Table of Contents

Start Using ROOT

Welcome to the first hands-on session. We will focus on introductory exercises for getting started working with the ROOT environment and writing our first ROOT macro. Two levels of help will be provided: a hint and a full solution. Try not to jump to the solution even if you experience some frustration. The help is organised as follow:

Hint Hide
Here the hint is shown.

Solution Hide
Here the solution is shown.

Some points linked to the exercises are optional and marked with a  icon: they are useful to scrutinise in more detail some aspects. Try to solve them if you have the time.

Before starting the exercise 1, follow what is written in the lecture

Start using the ROOT prompt

First of all open a Visual Studio Command window on your computer and type

```
root
```

If this does not work, then it means ROOT is not properly installed in your system. Stop here and ask for help to fix the installation. If it works then start playing with the prompt. Use as a calculator and type:

```
root [0] 2+2
```

or whatever you like (see for example the lecture slide) or page 5 of the booklet ("A ROOT Guide for Beginners"). Note that after having issue a statement from the ROOT prompt, you can omit the `;` required by C++. This will make ROOT printing the returned value, if there is one. For example:

```
root [0] TMath::Pi();
```

will not print anything, while

```
root [0] TMath::Pi()
(Double_t)3.14159265358979312e+00
```

Afterwards start writing your first ROOT macro. A ROOT macro is a file containing some C++ code which can be run from the ROOT prompt. You need to define a function and in the function scope you write the code. For example, you create a file, which you will call `mymacro.C`. Inside the file you define a function `mymacro(int value)` and you write the code of slide 8 of the lecture (or something else, if you prefer). The you can run the macro from the ROOT prompt by typing:

```
root [0] .x mymacro.C(42)
```

Note that if the function name is different than the macro (file) name, you need two steps to run. First you load the macro:

```
root [0] .L mymacro.C
```

Then you run the function in the macro. Let's assume the function name is `test(int value)`:

```
root [1] test(42)
```

After having followed this introduction, you can try to solve the first exercise.

Exercise 1: Computing PI

The aim of this exercise is to write a ROOT macro which can compute the number PI. You can deduce the number PI from the area of a circle with unitary radius. You can compute the area of the first quadrant of the circle by generating random points (x,y) , which are uniform distributed in the interval [0,1]. The function to generate uniform random number in the interval [0,1] is

```
root [0] gRandom->Rndm()
```

If you throw for example one million random points, which value of PI do you obtain ? How is it different from the real value of PI, which you can get in ROOT by using `TMath::Pi()` ?

Hint Hide
To solve the exercise you would need:

- make a loop on the `n` number of points you want to generate
- inside the loop you throw two random numbers, uniformly distributed between [0,1], that we can call `x` and `y`
- compute the radius of the point (x,y) : `r = x^2 + y^2`
- if `r < 1`= accept the point and increase a counter
- if `r > 1` do nothing and continue
- the ratio of the number of accepted points to n, will then be equal to the area of the first quadrant, which is PI/4

This we can do in C++ by making a `for` loop and we call the function to generate the random x and y numbers:

```
for (int i = 0; i < n; ++i) {
    double  x = gRandom->Rndm();
    double y = gRandom->Rndm();

    // count the number of accepted points (x^2 + y^2 <= 1)

}
```

After having written and run the macro, try to compile it using ACLIC. For compiling the macro, you need to add to include the required header files, thus you need to add at the beginning of the code:

```
#include "TRandom.h"       // for using  gRandom
#include "TMath.h"            // for using TMath
```

Then, you can compile with ACLIC, by adding a `+` at the end

```
root[0] .x computePI.C+
```

Note the difference in time by using ACLIC, by generating for example more points (e.g. 10E8).

Solution Hide

```
#include "TRandom.h"
#include "TStopwatch.h"
#include "TMath.h"
#include <iostream>
```

```
void computePI(int n = 1000000) {

   int nin = 0;

   TStopwatch w;    // for measuring the time

   for (int i = 0; i < n; ++i) {

      double x = gRandom->Rndm();
      double y = gRandom->Rndm();

      if ( (x*x + y*y) < 1.)  nin++;


   }

   double pi4 = nin/double(n);

   std::cout << "Computed  PI is " << pi4*4 << std::endl;
   std::cout << "Real PI  is  " << TMath::Pi() << " difference = " << TMath::Pi()-pi4*4 << std::e

   w.Print();
}
```

Can you measure the time of generating 10E8 points using CINT or ACLIC. How much faster is ACLIC ? You can use the ROOT class `TStopwatch` for measuring the elapsed time. Add at the begining of the macro (before the loop)

```
TStopwatch w;
```

and after the end of the macro

```
w.Print();
```

Exercise 2: Plotting a Function in ROOT

Following slide 13 of the lecture or page 6 of the Introductory Guide, create a TF1 class using the `sin(x)/x` function and draw it.

Create then a function with parameters, `p0 * sin ( p1 * x) /x` and also draw it for different parameter values. You can try to change the parameter values using `TF1::SetParameters` or by using the ROOT GUI. Try also to change the style of the line and its color. You can either use the ROOT GUI and/or use the methods of the class TAttLine (see TAttLine reference documentation). Since TF1 derives from TAttLine, it inherits all its functions. Try for example to set the colour of the parametric function to blue.

After having drawn the function, compute for the parameter values (p0=1, p1=2):

- function value for x = 1.
- function derivative for x = 1
- integral of the function between 0 and 3.

Hint Hide
To solve the exercise you need to :

- create a TF1 object using a formula expression. In the case of a parametric functions, the two parameters are defined as [0] and [1]
- call TF1::Draw()

You can also find the available function of the class TF1, by using the Tab key on the ROOT prompt, for example

```
root [0] TF1 * f1 = ....
root [1] f1-><TAB>
```

And you can get the full signature of a method by doing for example:

```
root [1] f1->Derivative(<TAB>
```

Solution Hide

```
#include "TF1.h"

void plotFunction() {

   TF1 * f1 = new TF1("f1","sin(x)/x",0,10);
   f1->Draw();


   TF1 * fp = new TF1("fp","[0]*sin([1]*x)/x",0,10);
   fp->SetParameters(1,2);
   fp->Draw("same");
   fp->SetLineColor(kBlue);

   // to change axis y margins
   // (or invert the order of plotting the functions)
   f1->SetMaximum(2);
   f1->SetMinimum(-2);

   std::cout << "Value of f(x) at x = 1 is       " << fp->Eval(1.) << std::endl;
   std::cout << "Derivative of f(x) at x = 1 is  " << fp->Derivative(1.) << std::endl;
   std::cout << "Integral of f(x) in [0,3] is    " << fp->Integral(0,4)  << std::endl;

}
```

You can try to use a different function, for example the Gamma distribution, defined in `ROOT::Math::gamma_pdf`. Try to make a plot as the one in Wikipedia, see here, where the function is plot for different parameter values. See the here the reference documentation of the gamma distribution in ROOT.

Solution Hide

```
#include "TF1.h"
#include "Math/DistFunc.h"


void plotGamma() {

   // Note that parameter [0] is called alpha in definition of gamma_pdf or kappa in Wikipedia
   // Note that parameter [1] is  theta

   // use range [0,20] as in Wikipedia plot
   TF1 * f1 = new TF1("f","ROOT::Math::gamma_pdf(x,[0],[1])",0,20);

   f1->SetLineColor(kRed);
   f1->SetParameter(0,1);
   f1->SetParameter(1,2);

   // use DrawClone because we will plot many different copies of same object but with different
   // parameter values
   f1->DrawClone();


   // now change parameters and draw at different parameter values
   f1->SetLineColor(kGreen);
```

```
   f1->SetParameter(0,2);
   f1->DrawClone("SAME");

   f1->SetLineColor(kBlue);
   f1->SetParameter(0,3);
   f1->DrawClone("SAME");

   f1->SetLineColor(kCyan);
   f1->SetParameter(0,5);
   f1->SetParameter(1,1);
   f1->DrawClone("SAME");


   f1->SetLineColor(kOrange);
   f1->SetParameter(0,9);
   f1->SetParameter(1,0.5);
   f1->DrawClone("SAME");


}
```

Exercise 3: Plotting Measurement Points in ROOT

We will learn in this exercise how to plot a set of points in ROOT using the TGraph class.

Suppose you have this set of points:

- x = 1; y = 0.4
- x = 2; y = 0.3
- x = 3; y = 0.5
- x = 4; y = 0.7
- x = 5; y = 1.3

Plot these points using the TGraph class. Use as a marker point a black box. Looking at the possible options for drawing the TGraph in TGraphPainter, plot a line connecting the points.

Hint Hide
To solve the exercise you need to :

- create an array for the X points: `double x[] = {1,2...`
- create an array for the y points
- create the TGraph from the X and Y arrays

Alternatively you can create first an empty TGraph object and set the point one by one by calling `TGraph::SetPoint`

Solution Hide

```
void plotGraph() {

   double x[] = {1.,2.,3.,4.,5.};
   double y[] = {0.4,0.3,0.5.,0.7,1.3};

   TGraph * g = new TGraph(5,x,y);

   g->Draw("AP");
   g->SetMarkerStyle(21);

}
```

Make a TGraphError by adding an error of 0.3 in X and 0.2 in Y for every point.

Exercise 4: Making an histogram in ROOT

We will learn in this exercise how to create a one-dimensional histogram in ROOT, how to fill it with data and how to plot it.

Create a one-dimensional histogram with 50 bins between 0 10 and fill it with 10000 gaussian distributed random numbers with mean 5 and sigma 2. Plot the histogram and, looking at the documentation in the THistPainter, show in the statistic box the number of entries, the mean, the RMS, the integral of the histogram, the number of underflows, the number of overflows, the skewness and the kurtosis.

Hint Hide
For generating gaussian random numbers use `gRandom->Gaus(mean, sigma)`

Solution Hide

```
#include "TH1.h"
#include "TRandom.h"
#include "TStyle.h"

void plotHistogram() {

   TH1D * h1 = new TH1D("h1","h1",50,0,10);

   for (int i = 0; i < 10000; ++i) {
      double x = gRandom->Gaus(5,2);
      h1->Fill(x);
   }

   h1->Draw();

   gStyle->SetOptStat(111111110);

}
```

After calling the function `TH1::ResetStats()` , you will see that the statistics (mean, RMS,..) of the histogram is slightly different. Try to understand the reason for this, by trying for example to compute the mean of the histogram yourself.

Solution Hide
The initial statistics is computed using the original (un-binned) data, while after calling ResetStats(), the statistics is computed using the bin contents and centres (binned data).

The following macro, creating, filling and plotting histogram contains an error, which one ?

```
#include "TH1.h"

void testPlotHistogram() {

   TH1D h1("h1","h1",50,-5,5);
   h1.FillRandom("gaus",10000);
   h1.Draw();

}
```

Solution Hide
The histogram objected is deleted at the end of the macro, therefore it is not shown in the plot after exiting the macro. To fix it, either call `TH1::DrawClone` or create the histogram using operator new as in the previous example.

This topic: Main > RootIRMMTutorial2013StartingROOTExercises
Topic revision: r8 - 2013-02-28 - LorenzoMoneta