Lecture 4

Graphs and Histograms
Exercises

① Modify example 25. of the tutorial to display three full periods of a sin-wave (and get rid of the ugly brown background color). Make the marker a full square and change the line color to yellow.
gROOT->Reset();
c1 = new TCanvas("c1","A Simple Graph Example",200,10,700,500);
c1->SetFillColor(42);
c1->SetGridx();
c1->SetGridy();

const Int_t n = 20;
Float_t x[n], y[n];
for (Int_t i=0;i<n;i++) {
    x[i] = i*0.1;
    y[i] = 10*sin(x[i]+0.2);
    printf(" i %i %f %f\n",i,x[i],y[i]);
}
gr = new TGraph(n,x,y);
gr->SetFillColor(19);
gr->SetLineColor(2);
gr->SetLineWidth(4);
gr->SetMarkerColor(4);
gr->SetMarkerStyle(21);
gr->SetTitle("asimple graph");
gr->Draw("ACP");

//Add axis titles.
//A graph is drawn using the services of the TH1F histogram class.
//The histogram is created by TGraph::Paint.
//TGraph::Paint is called by TCanvas::Update. This function is called by default
//when typing <CR> at the keyboard. In a macro, one must force TCanvas::Update.
c1->Update();
c1->GetFrame()->SetFillColor(21);
c1->GetFrame()->SetBorderSize(12);
gr->GetHistogram()->SetXTitle("Xtitle");
gr->GetHistogram()->SetYTitle("Y title");
c1->Modified();
Exercises

① Experiment with the histogram drawing options, starting from example 24. of the tutorials.
Why Should You Use a Tree?

In the Input/Output chapter, we saw how objects can be saved in ROOT files. In case you want to store large quantities of same-class objects, ROOT has designed the TTree and TNtuple classes specifically for that purpose. The TTree class is optimized to reduce disk space and enhance access speed. A TNtuple is a TTree that is limited to only hold floating-point numbers; a TTree on the other hand can hold all kind of data, such as objects or arrays in addition to all the simple types.
What is a Tree?

A Tree is like a large and wide table:

A Tree is an array of ‘entries’ or ‘events’, similar to a row of a table.

Within each entry, there are independent ‘branches’. Each branch can contain an object or sub-branches. This can be compared to a column of a table.

Within each branch, there are ‘leaves’, which hold the member-variables of complicated classes. These are the final values.

There are several Tree-like classes in ROOT:

- **Tree** ‘array’ of TObjects
- **TNtuple** TTree with only Float_t
- **TNtupleD** TTree with only Double_t
- **THbookTree** Direct access to a HBook (Paw) file
- **TChain** collection of files containing TTree objects
Making a Tree Object

A tree has a very simple constructor:

\[
\text{TTree(const char *name, const char *title, Int_t splitlevel = 99)}
\]

The Tree is created in the current directory

Use the various Branch functions to add branches to the Tree.

If the first character of ‘title’ is a "/", the function assumes a folder name. In this case, it creates automatically branches following the folder hierarchy. \text{splitlevel} may be used in this case to control the split level.

Example:

\[
\text{TTree* tree = new TTree("treeName","treeTitle");}
\]
Non-Object Branches

TBranch* Branch(const char *name, void *address, const char *leaflist, Int_t bufsize)

This constructor supports non-objects, e.g. C-style structs, or arrays of variables.

address is a pointer to the beginning of the data
leaflist is the list of variable names, separated by ":". Variable types are separated by "/"

Example:

```
ROOT [0] TTree* mytree = new TTree("mytree","Test Tree");
ROOT [1] Double_t values[5];
```
**Supported Simple Types**

<table>
<thead>
<tr>
<th>Supported Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a character string terminated by the 0 character</td>
</tr>
<tr>
<td>b</td>
<td>an 8 bit unsigned integer (UChar_t)</td>
</tr>
<tr>
<td>s</td>
<td>a 16 bit unsigned integer (UShort_t)</td>
</tr>
<tr>
<td>i</td>
<td>a 32 bit unsigned integer (UInt_t)</td>
</tr>
<tr>
<td>D</td>
<td>a 64 bit floating point (Double_t)</td>
</tr>
<tr>
<td>I</td>
<td>a 64 bit unsigned integer (ULong64_t)</td>
</tr>
<tr>
<td>B</td>
<td>an 8 bit signed integer (Char_t)</td>
</tr>
<tr>
<td>S</td>
<td>a 16 bit signed integer (Short_t)</td>
</tr>
<tr>
<td>I</td>
<td>a 32 bit signed integer (Int_t)</td>
</tr>
<tr>
<td>F</td>
<td>a 32 bit floating point (Float_t)</td>
</tr>
<tr>
<td>L</td>
<td>a 64 bit signed integer (Long64_t)</td>
</tr>
<tr>
<td>O</td>
<td>a boolean (Bool_t)</td>
</tr>
</tbody>
</table>
Object Branches

The constructor:

```c
TBranch* Branch(const char* name, const char* classname, void*** addobj, ...)
```

classname refers to the class of the object you want to store
addobj is the address of the pointer to the object you want to store
(poorly documented)

Example:

```c
ROOT [0] TTree* mytree = new TTree("mytree","Test Tree");
ROOT [1] TH1D* h = new TH1D("h","h",10,0,1);
ROOT [2] TBranch *branch = tree->Branch("hBranch","TH1D",&h);
```
Filling a Tree

Extremely simple:

```cpp
mytree->Fill();
```

This function loops over all the branches of the tree. For each branch, it copies the current values of the leaves to the branch buffer.

Example:

```
ROOT [0] TTree* mytree = new TTree("mytree","Test Tree");
ROOT [1] TH1D* h = new TH1D("h","h",10,0,1);
ROOT [2] Double_t values[5] = {0,0,0,0,0};
ROOT [3] TBranch *b1 = mytree->Branch("b1","TH1D",&h);
ROOT [5] mytree->Fill();
ROOT [6] h->FillRandom("gaus");
ROOT [8] mytree->Fill();
```
Browsing the Tree

A graphical interface to play with a tree is started using:

```cpp
mytree->StartViewer()
```
Tree on the Command Line

The contents of the tree can be drawn from the commandline:

```
Draw(const char *exp, const char *cut, Option_t *option, Long64_t nent, Long64_t fi rst)
```

- **exp**: expression describing what to draw, e.g. "y:x", or "sqrt(x/y*z*z)". For 2-D (or 3-D) plots, expressions are separated by a ".". Convention: z : y : x. Statement "x»histoname" will save to predefined histogram.

- **cut**: expression describing some conditions, e.g. "z>0"

- **option**: drawing option (see histograms)

**Example:**

```c
ROOT [0] TTree* mytree = new TTree("mytree","Test Tree");
ROOT [1] Double_t values[3];
ROOT [2] TBranch *b2 = mytree->Branch("b2",values, "x/D:y:z");
ROOT [3] macroToFillTree();
ROOT [4] mytree->Draw("sqrt(z):x*y","z>0","surf4",1000,10);
```
Tree on the Command Line (Cont’d)

The structure of the Tree can be printed:

```
mytree->Print();
```
**Tree on the Command Line (Cont’d)**

You can get a list of (part of) the contents:

```
mytree->Scan("a:b");
```

```
* Row * a * b *
* 0 * 0 * 0 *
* 1 * 1 * 0 *
```

KVI Root-course, March 8 2005 – Gerco Onderwater, KVI – p.16/21
You can also inspect the contents of a specific entry numerically:

```
mytree->Show(1);
```

```
EVENT: 1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>= 1</td>
</tr>
<tr>
<td>b</td>
<td>= 0</td>
</tr>
<tr>
<td>c</td>
<td>= 0</td>
</tr>
<tr>
<td>d</td>
<td>= -3.14</td>
</tr>
<tr>
<td>e</td>
<td>= 0</td>
</tr>
</tbody>
</table>
```
Reading a Tree from a Macro

Very similar to the filling method:

\[
\text{GetEntry(Long64_t entry = 0, Int_t getall = 0)}
\]

Of course, you have to tell the tree first, where to store the data:

\[
\text{SetBranchAddress(const char* bname, void** add)}
\]

Example:

```c
ROOT [0] TFile* file = new TFile("test.root");
ROOT [1] TTree* mytree = (TTree*)file->Get("mytree");
ROOT [2] TH1D* h = 0;
ROOT [3] mytree->SetBranchAddress("b1", &h);
ROOT [4] mytree->GetEntry(0);
ROOT [5] h->Draw();
ROOT [6] mytree->GetEntry(1);
ROOT [7] h->Draw();
```
Some Very Helpful Tools

For large trees, with many branches and leaves, and complicated objects, reading a tree may become a lot of work. Work is simplified with

\[
\text{MakeClass(const char* classname = "0", Option_t* option)} \\
\text{MakeCode(const char* filename = "0")}
\]

Example:

\[
\begin{align*}
\text{ROOT [0]} & \text{ TTree* mytree = new TTree("mytree","Test Tree");} \\
\text{ROOT [1]} & \text{ Double_t values[3];} \\
\text{ROOT [2]} & \text{ TBranch *b = tree->Branch("val",values, "a/D:b:c");} \\
\text{ROOT [3]} & \text{ mytree->MakeCode("fastCode.cxx");}
\end{align*}
\]
What the generated code looks like

```c
{  
   ... some stuff deleted here ....  
   TTree *mytree = (TTree*)gDirectory->Get("mytree");

   //Declaration of leaves types
   Double_t   b2_a;
   Double_t   b2_b;
   Double_t   b2_c;

   // Set branch addresses.
   mytree->SetBranchAddress("b2", &b2_a);
   mytree->SetBranchAddress("b2", &b2_b);
   mytree->SetBranchAddress("b2", &b2_c);

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

   Long64_t nentries = mytree->GetEntries();

   Int_t nbytes = 0;
   // for (Long64_t i=0; i<nentries; i++) {
   //     nbytes += mytree->GetEntry(i);
   // }
}
```
Now an Ntuple is straightforward!

**TNtuple** is a derived class from a **TTree**. So it can do all the things a **TTree** can. The only **difference** is that it can only contain **Float_t** (**TNtuple**) or **Double_t** (**TNtupleD**) variables. Each variable behaves as a separate branch.

**Constructor:**

```cpp
TNtuple(const char* name, const char* title, const char* varlist, Int_t bufsize = 32000)
```

**Filling:**

- `Fill(Float_t x0, Float_t x1 = 0, Float_t x2 = 0, ..... , Float_t x14 = 0)`
- `Fill(const Float_t* x)`

**Example:**

```cpp
ROOT [0] TNtuple* nt = new TNtuple("ntName","ntTitle","a:b:c");
ROOT [1] nt->Fill(3.1415,2.7182818,1.41421);
ROOT [2] nt->Fill(1/3.1415,1/2.7182818,1/1.41421);
ROOT [3] nt->Draw("cos(a):ln(b):c*c","b>0&&c>0");
```
Exercises

① Download the root-file on the website: http://kvir03.kvi.nl/rootcourse/. Inside you’ll find a tree containing histograms and an array of Double_t’s. Write a macro that draws the histograms of the third entry and prints the values of the array.

② Download the ascii file from the root-course website and convert it into an Ntuple. Make a 2D histogram with the fourth value on the x-axis and the second on the y-axis. Plot it with a smooth surface and label the axes. Send me the postscript file of this plot (onderwater@kvi.nl).