Introduction to ROOT

Prof. Silvia Masciocchi
dr. Federica Sozzi

Heidelberg University

Day 4 – Using Trees

Based on the slides created by dr. Jens Wiechula, Frankfurt University

Outline

- Summary of material from last week
- TTree:
  - Introduction
  - Basic functions
  - Writing data to a tree
    - Filling simple types
    - Filling ROOT objects
  - More on drawing
  - Chains
  - Browsing trees

What we learned last week

We started from Linux and C++ programming basics

- ✔ Compiler / Linker / Interpreter
- ✔ Variables
- ✔ Control structures
- ✔ Loops
- ✔ Functions
- ✔ Classes
- ✔ Dynamic memory

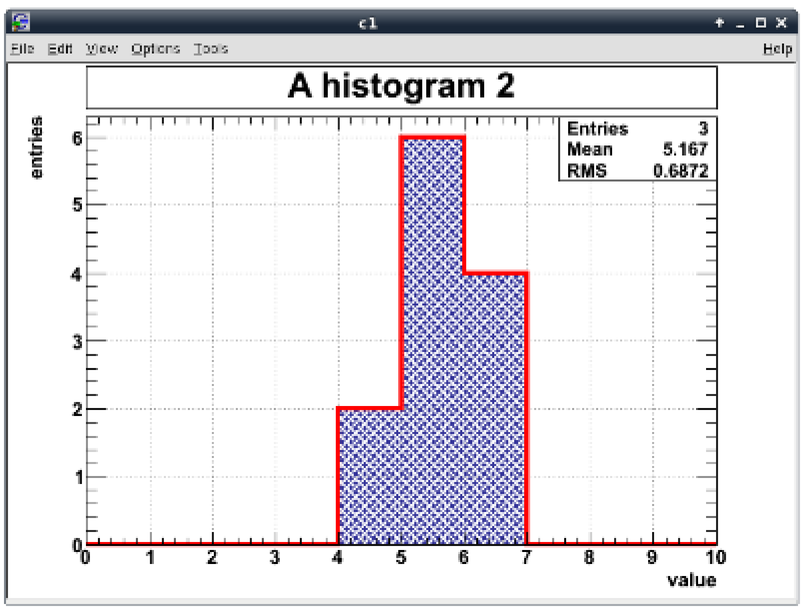
And you already used all these components during each of the exercise parts!

Understanding these concepts is fundamental for programming itself but also to be able to proficiently use ROOT

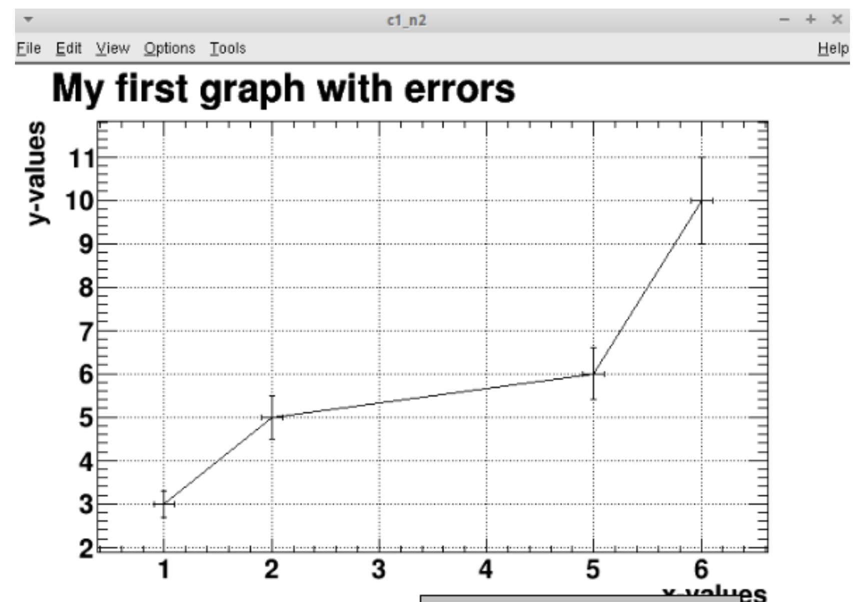
ROOT data visualization

Histograms and TGraphs:

the basis of data visualizations



histo_style.C



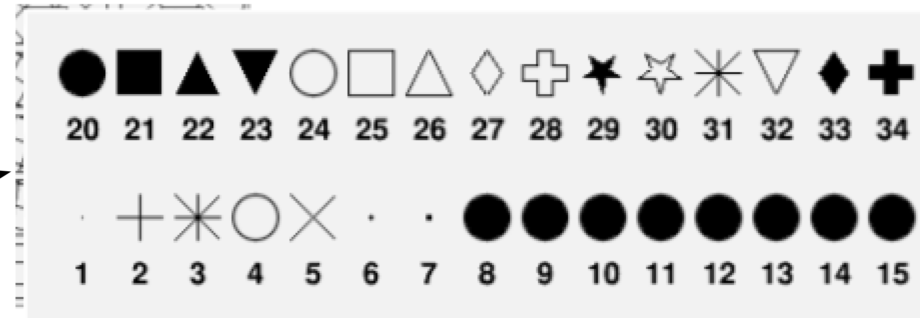
graph_examples.txt

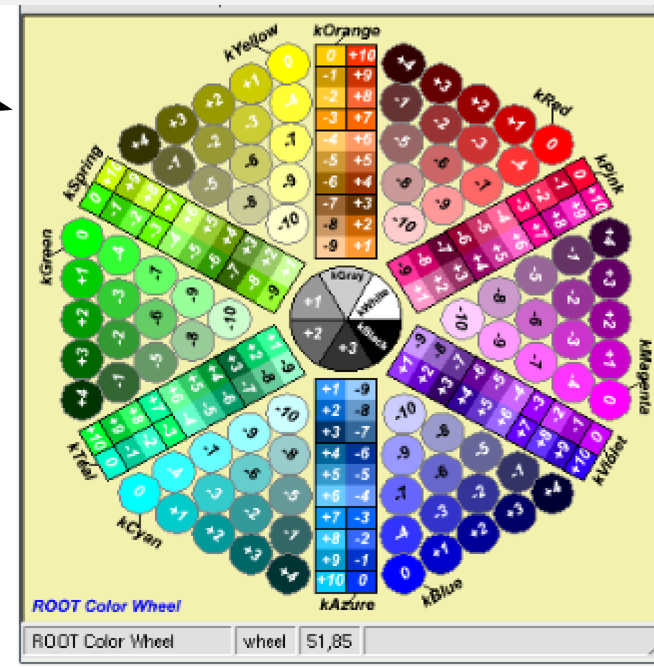
ROOT data visualization

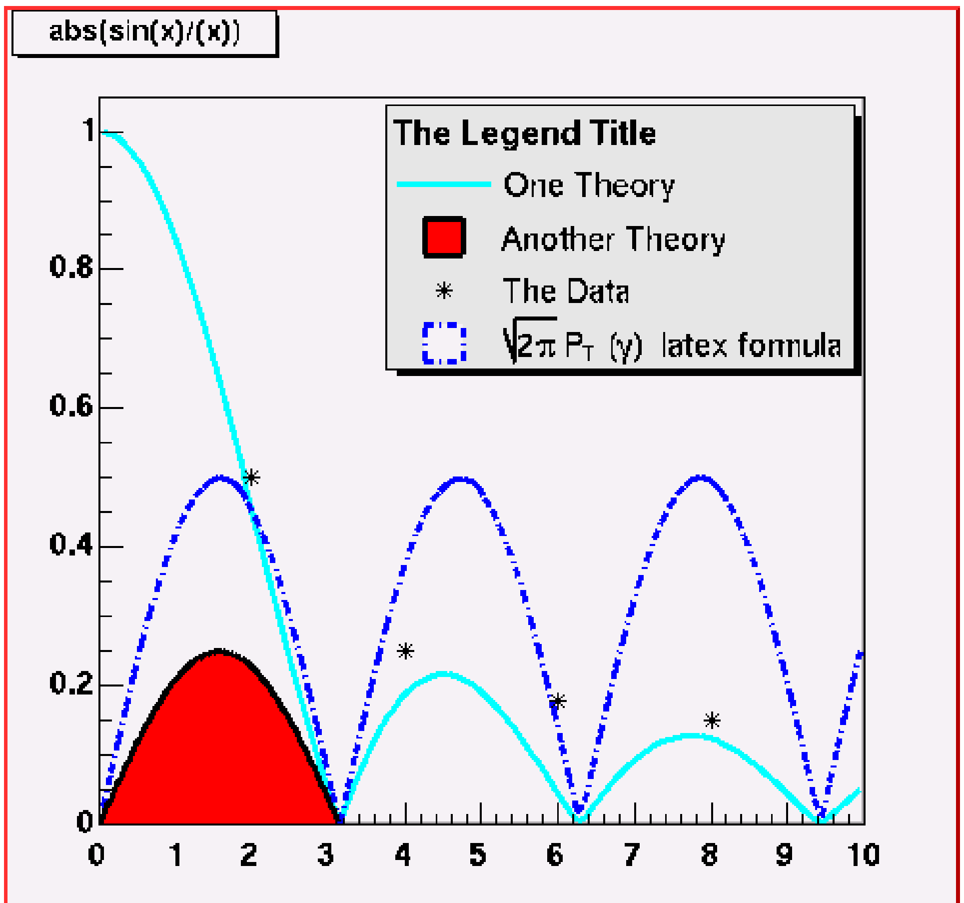
change style, TCanvas and TLegend

Many classes inherit from attribute classes (TAttLine, TAttFill, TAttMarker)

We learned how to use these functions to change the style







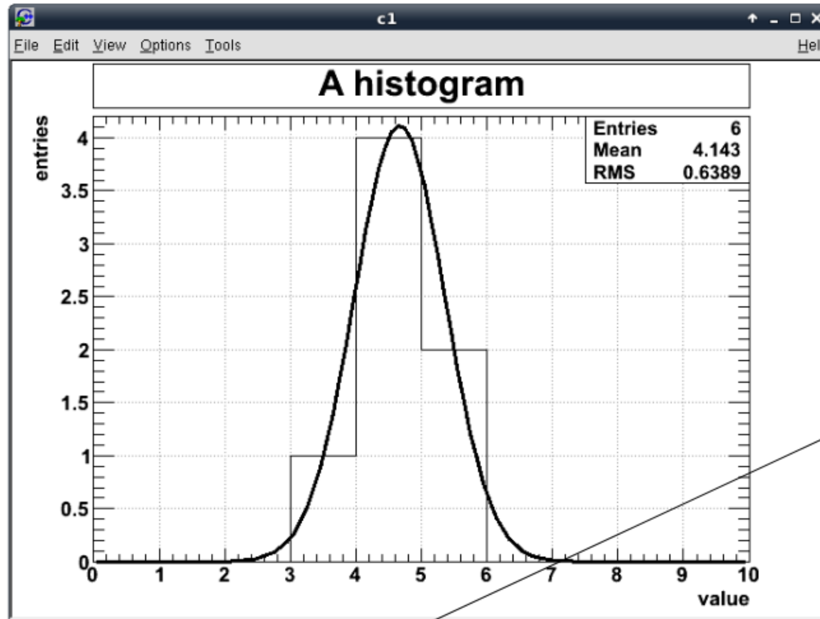
GUI and code

- Many actions in ROOT can be performed via the GUI or via code
- GUI allows fast checks, first exploration of the data, quick changes
- The data analysis is done via code/macros

Functions and fits

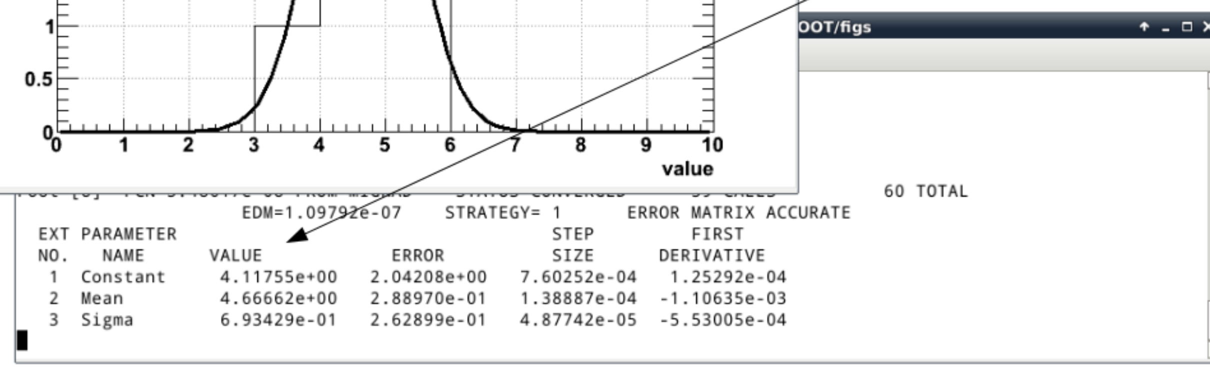
TF1

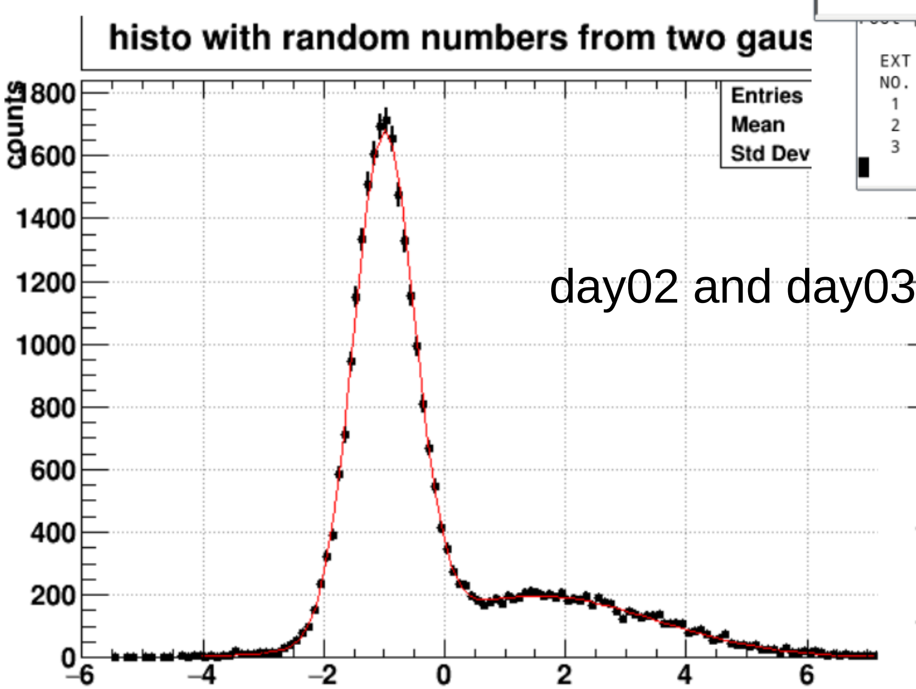
- Simple fits
- Composite fits



day02

Fit results are written in the ROOT shell





day02 and day03

Filling histograms with random number

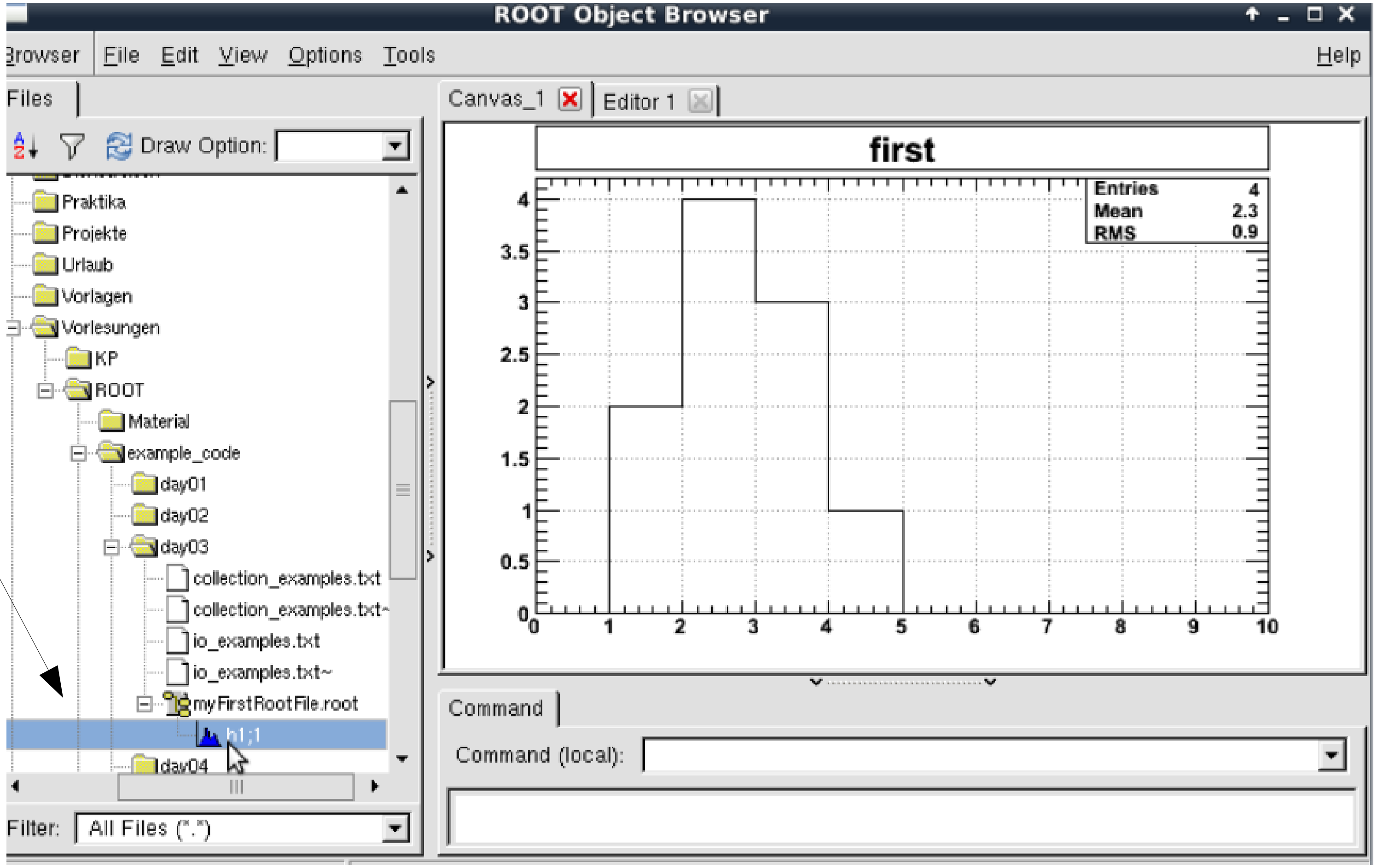
Common in physics, simulate the real experiment with a set of simulated data (pseudo-data), representing a version of the real experiment

“Toy MC” (no complete simulation of the unserlyying physis + detectors, but only simulation of the final distrubution and results)

Manipulating (T)Objects

I/O and grouping

- I/O: ROOT files:
  Any ROOT object can be saved into a ROOT FILE
- Concept of directories in ROOT



- Methods to group connected objects: TCollection
  Any TObject can be saved into a TCollection

- TObject is the mother of many of the classes in ROOT: tomorrow we will return to this concept

TTree

Introduction

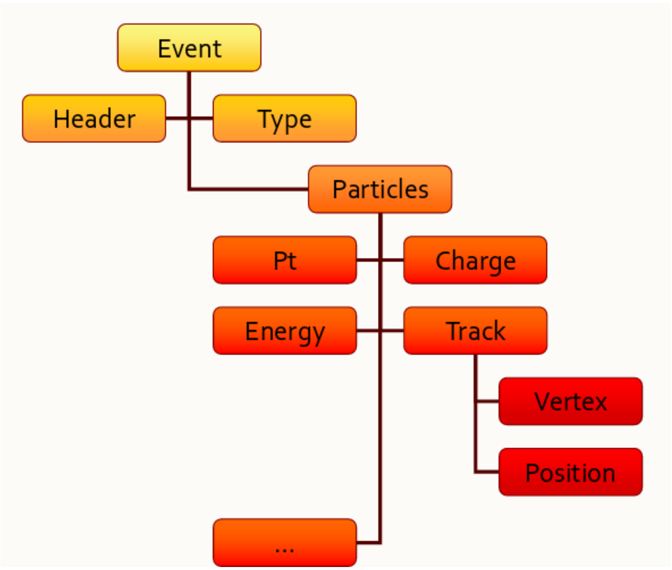
Specific classes to store large amounts of data, visualize, analyse them:

TNtuple and TTree

Variables are stored in table-like structures
Only float are handled

Like TNtuple, but TTree can store any kind of object
Adapt for complex structures, e.g. event with tracks

x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
0.552454	-0.21231	0.350281
-0.18495	1.187305	1.443902
0.205643	-0.77015	0.635417
1.079222	-0.32739	1.271904
-0.27492	-1.72143	3.038899
2.047779	-0.06268	4.197329
-0.45868	-1.44322	2.293266
0.304731	-0.88464	0.875442
-0.71234	-0.22239	0.556881
-0.27187	1.181767	1.470484
0.886202	-0.65411	1.213209
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
[illegible]	[illegible]	3.562347



TTree
Introduction

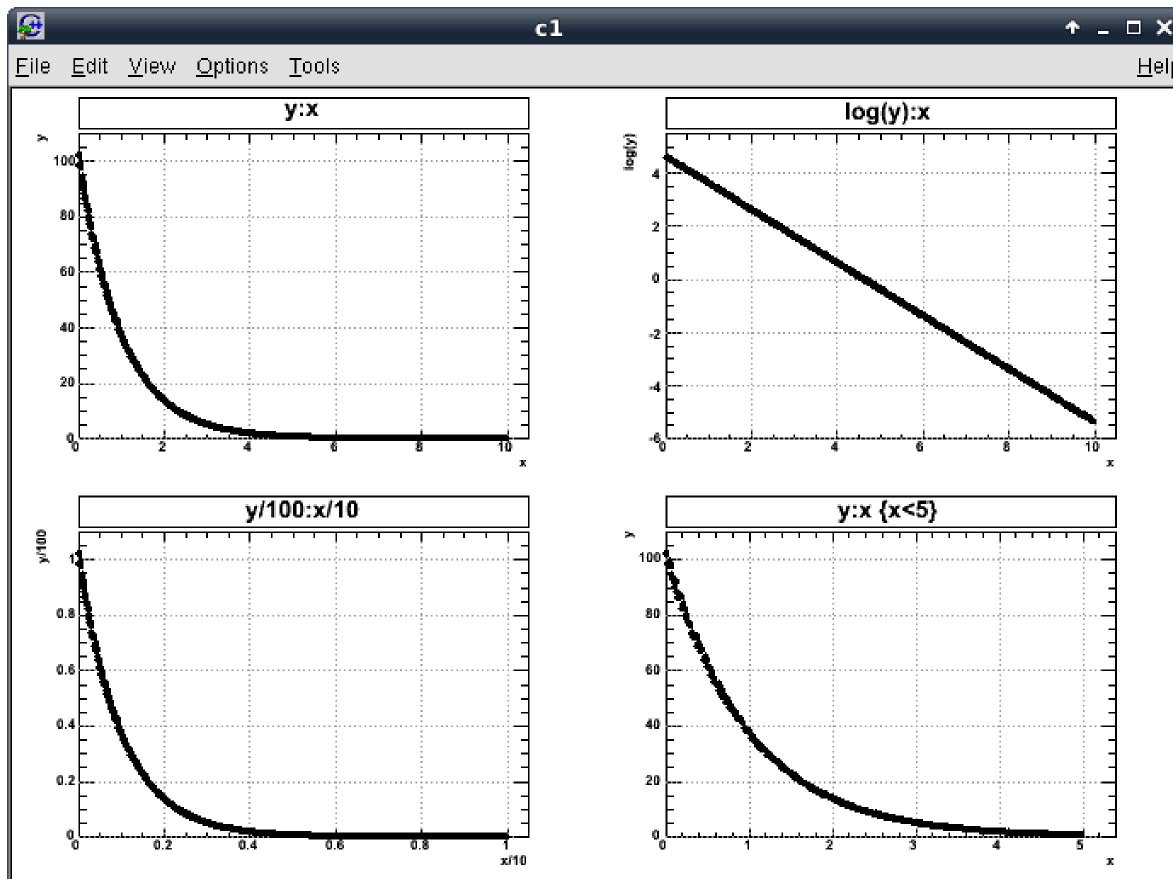
- Trees provide a powerful method for data analysis
- Storing and reading of information is highly optimised
- Allows for parallel processing
- Whole structures of ROOT objects can be stored and retrieved

Basic usage

A first example

Read a white-space separated file with x and y value into a tree and draw something

```
TTree t;
t.ReadFile("data.txt","x:y");

// draw y against x, log(y) against x,
// y/100 against x/10, and y against x for x<5
TCanvas c
c.Divide(2,2);

c.cd(1);
t.Draw("y:x");

c.cd(2);
t.Draw("log(y):x");

c.cd(3);
t.Draw("y/100:x/10");

c.cd(4);
t.Draw("y:x","x<5");
```



tree_examples.txt

Basic usage

Most important functions

All functions will be explained in the following slides

Long64_t	ReadFile(...)
void	Print(...)
Long64_t	Draw(...)
Long64_t	Scan(...)
Long64_t	GetEntries()
Bool_t	SetAlias(...)
Long64_t	GetSelectedRows()
Double_t*	GetV1() / GetV2() / GetV3() / GetV4()
Int_t	SetBranchAddress(...) – several overloaded functions
Int_t	GetEntry(...)
TBranch*	Branch(…) – several overloaded functions

Basic usage
The ReadFile function

Long64_t ReadFile(const char* filename, const char* branchDescriptor = "", char delimiter = ' ')

The function returns the number of read lines

filename Name of the input file

branchDescriptor each column in the file will be stored in a branch. Here the naming of the branches is given, e.g. x:y = first column gets the name 'x' the second the name 'y'.

Optionally the variable can be given a type, e.g. x/D:y means that it will be stored with double precision, the default is floating point. If no individual type is given, the type of the previous branch is used.

Further types: B,S,I,L for char, short, int, long (signed, a small letter for unsigned, e.g. x/i is an unsigned int)

The branch descriptor can also be given in the first line of the file

delimiter allows for the use of another delimiter besides whitespace (default).

If the filename ends with extensions .csv or .CSV and a delimiter is not specified (besides ' '), the delimiter is automatically set to ','.

Basic usage

The Print function

void Print(Option_t* **option** = "")

Print a summary of the tree contents.

If option contains "all" friend trees are also printed.

If option contains "toponly" only the top level branches are printed.

If option contains "clusters" information about the cluster of baskets is printed.

Wildcarding can be used to print only a subset of the branches, e.g.,

T.Print("Elec*") will print all branches with name starting with "Elec".

```
root [16] t.Print()
******************************************************************************
*Tree    :          :                                                        *
*Entries :     1000 : Total =            9477 bytes  File  Size =          0 *
*        :          : Tree compression factor =   1.00                       *
******************************************************************************
*Br    0 :x         : x/F                                                    *
*Entries :     1000 : Total  Size=       4602 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    1 :y         : y/F                                                    *
*Entries :     1000 : Total  Size=       4602 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
```

Basic usage

The Draw function

Long64_t Draw(const char* varexp, const char* selection, Option_t* option = "",
Long64_t nentries = 1000000000, Long64_t firstentry = 0)

The function returns the number of selected lines

varexp a variable expression that can contain the branch names and functions

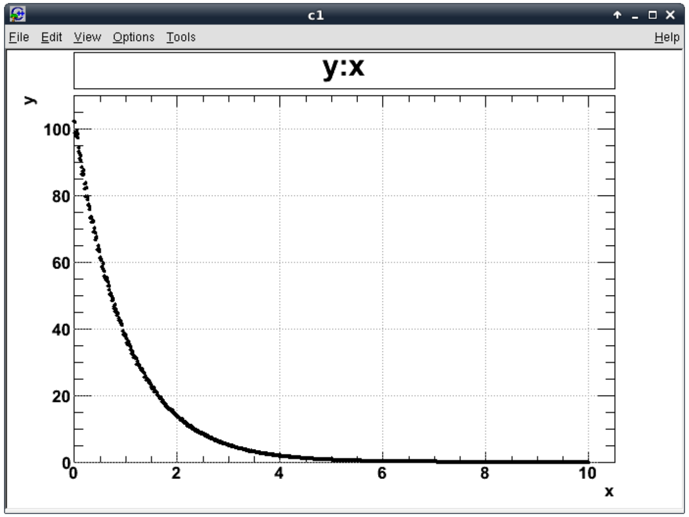
selection selection of the data, a so called 'cut'. This is a logical expression and can use any number of logical expressions connected with && or ||, ...

option a draw option, like 'same, colz, prof'

nentries maximum number of entries to use (number of lines)

firstentry the first entry to use (first line)

```
root [18] t.Draw("y:x")
```



Basic usage
The Scan function

Long64_t Scan(const char* **varexp**, const char* **selection**, Option_t* **option** = "",
Long64_t **nentries** = 1000000000, Long64_t **firstentry** = 0)

The function returns the number of selected lines and prints the selection to stdout

varexp a variable expression that can contain the branch names and functions

selection selection of the data, a so called 'cut'. This is a logical expression and can use any number of logical expressions connected with && or ||, ...

option a draw option, like 'same, colz, prof'

nentries maximum number of entries to use (number of lines)

firstentry the first entry to use (first line)

```
root [22] t.Scan("x:y","x>2&&x<2.03")
************************************
*    Row   *          x *          y *
************************************
*      201 * 2.0099999 * 13.862178 *
*      202 * 2.0199999 * 13.384497 *
*      203 * 2.0299999 * 13.338671 *
************************************
==> 3 selected entries
```

tree_examples.txt

Basic usage

The GetEntries function

Long64_t GetEntries()

The function returns the number of entries in the tree

```
root [23] t.GetEntries()
(const Long64_t)1000
```

tree_examples.txt

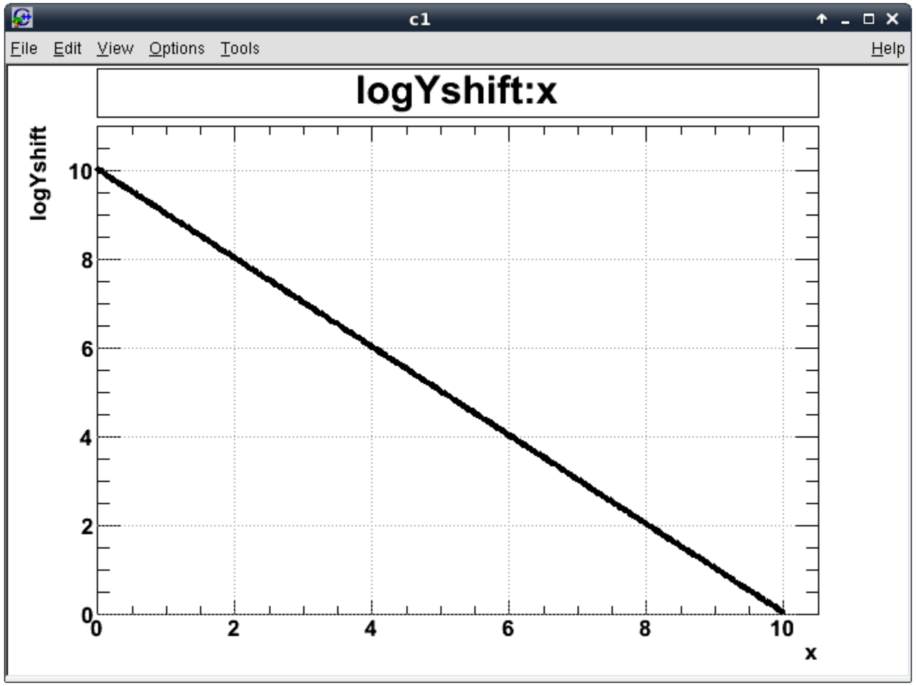
Basic usage

The SetAlias function

Bool_t SetAlias(const char* **name**, const char* **formula**)

	The function defines an alias for a complete formula
name	name of the alias used in the draw function
formula	formula the alias represents

```
root [31] t.SetAlias("logYshift","log(y)+5.4")
(Bool_t)1
root [32] t.Draw("logYshift:x")
```



tree_examples.txt

Basic usage

Access the buffers of the selected data

Long64_t	GetSelectedRows()
Double_t*	GetV1() / GetV2() / GetV3() / GetV4()

Number of selected rows in the draw statement

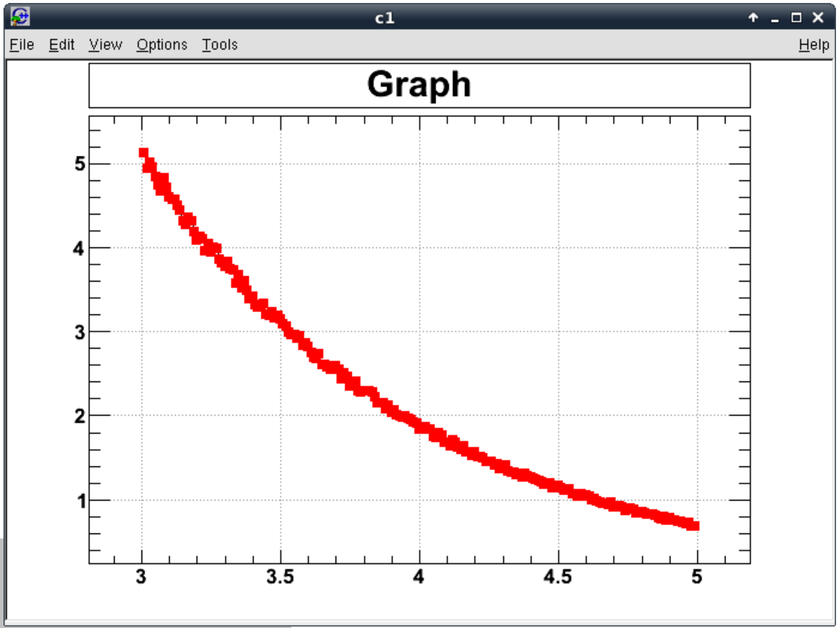
Array of selected data in the first to forth draw string

```
root [32] t.Draw("logYshift:x")
```

GetV1()   GetV2()

Don't draw only fill the buffers

```
root [35] t.Draw("y:x","x>3&&x<5","goff")
(Long64_t)199
root [36] TGraph gr(t.GetSelectedRows(), t.GetV2(), t.GetV1())
root [37] gr.SetMarkerColor(kRed)
root [38] gr.SetMarkerStyle(21)
root [39] gr.SetMarkerSize(1)
root [40] gr.Draw("alp")
```



tree_examples.txt

Basic usage

Directly accessing branch information / looping over the data

Int_t SetBranchAddress(const char* **bname**, void* **add**, TBranch** **ptr** = 0)

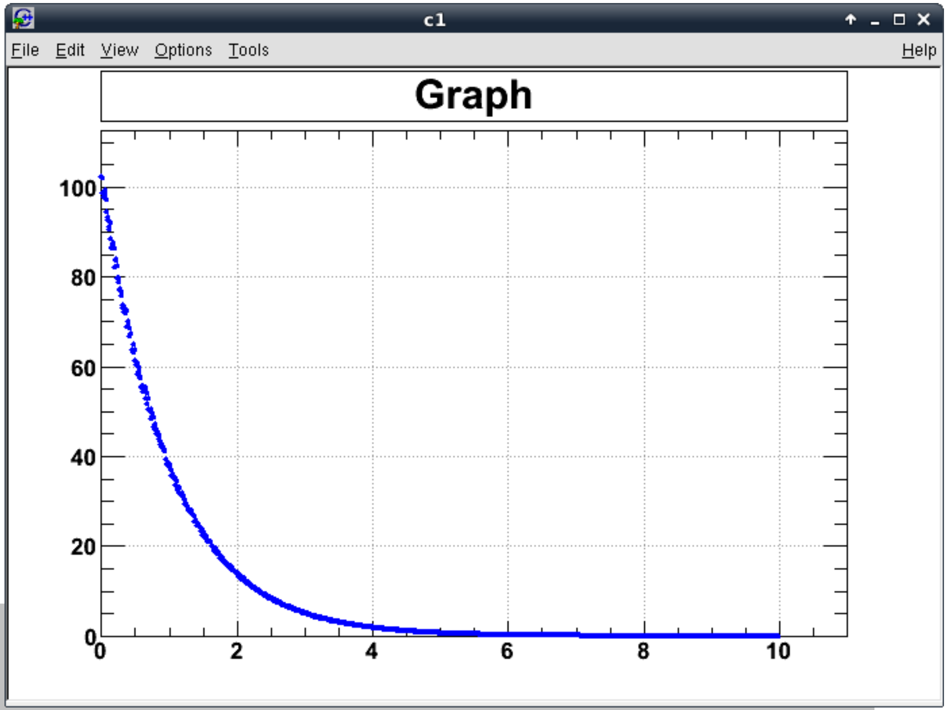
Set the memory address of an object to the branch

bname name of the branch

add address of the object

ptr usually not used

When a branch address is set and GetEntry(*i*) of the tree is called, the value of the branch for entry *i* is stored in the object (variable)




```
root [44] Float_t x=0,y=0;
root [45] t.SetBranchAddress("x",&x)
root [46] t.SetBranchAddress("y",&y)
root [47] TGraph gr(t.GetEntries())
root [48] for (Int_t i=0; i<t.GetEntries(); ++i) { t.GetEntry(i); gr.SetPoint(i,x,y); }
root [49] gr.SetMarkerColor(kBlue)
root [50] gr.Draw("ap")
```

tree_examples.txt

Writing data to a tree

Most important functions

TBranch* Branch(const char* **name**, void* **obj**, Int_t **bufsize** = 32000, Int_t **splitlevel** = 99)

	Add a branch to the tree
name	name of the branch
obj	address of the object to be stored in the branch
bufsize	size of the internal buffers for this branch
splitlevel	how the data of the object are split (browsable), details, see http://root.cern.ch/root/html/TTree.html

Int_t Fill() Fill all branches, by storing the data in the objects of the branches

Writing data to a tree

Simple example storing single values

```
root [0] TTree t
root [1] Float_t val=0
root [2] t.Branch("val",&val)
(class TBranch*)0x226c010
root [3] for (Int_t i=0; i<100; ++i) { val=i/100.; t.Fill(); }
root [4] t.GetEntries()
(const Long64_t)100
root [5] t.Print()
******************************************************************************
*Tree    :          :                                                        *
*Entries :      100 : Total =            1290 bytes  File  Size =          0 *
*        :          : Tree compression factor =   1.00                       *
******************************************************************************
*Br    0 :val       : val/F                                                  *
*Entries :      100 : Total  Size=       1014 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
root [6] t.Draw("val")
```

tree_examples.txt

Writing data to a tree

Simple example storing root objects

```
root [0] TTree t;
root [1] //create a branch to hold the TLorentzVector
root [2] TLorentzVector *v=new TLorentzVector;
root [3] t.Branch("particle",&v);
root [4]
root [4] //generate momentum and energy of the particle
root [5]  for (int i = 0; i < 1000; ++i) {
root (cont'ed, cancel with .@) [6]     double Px = gRandom->Gaus(0,1);
root (cont'ed, cancel with .@) [7]     double Py = gRandom->Gaus(0,1);
root (cont'ed, cancel with .@) [8]     double Pz = gRandom->Gaus(0,1);
root (cont'ed, cancel with .@) [9]     double E  = gRandom->Gaus(10,5);
root (cont'ed, cancel with .@) [10]     //fill the TLorentzVector
root (cont'ed, cancel with .@) [11]     v->SetPxPyPzE(Px,Py,Pz,E);
root (cont'ed, cancel with .@) [12]   t.Fill();
root (cont'ed, cancel with .@) [13]}
```

tree_examples.txt

Writing data to a tree

Simple example storing root objects

```
root [10] t.Print()
******************************************************************************
*Tree    :          :                                                        *
*Entries :      100 : Total =           13352 bytes  File  Size =          0 *
*        :          : Tree compression factor =   1.00                       *
******************************************************************************
*Branch  :particle                                                           *
*Entries :      100 : BranchElement (see below)                              *
*............................................................................*
*Br    0 :fUniqueID : UInt_t                                                 *
*Entries :      100 : Total  Size=       1054 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    1 :fBits     : UInt_t                                                 *
*Entries :      100 : Total  Size=       1434 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    2 :fP        :                                                        *
*Entries :      100 : Total  Size=       7917 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    3 :fP.fUniqueID : UInt_t                                              *
*Entries :      100 : Total  Size=       1072 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    4 :fP.fBits  : UInt_t                                                 *
*Entries :      100 : Total  Size=       1452 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    5 :fP.fX     : Double_t                                               *
*Entries :      100 : Total  Size=       1430 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    6 :fP.fY     : Double_t                                               *
*Entries :      100 : Total  Size=       1430 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    7 :fP.fZ     : Double_t                                               *
*Entries :      100 : Total  Size=       1430 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
*Br    8 :fE        : Double_t                                               *
*Entries :      100 : Total  Size=       1412 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*............................................................................*
```

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

private:
UInt_t **fBits**
UInt_t **fUniqueID**
static Long_t **fgDtorOnly**
static Bool_t **fgObjectStat**

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

private:
Double_t **fE**
TVector3 **fP**

http://root.cern.ch/root/html/TVector3.htm
l

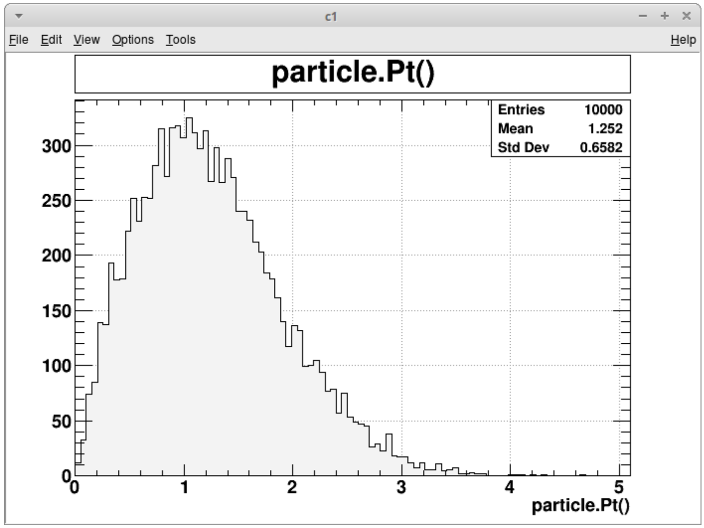
private:
Double_t **fX**
Double_t **fY**
Double_t **fZ**

Drawing from ROOT objects

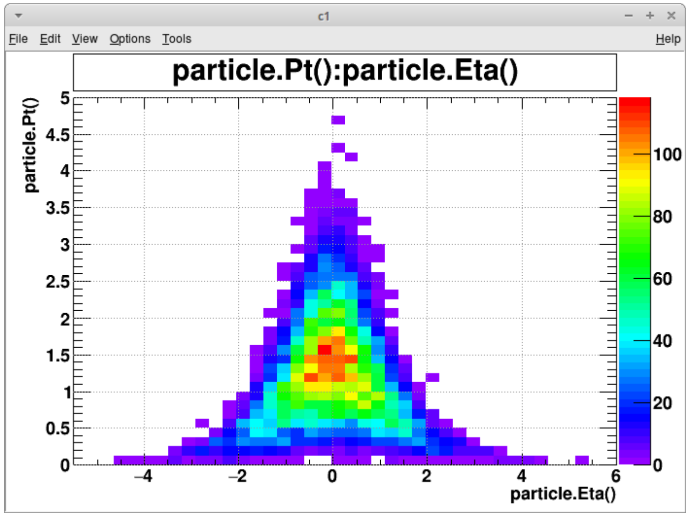
Simple example

An interesting feature when drawing from ROOT objects stored inside a tree is that also the functions of the object can be used

```
root [11] t.Draw("particle.Pt()");
```




```
root [14] t.Draw("particle.Pt():particle.Eta()","","colz");
```



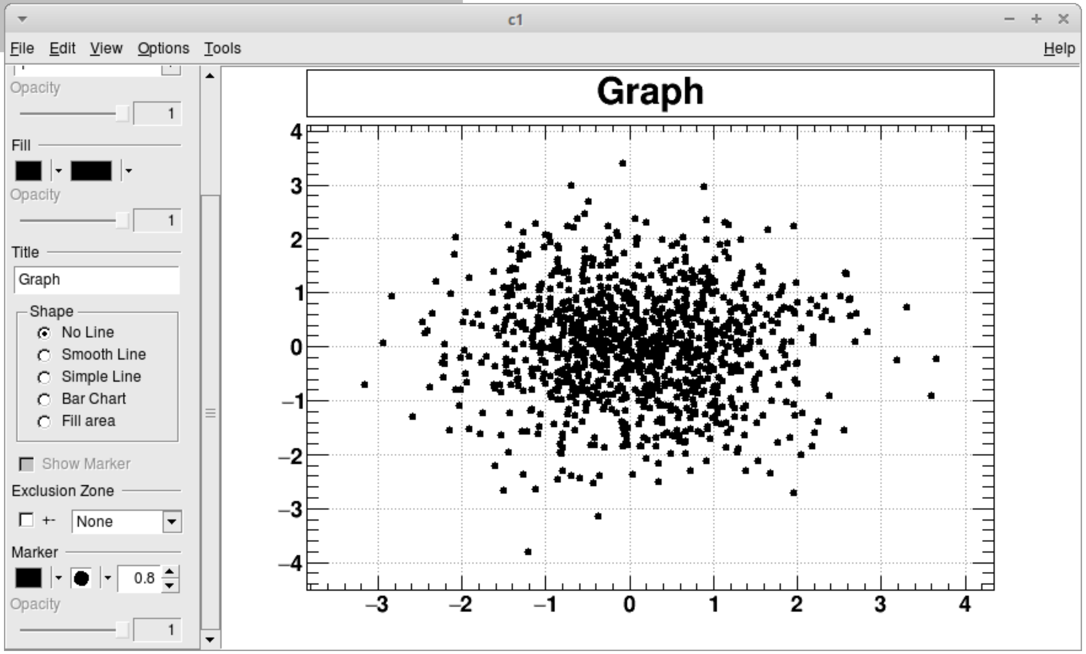
tree_examples.txt

Basic usage

Directly accessing branch information of ROOT objects

- Same syntax as for normal data type
- Objects must be on the heap!

```
root [19] TLorentzVector *v=new TLorentzVector;
root [20] t.SetBranchAddress("particle",&v);
root [21] TGraph gr(t.GetEntries());
root [22] for (Int_t i=0; i<t.GetEntries(); ++i) {
end with '}', '@':abort >   t.GetEntry(i);
end with '}', '@':abort >   gr.SetPoint(i,v->Px(),v->Py());
end with '}', '@':abort > }
root [23] gr.Draw("ap");
```

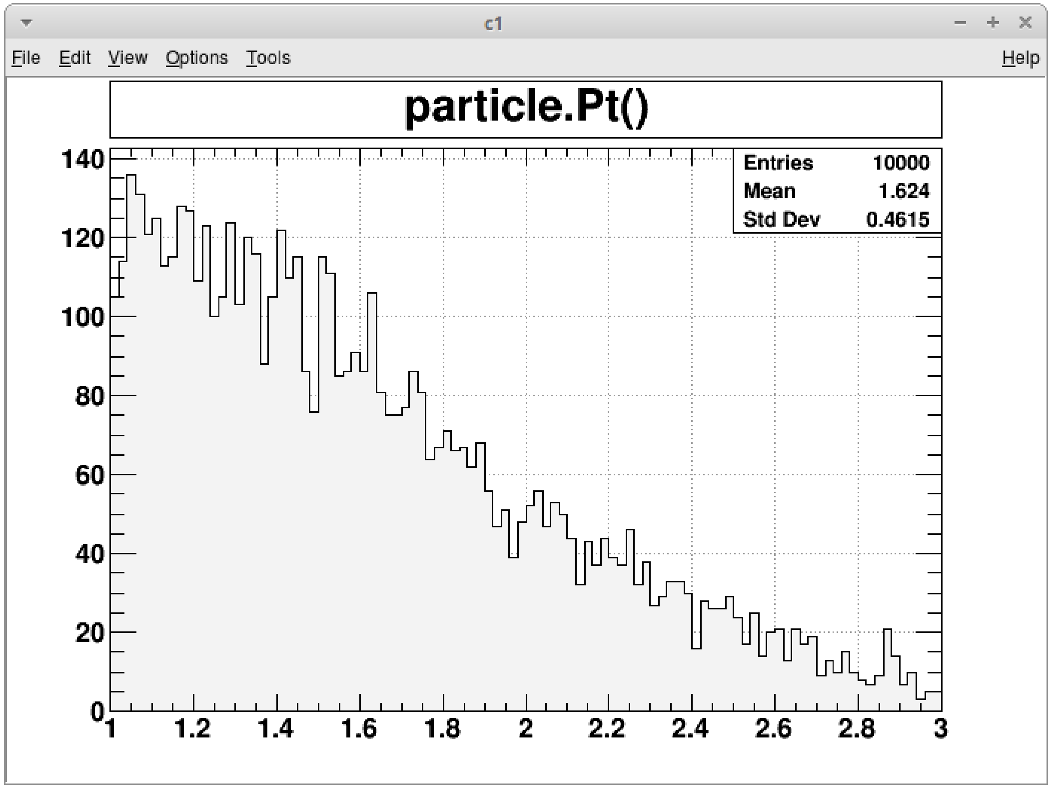


More on drawing

Defining histogram ranges

Histograms can be directly defined in the draw string with the name and the binning with the syntax:
DrawString >> histName(binning);

```
root [19] t.Draw("particle.Pt()>>hPt(100,1,3)");
root [20] gDirectory->ls()
 OBJ: TH1F      hPt     particle.Pt() : 0 at: 0x27616b0
```



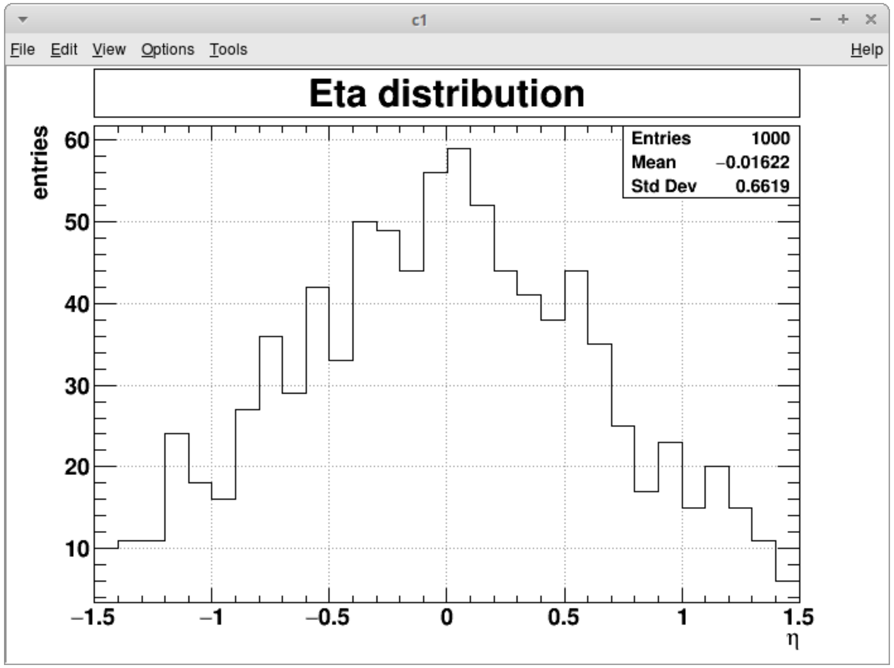
tree_examples.txt

More on drawing

Defining histogram outside the draw function

Histograms to fill in the TTree::Draw(...) function can also be defined before and then identified in the drawString by their name

```
root [23] TH1F hEta("hEta","Eta distribution;#eta;entries",30,-1.5,1.5);
root [24] t.Draw("particle.Eta()>>hEta");
```



tree_examples.txt

More on drawing

Special variables

For details see:
http://root.cern.ch/root/html/TTree.html#TTree:Draw@2

Entry$: return the current entry number (== TTree::GetReadEntry())

Entries$: return the total number of entries (== TTree::GetEntries())

Length$: return the total number of element of this formula for this entry (==TTreeFormula::GetNdata())

Iteration$: return the current iteration over this formula for this entry (i.e. varies from 0 to Length$)

Length$(formula): return the total number of element of the formula given as a parameter

Sum$(formula): return the sum of the value of the elements of the formula given as a parameter. For example the mean for all the elements in one entry can be calculated with: Sum$ (formula)/Length$(formula)

Min$(formula): return the minimum (within one TTree entry) of the value of the elements of the formula given as a parameter

Max$(formula): return the maximum (within one TTree entry) of the value of the elements of the formula given as a parameter

And many more …

Saving a tree

Recommendation

- Most of the time trees shall be written to a file
- The easiest way to associate a tree to a file is to open a TFile first and then create the tree

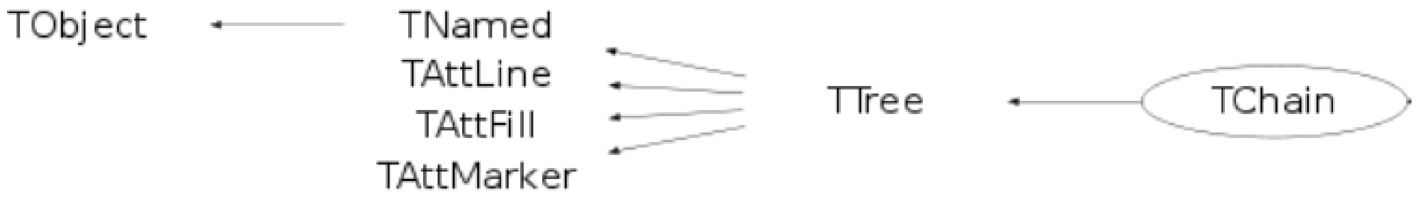
```
TFile f("myFileWithTree.root","recreate");
TTree t("myTree","tree with my data"); //<- give trees a name and title!
.
.
.
f.Write();
f.Close();
```

See example in 'chain.C' (function: writeTree())

Chains

Introduction

- For larger amounts of data it is often necessary to store them in separate files (parallelisation of production …)
- ROOT offers a method for grouping those data by 'chaining' the files (including trees) together
- The class used is TChain, which to the user looks like a simple tree (inherits from TTree)
- The files need to contain trees with the same name and tree structure



Chains

An example 1

```
TChain chain("T");   // name of the tree is the argument
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

The name of the chain needs to correspond to the tree name in the added files.
All files need to have trees with the same name!

```
TChain *c=new TChain("particleTree");
// add all files to the chain
for (Int_t ifile=0; ifile<nFiles; ++ifile){
  TString fileName("particleTree");
  fileName+=ifile;
  fileName+=".root";
  c->AddFile(fileName.Data());
}
```

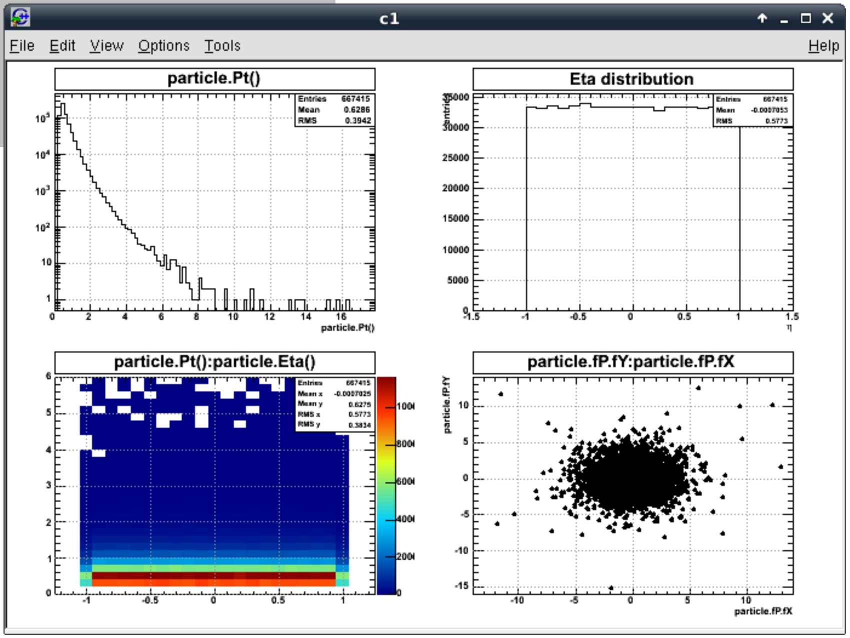
chain.C

```
root [0] TFile f("particleTree1.root")
root [1] f.ls()
TFile**         particleTree1.root
 TFile*         particleTree1.root
  KEY: TTree    particleTree;1  A tree

root [2] TFile f2("particleTree2.root")
root [3] f2.ls()
TFile**         particleTree2.root
 TFile*         particleTree2.root
  KEY: TTree    particleTree;1  A tree

root [4] TFile f3("particleTree3.root")
root [5] f3.ls()
TFile**         particleTree3.root
 TFile*         particleTree3.root
  KEY: TTree    particleTree;1  A tree
```

Chains

An example 2 – drawing

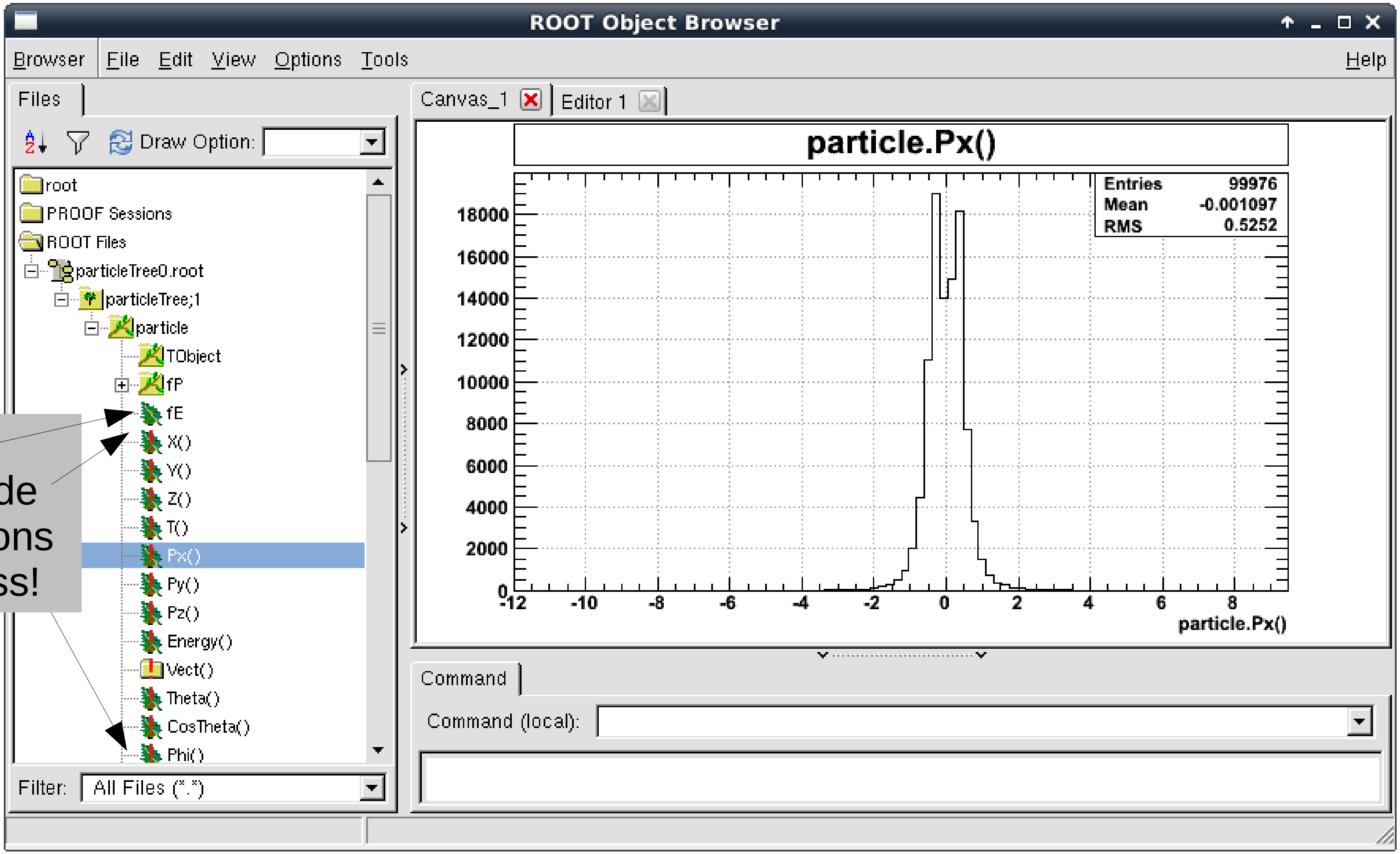
```
root [0] .L chain.C+
root [1] TChain *tree=chain();
root [2] TCanvas c;
root [3] c.Divide(2,2);
root [4] c.cd(1);
root [5] gPad->SetLogy();
root [6] tree->Draw("particle.Pt()");
root [7] c.cd(2);
root [8] TH1F hEta("hEta","Eta distribution;#eta;entries",30,-1.5,1.5);
root [9] tree->Draw("particle.Eta()>>hEta");
root [10] c.cd(3);
root [11] tree->Draw("particle.Pt():particle.Eta()>>hPtEta(25,-
1.25,1.25,30,0,6)","","colz");
root [12] c.cd(4);
root [13] //draw using directly the data members
root [14] tree->Draw("particle.fP.fY:particle.fP.fX");
```



chain.C

Browsing trees

- One can open trees with the TBrowser and inspect the structure
- By clicking on the leaves it is possible to directly draw the data



Leafs, also include the functions of the class!

Exercises

- Write your lab data to a tree and save it to file (alternatively, use file *example_code/day04/vertex.txt containing the coordinates of interaction points in a fixed target experiment*)
  - Draw the data through the tree (via browser or code; e.g draw the different variables and one against the other - ”x:y”, “x:y:z”...- using cuts)

    solution: exercises/day04/trees.C, functions writeDataToTree(), readDataToTree()
- Create a tree with two branches
  - One branch is a simple number (random from a Gaussian distribution)
  - One branch holding TGraphs (only few points)
  - Read back both branches using the branch address

    solution: exercises/day04/trees.C, functions createComplexTree,readComplexTree

Exercises

- Look at the example 'chain.C'.
  - Modify it to create different types of trees
  - Chain your trees and draw results
  - Access results of the chain via the branch address

- Use some of the root data files provided (e.g. run_1245.root, run_1246.root,run_1247.root) :
  - Build a TChain for the TTree named “events”
  - Investigate the structure with the function seen (Print, Scan…)
  - Search correlations between variables in the branches:

    energy (“gamma” variables) in different channels

    time (“time_gamma” variables)

    peak energy (“peak_gamma” variables)

    ….

    Use different cuts on the variables and different drawing option