping on ntuples

- ROOT makes Looping on Ntuples really easy by using: makeClass method. The steps to be followed are:
  - start a root interactive session
  - Load the tree you want to analyze
  - call the *tree->MakeClass("classname");* method. This method will create 2 files: classname.C and classname.h
    - classname.C contains the Loop() method where to put your analysis code
    - classname.h contains all the variables and helper methods needed to read and loop onto a given ntuple
  - Add your code to Loop() method in classname.C
  - Load and run the macro:
    - *.L classname.C*
    - *classname d;*
    - *d.Loop();*

```
#include <TCanvas.h>

void ntpAna::Loop()
{
//   In a ROOT session, you can do:
//      Root > .L ntpAna.C
//      Root > ntpAna t
//      Root > t.GetEntry(12); // Fill t data members with entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//

//     This is the loop skeleton where:
//    jentry is the global entry number in the chain
//    ientry is the entry number in the current Tree
//  Note that the argument to GetEntry must be:
//    jentry for TChain::GetEntry
//    ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//    fChain->SetBranchStatus("*",0);  // disable all branches
//    fChain->SetBranchStatus("branchname",1);  // activate branchname
// METHOD2: replace line
//    fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch
   if (fChain == 0) return;

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      // if (Cut(ientry) < 0) continue;
      std::cout<<p_lab1[0]<<std::endl;
   }
}
```

Customizing loops

- Code that should run BEFORE the loop on each event:
  - Histos and ntuple initialization
- Code that should run for each event (access ntuple variables)
  - Analysis code
- Code that should run at the end of the analysis process
  - Root file writing, histo displaying, etc etc
- NO variable definition is needed (everything is already done for you in .h file!)

```
#include <TCanvas.h>

void ntpAna::Loop()
{
//   In a ROOT session, you can do:
//      Root > .L ntpAna.C
//      Root > ntpAna t
//      Root > t.GetEntry(12); // Fill t data members with entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//

//     This is the loop skeleton where:
//    jentry is the global entry number in the chain
//    ientry is the entry number in the current Tree
//  Note that the argument to GetEntry must be:
//    jentry for TChain::GetEntry
//    ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//    fChain->SetBranchStatus("*",0);  // disable all branches
//    fChain->SetBranchStatus("branchname",1);  // activate branchname
// METHOD2: replace line
//    fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch
   if (fChain == 0) return;

   Long64_t nentries = fChain->GetEntriesFast();

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      // if (Cut(ientry) < 0) continue;
      std::cout<<p_lab1[0]<<std::endl;
   }
}
```

Functions in ROOT

- Functions in root are handled trough class TF1.
- A very nice documentation can be found in the ROOT page
- Several user cases are covered:
  - A - Expression using variable x and no parameters
  - B - Expression using variable x with parameters
  - C - A general C function with parameters
  - D - A general C++ function object (functor) with parameters
  - E - A member function with parameters of a general C++ class

Example of case C

```
// Macro myfunc.C
Double_t myfunction(Double_t *x, Double_t *par)
{
   Float_t xx =x[0];
   Double_t f = TMath::Abs(par[0]*sin(par[1]*xx)/xx);
   return f;
}
void myfunc()
{
   TF1 *f1 = new TF1("myfunc",myfunction,0,10,2);
   f1->SetParameters(2,1);
   f1->SetParNames("constant","coefficient");
   f1->Draw();
}
void myfit()
{
   TH1F *h1=new TH1F("h1","test",100,0,10);
   h1->FillRandom("myfunc",20000);
   TF1 *f1=gROOT->GetFunction("myfunc");
   f1->SetParameters(800,1);
   h1.Fit("myfunc");
}
```

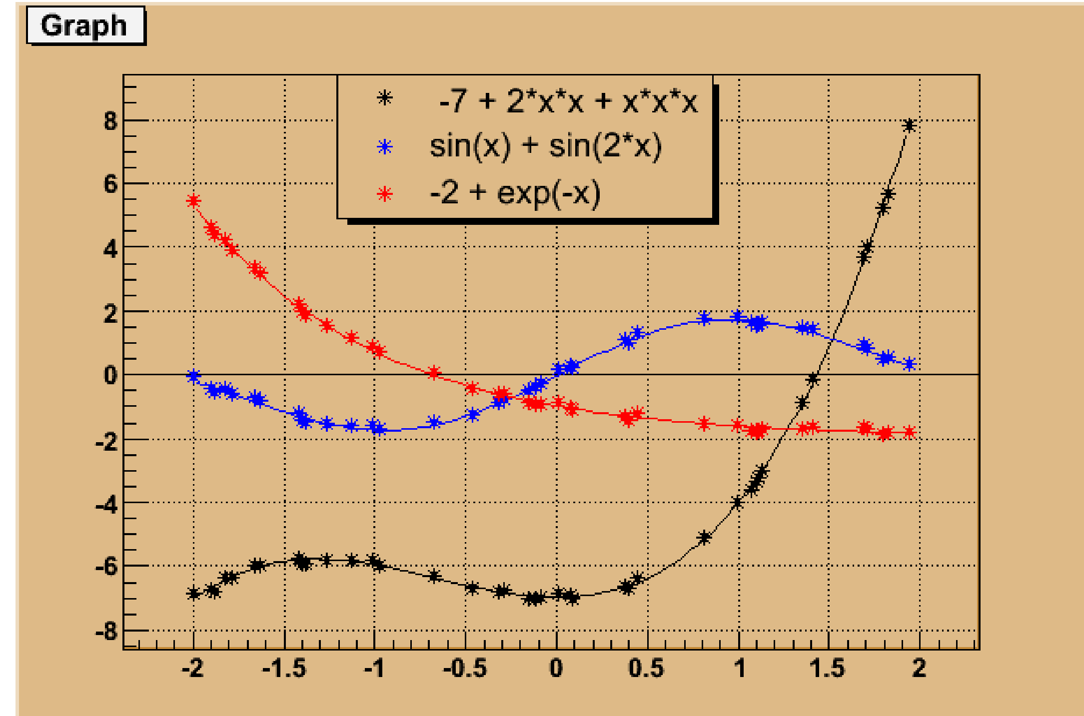
Range, npars

In interactive session:

```
Root > .L myfunc.C
Root > myfunc();
Root > myfit();
```

Simple fitting in root

- .... for the rest you'll have RooFit!
- To fit a Graph/histo in root you need:
  - The fitting function (if 1-D you can use TF1 builder): *TF1 *myf = new TF1("fname", "sin(x) ++ sin(2*x)", -2, 2);* The "++" mean that the linear fitter should be used, and the following formula is equivalent to *"[0]*sin(x) + [1]*sin(2*x)"* (the other way to specify 1-D functions and parameters).
  - You then just ask for *histo->Fit(myf);* Minuit is used to fit.
  - Parameters definition, chisquare, etc etc are all available inside TF1 class
  - You can find everything inside the directory $ROOTYSYS/tutorial/fit
    - fitLinear.C is a very good starting point also to play with graphics!



Graphics Introduction

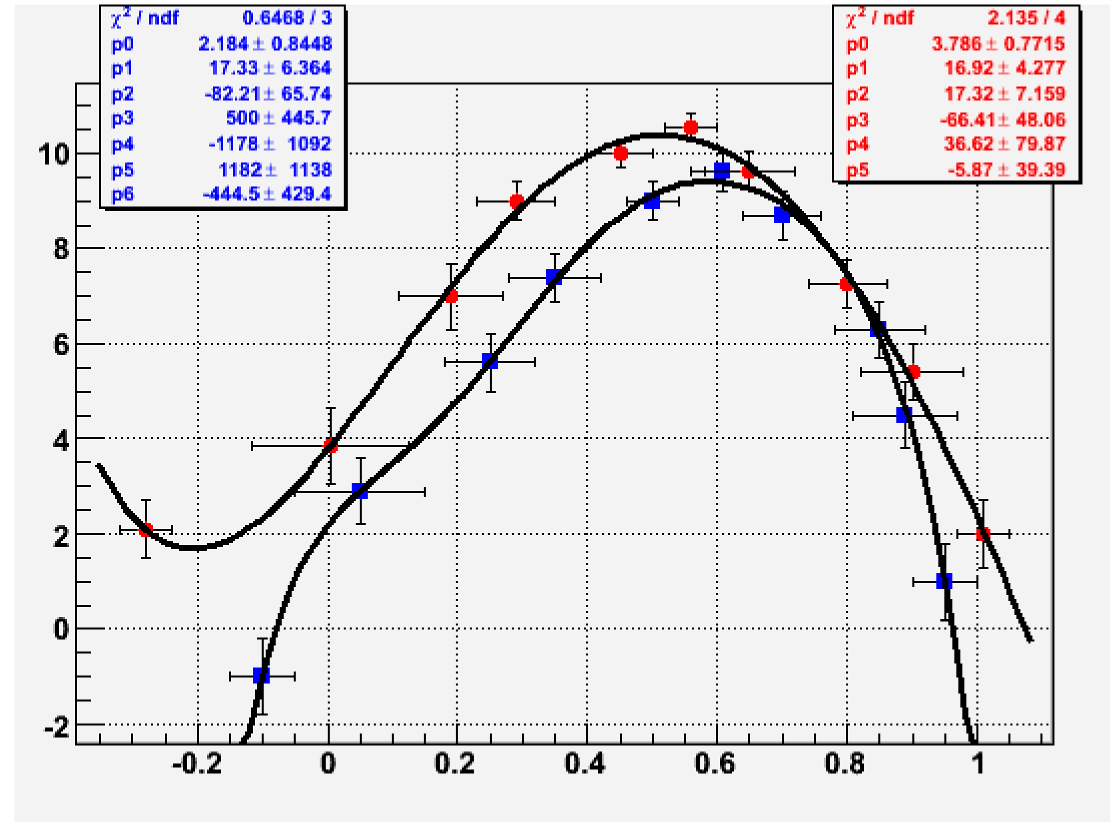
- Basic classes you should know about when dealing with root graphics:
  - TCanvas: create a canvas (graphic box that contains everything and that can be printed/saved in many useful formats eps, png, ...)
  - TPad: graphic object that is drawn on the canvas. The canvas can be subdivided in several Pads in order to plot many histos in 1 canvas.
  - THistPainter: histograms are drawn using this class. You can browse it to view the full capabilities of root
  - TGraph: class that handles graphs. (TGraphErrors or TGraphAsymmErrors do inherit from it)
- Canvases do contain Pads. Pads can be used to retrieve/set most of graphics settings. Histograms also have lots of methods to customize graphics (axis, labels, markers, etc. etc.)
- Remember that you have gPad and gStyle pointers to current pads and style to help you customizing graphics and also to set the statistics output in your plots.

Playing with graphics

- Select style
- Create a canvas (customize it)
- Create and customize Fit function
- Create and customize Graph
- Use gPad to customize the Pad
  - SetTicks, set margins
- Use gStyle to customize stat and fit printout
- Use myframe (canvas DrawFrame() method) to retrieve axis and customize them
- Print a Latex formula on the Pad
- Retrieve and displace the stat box (use Modified!)

```
gROOT->SetStyle("Plain");
TCanvas *c1= new TCanvas("canv","canv",800,900);
TH1F* myFrame = c1->DrawFrame(-0.04,-0.3,0.05,0.2);
gPad->SetTicks(1,1);
TGraphErrors *myG = new TGraphErrors(myVecX,myVecY,myeVecX,myeVecY);
TF1 *f1 = new TF1("f1","[1]+[0]*x",-0.02,0.04);
f1->SetParName(0,"#alpha");
f1->SetParName(1,"#beta");
f1->SetLineColor(1);
gPad->SetBottomMargin(0.14);  gPad->SetLeftMargin(0.21);
gPad->SetRightMargin(0.05);  gPad->SetTopMargin(0.05);
gStyle->SetOptStat(0); gStyle->SetOptFit(0);
gStyle->SetFitFormat("1.3lf");
myG->SetTitle("");
myG->SetMarkerStyle(27);
myG->SetMarkerColor(1);
myG->SetMaximum(0.2);
myG->SetMinimum(-0.3);
myFrame->GetYaxis()->SetLabelOffset(0.02);
myFrame->GetYaxis()->SetTitleOffset(2.4);
myFrame->GetYaxis()->SetTitle("#frac{#Delta G}{G_{0}}  ");
myFrame->GetYaxis()->CenterTitle();
myFrame->GetXaxis()->SetTitleOffset(1.5);
myFrame->GetXaxis()->SetLabelOffset(0.02);
myFrame->GetXaxis()->SetTitle("#frac{#Delta T}{T_{0}} - #frac{#Delta P}{P_{0}}");
myFrame->GetXaxis()->CenterTitle();
//  myG->GetXaxis()->Set(100,-0.04,0.05);
myG->Fit("f1","R");  myG->Draw("P");
TLatex tl; tl.SetNDC(kTRUE);  char line[300];
sprintf(line,"(a)");  tl.SetTextSize(0.08);
tl.DrawLatex(0.28, 0.85, line);
c1->Modified();
TPaveStats *st2 = (TPaveStats*)myG->GetListOfFunctions()->FindObject("stats");
st2->SetX1NDC(0.57);   st2->SetX2NDC(0.92);   st2->SetY1NDC(0.15);
st2->SetY2NDC(0.32);   st2->SetTextColor(2);
c1->Print("FactFig1LErms.eps");
```

Playing with graphics (outcome!)



$ROOTSYS/tutorials/graph/multigraph.C

Ntuple creation

- To create/fill your ntuple you need to:
  - Declare/initialize the variables that are going to be filled in the ntuple
  - Create the ntuple with the command: *TTree * myT = new TTree("data","data");*
  - Add the variables to the ntuple using the Branch() method:
  - Set the variables to the value you want to and then call the Fill() method

```
int f_myInt; double f_myDou;
vector<string> *f_strVct;
vector<double> *f_douVct;

TTree *dataTree = new TTree("data","data"); //Creates ntuple
dataTree->Branch("myInt",   &f_myInt          , "myInt/I");   //Int
dataTree->Branch("myDou",   &f_myDou          , "myDou/D");   //Double
dataTree->Branch("myStrVct", "vector<string>" , &f_strVct,32000,0);
dataTree->Branch("myDouVct" ,"vector<double>" , &f_douVct,32000,0);

//Inside the reading loop
double fact,top,err;  int flag; char buffertC[200]; string str;
ifstream tsC("FinalCorrDatRightErr.dat");
while (tsC.getline(buffertC, 200, '\n')) {
  sscanf(buffertC, "%lf %lf %lf %d",&fact,&err,&top,&flag);
  f_myInt = f_myDou = 0; f_strVct->clear(); f_douVct->clear();
  f_myInt = fact+1;
  f_myDou = err;
  str = "Time over Press: ";
  f_douVct->push_back(flag);
  f_strVct->push_back(str);
  dataTree->Fill();
}
```

Writing ROOT files

- Root files can be created with the following commands:
- To save objects inside the root file you only need to:
  - Create the root file first (when you create a file you are automatically cd'ed into it)
    - TFile *fNewOutFile = new TFile(“root_file.root”, “RECREATE”); In this case, if an existing file with the same name is found the file is overwritten
    - TFile *fNewOutFile = new TFile(“root_file.root”, “UPDATE”); In this case, if an existing file with the same name is found the file is updated with the new information
  - Declare the objects you want to write as 'new':
    - TTree *NtpToBeSaved = new TTree(“data”,”data”);
    - TH1D *HistoToBeSaved = new TH1D(“histo”,”histo”,100,0,100);
  - Call the Write() and Close() method before exiting root:
    - fNewOutFile->Write(); fNewOutFile->Close();
- The result is:

```
root [0] TFile F("root_file.root");
root [1] F.ls()
TFile**         root_file.root
 TFile*         root_file.root
  KEY: TTree    data;1  data
  KEY: TH1D     histo;1 histo
```

Executables creation

LHCb

- What is an executable:
  - Compiled code: you provide, @ compiling time, all the libraries and information to compile your code and then you can run it without calling a root session or using the root executable.
- Executables should be preferred:
  - Anytime! Compiled code is more robust and quick. You have the compiler that helps you spotting problems and you can easily debug the code with gdb in case of core dumps. You can also run valgrind to check for memory corruption and code timing!
  - But for small tasks (producing plots) macros are indeed good enough ;)

main() method creation

- The executable that you're going to compile should have the form:

```
#include <iostream>
int main (int argc, char *argv[]) {
  cout << "Hello World!" << endl;
  return 0;
}
```

  - Where: argc and argv are used to load options from the command line

- A more intresting example:

```
#include "TROOT.h"
#include "iostream.h"
#include "fstream.h"

int main (int argc, char *argv[]) {
  gROOT->SetBatch(kTRUE);
  gROOT->SetStyle("Plain");

  TString fname("dumb");
  Int_t boolean(0); Int_t number(0);
  // -- command line arguments
  for (int i = 0; i < argc; i++){
    if(strcmp(argv[i],"-b") == 0) boolean = 1; // boolean
    if(strcmp(argv[i],"-f") == 0) {fname  = TString(argv[++i]); } //string
    if(strcmp(argv[i],"-n") == 0) {number = atoi(argv[++i]); } //int
  }

  cout<<"Supplied options: "<<boolean<<" "<<number<<" "<<fname.Data()<<endl;
  return 0;
}
```

(simple) makefile creation

- Simple makefile (named mymake)
- Provides the basic ROOT libraries to compile
- Provides two tags:
  - test: that compile the test executable named 'myTest.cc'
  - clean: that erase the library and the executable
- Usage:
  - gmake -f mymake test
  - gmake -f mymake clean

```
ROOTCFLAGS     = $(shell $(ROOTSYS)/bin/root-config --cflags)
ROOTGLIBS      = $(shell $(ROOTSYS)/bin/root-config --glibs)
CXX            = g++
CXXFLAGS       = -g -Wall -fPIC
LD             = g++
LDFLAGS        = -g

NGLIBB          = $(ROOTGLIBS)
GLIBB           = $(filter-out -lNew, $(NGLIBB))

CXXFLAGS       += $(ROOTCFLAGS)

.SUFFIXES: .cc,.cxx

.cc.o:
        $(CXX) $(CXXFLAGS) -c $<

# ==============================================================
test:  myTest.o
# --------------------------------------------------------------
        $(LD) $(LDFLAGS)  -o test myTest.o $(GLIBB)

clean:
        rm -f *.o
        rm -f test
```

compiling / debugging

- How to compile:
  - gmake (or make) followed by the makefile that should be used and the tag (what to compile). E.g.:
    - gmake -f mymakefile myExec
    - gmake clean (assume that a makefile or Makefile is present)
  - READ CAREFULLY ANY ERROR-WARNING MESSAGE!
    - Typically the error message is self explaining IF carefully read
- How to debug:
  - Useful tool is gdb. You can run it by calling gdb and your exec. E.g. gdb ./myExec . After gdb is started you can run your program with the command line options by typing: run (options). E.g. run -outfile myAnalysis
    - When the Exec crashes you can have informations on the reasons by issuing the 'where' command
  - Another useful (advanced) tool is valgrind. Instructions on how to run it can be found on that twiki page: https://twiki.cern.ch/twiki/bin/view/LHCb/CodeAnalysisTools

External classes usage

- Typical (?) use case:
  - want to analyze an ntuple produced by a different executable with advanced algorithms
  - want to develop some (Roo) Fit using “home made” classes
- How to:
  - You create the .cc and .hh files for your classes, you include the .hh in the exec file, you compile the classes libraries first and then you use the library when linking your executable
- Example: analysis of DaVinci ntuple
  - use makeClass on the ntuple you want to analyze to create .cc and .hh files (renaming .C and .h)
  - include the class.h file in the exec
  - create a “class” obejct with the proper builder in the exec and call the class.Loop() method.
  - add to the makefile the making of class library
  - add to the makefile the use of class library when linking your exec
  - compile it and run it!

DaVinci ntuple analysis

- Create the Class to loop on the ntuple:
  - *root*
  - *TFile f("myntpfile.root");*
  - *((TTree*)gDirectory->Get("pathtontp/ntpname")) ->MakeClass ("ntpAna");*
- rename ntpAna.C into .cc [typically .C stands for macro...
- Create the main() method of your exec:
  - *emacs myExec.cc &*
  - add the include of ntpAna.h: *#include "ntpAna.h"*
  - add in your main method the creation of ntpAna object: *ntpAna* myAna= new ntpAna();*
  - Call the myAna.Loop() method in your exec.
- Modify your makefile in order to compile ntpAna first and then myExec.cc:
  - Add the -shared flag to compile the class libraries
  - Add the library tag and variables

```
ROOTCFLAGS    = $(shell $(ROOTSYS)/bin/root-config --cflags)
ROOTGLIBS     = $(shell $(ROOTSYS)/bin/root-config --glibs)
ROOTGLIBS     += $(ROOTSYS)/lib/libHtml.so
CXX           = g++
CXXFLAGS      = -g -Wall -fPIC
LD            = g++
LDFLAGS       = -g
SOFLAGS       = -shared

NGLIBB         = $(ROOTGLIBS)
NGLIBB        += ./libntpAna.so
GLIBB          = $(filter-out -lNew, $(NGLIBB))

NTUPLEB = ntpAna.o ntpAnaDict.o

CXXFLAGS      += $(ROOTCFLAGS)

.SUFFIXES: .cc,.cxx

.cc.o:
	$(CXX) $(CXXFLAGS) -c $<

# ==============================================================
lib:  $(NTUPLEB)
#----------------------------------------------------------------
	$(CXX) $(SOFLAGS) $(NTUPLEB) -o libntpAna.so

ntpAnaDict.cc:  ntpAna.h
	$(ROOTSYS)/bin/rootcint -f ntpAnaDict.cc -c -I../ ntpAna.h



# ==============================================================
test:  myTest.o lib
# --------------------------------------------------------------
	$(LD) $(LDFLAGS)  -o test myTest.o $(GLIBB)

clean:
	rm -f *.o
	rm -f *.so
	rm -f *Dict*
	rm -f test
```