

ROOT

An Object-Oriented
Data Analysis Framework



Part 3

Maddalena Antonello
INFN, LNGS

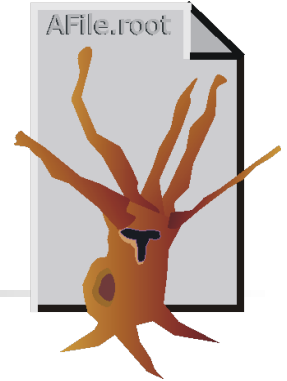
Thanks to: N. Di Marco, S. Panacek and A. Tramontana, L. Pandola



The TTree (finally!)



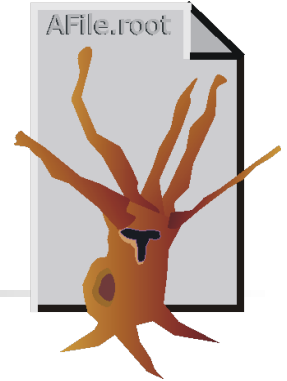
The ROOT trees (TTree)



- A **TTree** is the ROOT implementation of a old-dear **ntuple**
 - **Table** of **correlated** values/objects
 - E.g. energy, time and id number of the same event
- The objects are **not necessarily numbers**
 - It can be an **array** or **any ROOT object** (histos, functions, ...)
 - This includes **user-custom** ROOT objects
 - The **arrays** can be also of **variable size** for each row
 - The actual size of the array is **stored** in an **other column** of the tree
- Binary format, to save and efficiently manage a **large number of entries**
 - It is a real option for **storage** (e.g. raw data)



The ROOT trees (TTree)



- The **TTree** is organized in a **hierarchical** structure of branches (TBranch) and leaves (TLeaf)
 - It is possible to **read selectively** from one branch or leaf only → **no need** to **load** the entire tree
- **Additional branches** can be added at a **later stage**
 - E.g. as a result of some kind of analysis
- Surely the **most powerful** and flexible ROOT object

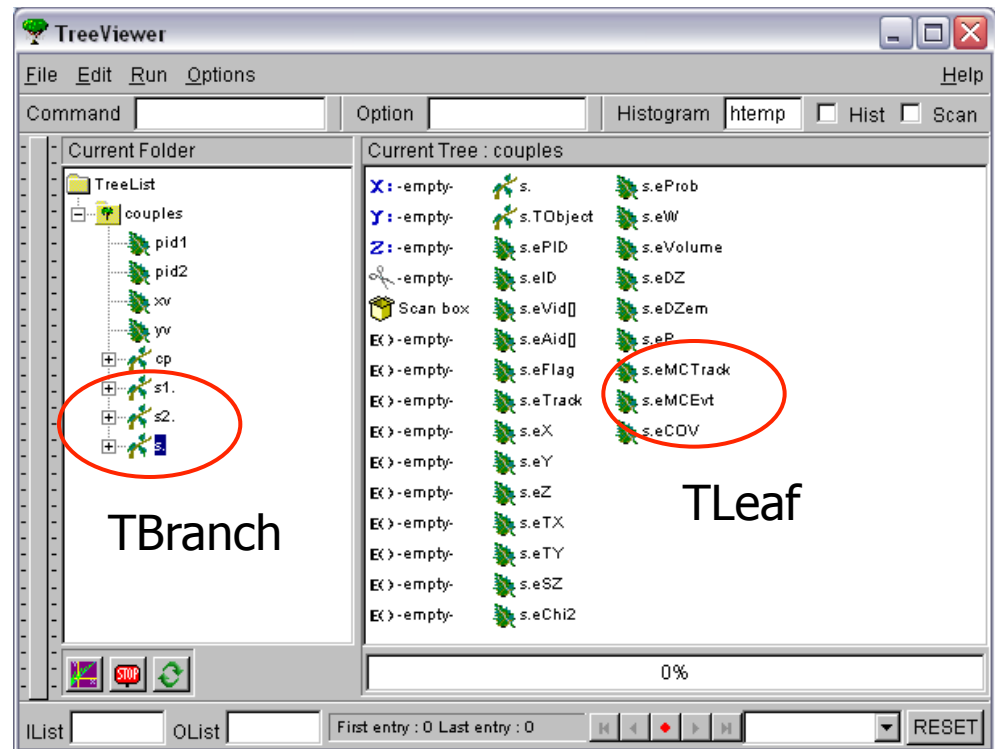
Explore the content of a TTree

- A TTree can be loaded from a TFile exactly like a histogram, i.e. via `->Get()`

```
[ ] TTree* myTree =  
    (TTree*) f.Get("name");
```

```
[ ] myTree->StartViewer();
```

The **tree viewer** allows the **interactive access** to the tree and to all branches and leaves → **double click** to plot



Command-line handling of Ttrees - 1

List of all variables (leaves and branches):

```
[ ] tree->Print()
```

One-dimensional plot of a variable

```
[ ] tree->Draw("varname")
```

Scatter plot of two variables

```
[ ] tree->Draw("varname1:varname2")
```

Add a graphical option (lego2)

```
[ ] tree->Draw("varname1:varname2", "", "lego2")
```

Add a **cut** based on an other variable

```
[ ] tree->Draw("varname1:varname2", "varname3>0", "lego")
```

Scatter plot of three variables

```
[ ] tree->Draw("varname1:varname2:varname3")
```



Command-line handling of TTrees - 2

Show completely the content of one event (all leaves)

```
[ ]> tree->Show(eventNumber);
```

Fit of the 1-dim distribution of one variable

```
[ ]> tree->Fit("func", "varname")
```

Fit adding a **cut**

```
[ ]> tree->Fit("func", "varname", "varname > 10")
```

Class **TCut** to define specific cuts

```
[ ]> TCut cut1="varname1>0.3"
```

```
[ ]> tree->Draw("varname1:varname2",cut1)
```

```
[ ]> TCut cut2="varname2<0.3*varname1+89"
```

```
[ ]> tree->Draw("varname1:varname2",cut1 && cut2)
```



Create, fill and store a TTree

- It is a bit worksome: **5 steps** required
 1. Create the TFile
 2. Create the TTree
 3. Register TBranches to TTree
 4. Fill the TBranches
 5. Write the output file
- **Easy** situation: load branches (**only numbers!**) from an **existing ASCII file**

```
TTree* tree = new TTree("tree", "My Tree Title");  
tree->ReadFile("myfile.dat", "energy/D:time/D:id/I");  
                        filename                Branches and types (D, I)
```




Building a TTree - 1



AFile.root

■ Step 1: Create a new TFile

```
TFile *myfile = new TFile("test.root", "RECREATE") ;
```

The **constructor** of TFile has **arguments**:

- ✓ **file name** (i.e. "test.root ")
- ✓ **mode**: NEW or CREATE, RECREATE, UPDATE, or READ

■ Step 2: Create a TTree object

```
TTree *tree = new TTree("myTree", "A ROOT tree") ;
```

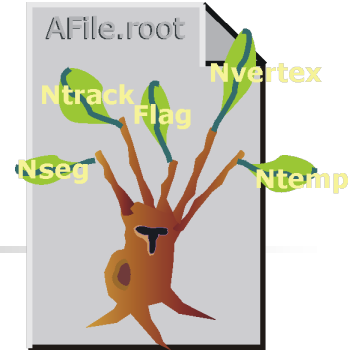
The **constructor** of TTree has **arguments**:

- ✓ **Tree Name** (e.g. "myTree")
- ✓ **Title** (choose a descriptive one, possibly!)



AFile.root

Building a TTree - 2



- Step 3: Add the branches
- Simplest option: **TBranch = TLeaf**
 - Each branch contains only one variable
- Map each branch into a memory address (i.e. a pointer)

```
Int_t ntrack;  
Double_t energy;  
Double_t myArray[10];  
myTree->Branch("NTrack", &ntrack, "ntrack/I");  
myTree->Branch("Energy", &energy, "energy/D");  
myTree->Branch("MyArray", myArray, "myArray[10]/D");
```

Memory address where read the value from

Variable type

Notice: an array is already a pointer



Building a TTree - 3

- Many possible **types**

```
- C : a character string terminated by the 0 character  
- B : an 8 bit signed integer (Char_t)  
- b : an 8 bit unsigned integer (UChar_t)  
- S : a 16 bit signed integer (Short_t)  
- s : a 16 bit unsigned integer (UShort_t)  
- I : a 32 bit signed integer (Int_t)  
- i : a 32 bit unsigned integer (UInt_t)  
- F : a 32 bit floating point (Float_t)  
- D : a 64 bit floating point (Double_t)  
- L : a 64 bit signed integer (Long64_t)  
- l : a 64 bit unsigned integer (ULong64_t)  
- O : [the letter 'o', not a zero] a boolean (Bool_t)
```

- But one can also use **user-custom classes** as TBranch
 - Typical case: the **class** already "**packs**" in itself all the **relevant information** (e.g. MyEvent)
 - So, have a TTree of MyEvents

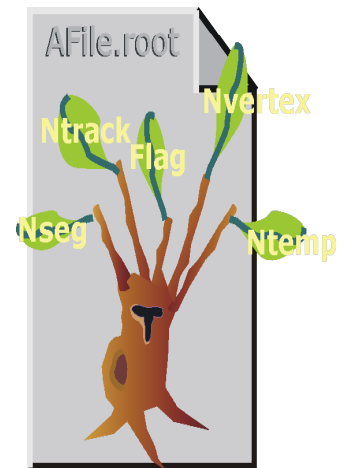
Building a TTree - 4

- Step 3 (alternative): Add the Branches from user-defined classes

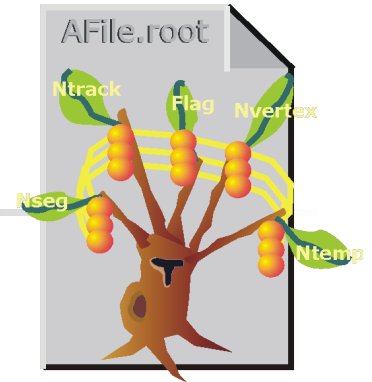
```
MyEvent *event = new MyEvent();  
myTree->Branch("EventBranch", "MyEvent", &event);
```

← User ROOTified custom class

- ✓ Branch Name
- ✓ Class name (optional)
- ✓ Memory address (pointer) of the object to be stored
 - ✓ The class MyEvent may contain several data members (e.g., Ntrack, Flag)
 - ✓ Each of them becomes a TLeaf



Building a TTree - 5



■ Step 4: Fill the TTree

- ✓ Set the **proper values** to all **variable/objects** that have been **registered** as branches or leaves and **Fill()**

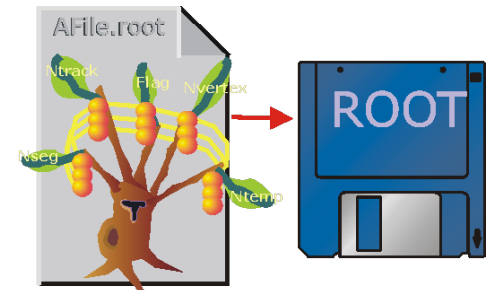
```
event->nTrack = 5;  
event->energy = 12.5;  
myTree->Fill();
```

- ✓ The operation can be **repeated** within a **for()** loop

■ Step 5: Save the TTree on the TFile

The method **Write()** of TFile **writes automatically** all TTrees and all histograms

```
myFile->Write();
```





Extra filling options

- There is the possibility to have **arrays of variable size** as leaves of a TTree
 - Typical case: 1000 detectors and only **one or two** of them are fired
 - Would you store two numbers and **998 zeroes**?
 - Store **only** the **two numbers** (and the detector ID!)
- The **number of elements** (n. of fired detectors) is stored in **another leaf**

```
Int_t nDetectors;  
Double_t energy[NMAX];  
myTree = new TTree("tree","Global results");  
myTree->Branch("NDetectors",&nDetectors,"NDetectors/I");  
myTree->Branch("Energy",energy,"energy[NDetectors]/D");
```



Ok, now we want to read the TTree back

- Already described how to open, read and plot a TTree from **command line** (interactively)
 - Print(), Draw(), Show(), ...
 - Scatter plots, cuts on variables,...
- But what about **retrieving** the **content** of each **TLeaf** for each event **from a macro** or from a **C++ code**?
- ROOT **tutorial** available in
`$ROOTSYS/tutorials/tree1.C`

How to read a TTree - 1

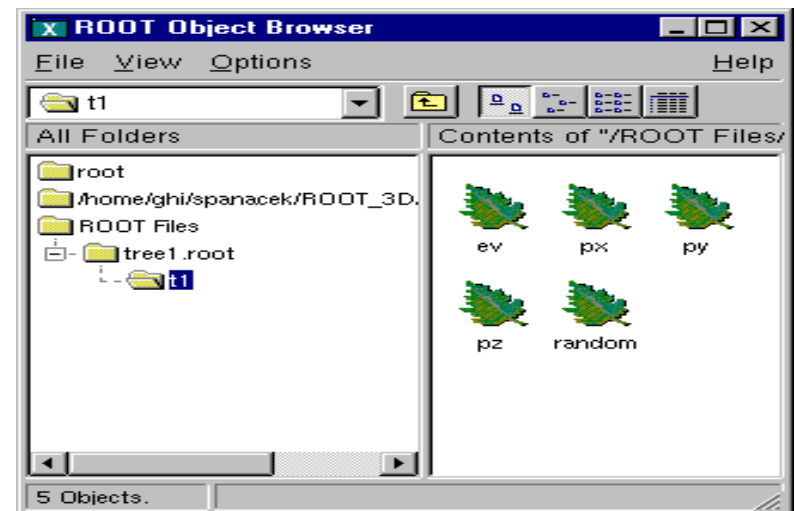
- Open the TFile which contains the TTree

```
TFile* file = new TFile ("tree1.root")  
file.ls();
```

- Retrieve the TTree (via the name)

```
TTree * t1 =  
(TTree*) file.Get("t1")  
t1->Print();  
(or) t1->StartViewer();
```

The TTree here has 5 leaves,
named ev, px, py, pz and random





How to read a TTree - 2

- Create the **appropriate variables** to **store** the data of leaves

```
Float_t px, py;
```

- **Map** the branches/leaves that you want to read into your **local variables** (passing the **memory address** of them)

- You do **not** have to **read all branches**, but only some of them, if you wish

```
t1->SetBranchAddress ("px" , &px)
```

```
t1->SetBranchAddress ("py" , &py)←
```

Branch name

Memory address



How to read a TTree - 3

- Read each row of the TTree using GetEvent(ID);
t1->GetEvent(0); //read first event
- After *each call* of GetEvent() , the **variables** that are **mapped** to a branch **get their actual values**
- One can **loop** over entries and **read** the entire tree

```
for (Int_t i=0;i<t1->GetEntries(); i++)  
{  
    t1->GetEvent(i);  
    //do what you need with the tree content  
}
```

Adding a branch to an existing TTree

- It is possible to add a new TBranch to a TTree which already exists
 - Typical case: you want to add some extra variable calculated from the others

```
TFile f("tree3.root", "update");
TTree *t3 = (TTree*)f->Get("t3");
Float_t new_v;
TBranch *newBranch = t3->Branch("newbr", &new_v, "newbr/
F");
for (Int_t i = 0; i < t3->GetEntries(); i++) {
    new_v = gRandom->Gaus(0, 1);
    newBranch->Fill(); ← Fill only the new branch
}
t3->Write("", TObject::kOverwrite); ← Save only new version
```

Register the
new branch
to the tree

Load many TTrees: the TChain

- Sometimes, you want to merge/load trees **split in many files**
 - Same tree **name**, same **branches**
- May happen e.g. because
 - The tree is **too big** and it is **split** in many files
 - There is **one file** per each **run** of your experiment and you want to load the **entire dataset**

```
TChain *ch = new TChain("tree");
```

```
ch->Add("run1.root");
```

```
ch->Add("run2.root");
```

```
ch->Print();
```

```
ch->GetEntries();...
```

Add files

Use TChain **as a TTree**

Common name
of all trees

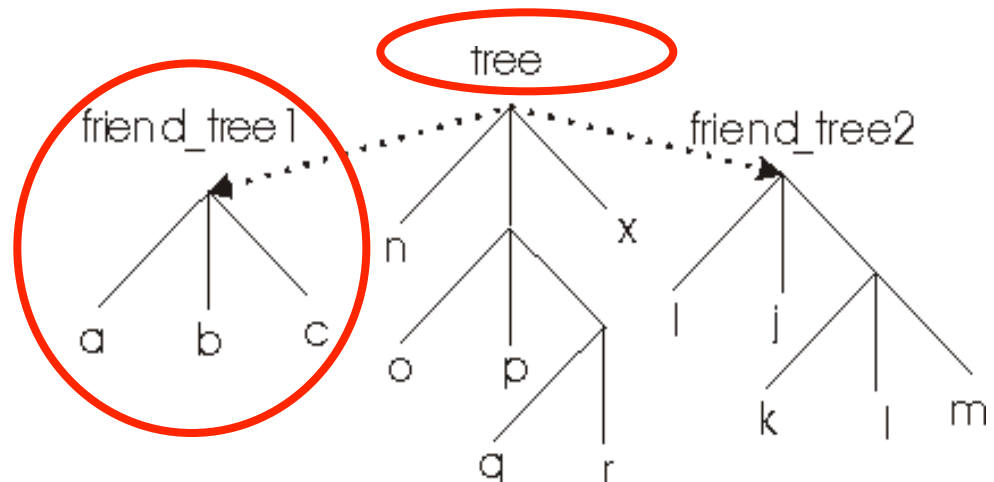


The TTree friendship

TTree friends

- In some cases, it is **not possible/advisable** to add a **new branch** to an existing **tree**
 - The parent tree might be **readonly** (raw data!)
 - **Risk of losing the original tree** with an unsuccessful attempt to save the modification
- Solution: add a **TTree friend**
 - Each TTree has **unrestricted access** to **all** fields/data of its own friends

To all practical purposes,
this is **equivalent** to a
single TTree which
contains tree, friend_tree1
and friend_tree2





Add friends to a TTree

- **AddFriend**("friendTreeName", "fileName")

```
mytree->AddFriend("ft1", "ff.root")
```

- If no file name is given, the friend tree is looked for in the **same TFile** as the starting tree

- If the TTree's have the **same name**, it is mandatory that the friend gets an "**alias**" so that the trees **can be distinguished**

```
mytree->AddFriend("tree1 = tree",  
"ff.root")
```

alias

original name



Access to the friends

- Access:

`"friendTreeName.branchName.leafName"`

- The **leafName** is **sufficient** if it **unambiguously** identifies the leaf

Access to **all**
variables of **all**
TTrees

- Example:

```
mytree->Draw ("t2.px")
```

```
mytree->Draw ("t2.pz", "t1.px>0")
```

```
mytree->SetBranchAddress ("t2.px", &p)
```

- List of **all** branches

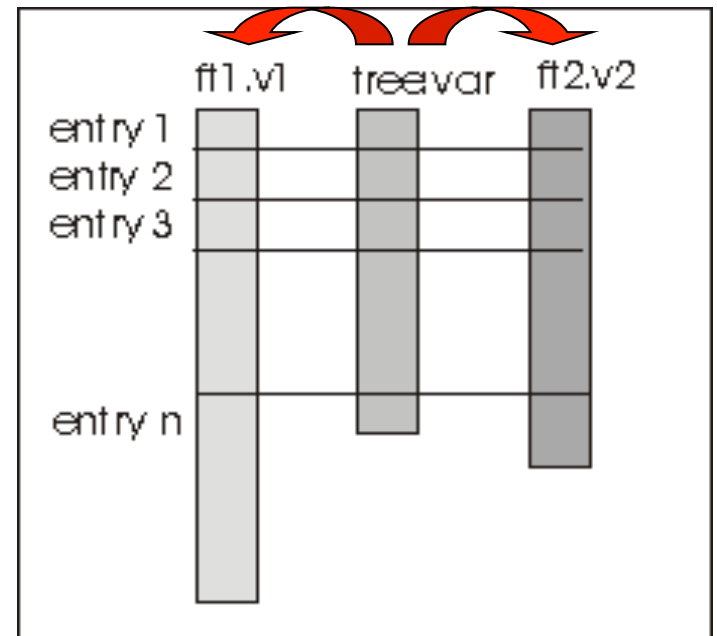
```
mytree->Print ("all");
```


The friend list

- The number of **entries** of the friends tree must be **equal or larger** than the "main" tree
- The "main" tree must be **the shortest one**
- **ft1** can be friend of **tree**, but **tree** cannot be friend of **ft1**

Access to the **friend list**:

TTree::GetListOfFriends()



Definition of user-custom ROOT classes





One more step ahead: "ROOTify" your own class

- It is possible to **ROOTify user-classes**, expanding the ROOT list of classes, so that can be:
 - Instantiated by **command line**
 - written in **ROOT files**
 - used as **branches** in a Tree
- Typical case: customized "**containers**" and new **objects** inheriting from TObject (or TNamed) ROOT features
 - **Encapsulate** your **event** information in a MyEvent class and, in turn, define a "run = TTree of MyEvent objects"
- Can be also done:
 - **Command line** (but **no I/O**)
 - Via **ACLIC** (= compiled code)



Define your own class in ROOT

- Step 1: the user class must **inherit** from **TObject** (or from the derived class **TNamed**)
 - The user class inherits **all characteristics** of the **ROOT objects**, as the **name** (string) and all **methods** for **I/O** and **management** (e.g. **Write()**)
- Step 2: **add** to the **source code** the lines
ClassDef (ClassName, ClassVersionID)
At the end of the *header* (.h)
ClassImp (ClassName)
At the beginning of the *implementation* (.c)



ClassDef() and ClassImp()

- `ClassDef()` and `ClassImp()` are **macros** defined in `ROOT (Rtypes.h)`
- They are **required** to manage the **I/O** of the object and other features:
 - The **streamer** methods to **write** the objects in a **ROOT file** or as **branches of a TTree**.
 - Method **ShowMembers()** to list **public class members**
- User must provide a **default constructor**



A concrete example

```
class MyTRun : public TNamed, public MyRun
{
public:
    MyTRun() {;};
    virtual ~MyTRun() {;};

    ClassDef(MyTRun, 1) // Run class
};
```

.h

```
#include "MyTRun.hh"
```

```
ClassImp(MyTRun);
```

.C

Double inheritance



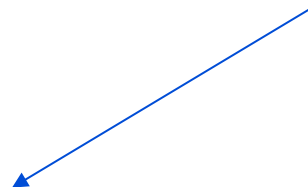


That's not enough...

- Step 3: create a file called **LinkDef.h**. It is required to **notify** ROOT of the presence of a **new user-custom class**, to be included in the dictionary

```
#ifdef __CINT__  
#pragma link off all globals;  
#pragma link off all classes;  
#pragma link off all functions;  
#pragma link C++ class MyTRun;  
#endif
```

Line to add





Still, that's not enough...

- Step 4 (and last): prepare a Makefile which calls the command `rootcint`, which generates the class dictionary

```
$ (ROOTSYS) /bin/rootcint
```

```
-f MyDictionary.cxx
```

```
-c MyTRun.h LinkDef.h
```

- `LinkDef.h` must be the last argument of the `rootcint` command line
- The name of the LinkDef file must contain the string `LinkDef.h` or `linkdef.h`:
 - `MyNice_LinkDef.h` is ok



Loading and Using your brand-new ROOT class (library)

- From command line:

```
root[0] .L libMyTRun.so;  
root[0] MyTRun theRun;  
root[1] ...  
root[2] Tfile f("run001.root","CREATE");  
root[3] theRun.Write("run001");
```

- From script:

```
gSystem->Load("libMyTRun.so");  
MyTRun theRun;
```

- For a complete example see `Event.cxx`, `Even.h`, `EventLinkDef.h`, `Makefile` in `$ROOTSYS/test`



ROOT extras



Other tools available in ROOT

- In these lectures, there was only an overview of the **main tools** available in ROOT
- There are **many more**, e.g.
 - Linear algebra
 - Physics Vectors
 - Support for **custom GUI's** and interface to Qt
 - Handling of spectra (**TSpectrum**)
 - Python module (**PyROOT**)
 - **Geometry** package
 - HTML **Automatic documentation**
- **Not** all tools are **compiled** by **default** when building ROOT. Some of them have to be **activated explicitly**



Other tools available in ROOT

- Additional tools for (advanced) fitting
 - Minuit2
 - **RooFit**
- **RooFit** initially developed by BaBar
 - Model the expected event distribution of events
 - Unbinned maximum likelihood fits
 - Generate "toy Monte Carlo" samples for various studies

```
gSystem->Load("libRooFit") ;  
using namespace RooFit ;
```
- Some modules/tools were provided by experiments or by other users



Other tools available in ROOT

- Toolkit for **Multivariate Data Analysis** (TVMA)
 - **External** package, distributed with ROOT



- Includes **advanced** analysis tools of the “supervised learning” family
 - Artificial Neural Networks, Boosted/Bagged decision trees, Support Vector Machine, Multidimensional probability density estimation, Rectangular cut optimisation
- Since root version 5.11/06 TMVA is **integrated** in ROOT and can be used directly from ROOT prompt



It is your turn, now:

- Try Task2 under

`http://geant4.lngs.infn.it/ROOTCatania2014/
introduction/index.html`



Extra filling options - 2

- Dynamic vectors (variable size) as leaves of a Ttree

*Vector size = 2
Is the number of fired
detectors*

```
std::vector<Double_t> energy;  
energy.push_back(3.4);  
energy.push_back(2.7);  
myTree = new TTree("tree", "Global results");  
myTree->Branch("Energy", &energy);
```



How to read a TTree - 4

- With dynamical vectors