Working with ROOT in Particle Physics

by Andrea Fontana
Dept. of Nuclear and Theoretical Physics
University of Pavia, 21-25 may 2001

Interaction of photons with matter: the new ROOT interface to GEANT and a simple calorimeter tutorial.

- Review of OO concepts with the aid of ROOT.
- ROOT classes used in HEP: analysis, physics, GUIs.
- Extending the ROOT library: the ROOT dictionary.
- Debugging a ROOT program: scripts vs executables.

What is ROOT?

ROOT is an object oriented framework designed for solving the modern High Energy Physics challenges in data storage and analysis, but it is also used in other fields (medicine, finance…).

ROOT evolved from PAW and the CERN library of fortran routines used by physicists over the past 30 years. It is strongly based on an OO approach and it consists of a C++ interpreter (CINT) and a series of libraries for creating histograms, graphs, ntuples and other structures to store data.

The latest release also includes functions useful for Physics simulations and computations, like a Phase Space generator, a PDG particle database and an interface to *GEANT 3.21*, the package for detectors design and simulations of particles interaction with matter!

It is continuously evolving at http://root.cern.ch

What Object Oriented programming means and what has it to do with Physics?

OO programming is the latest and most powerful way to design and implement software systems. It evolved from a better understanding of the rules behind complex systems in the real world and mainly from the fact the complexity is highly structured and that systems can be broken down in small subsystems.

Computation in Physics today makes an extensive use of these techniques and the two worlds are evolving in parallel and together, as that most (if not all) the software of new HEP projects is OO *(see Conference ACAT 2000 at FNAL).*

Important to keep Physics and OO technicalities well separated!

Aim of these lessons

To use ROOT for writing a simulation program for a simple electromagnetic calorimeter and to use it as a computer lab to review the basics of the interaction of photons with matter.

To present a ROOT interface to GEANT (developed within the Alice and Athena *(V. Filippini)* Collaborations).

To explain how ROOT and an OO approach can be used in Particle Physics to study detector performances.

To show a few ROOT tricks I've learned on the way…

Interaction of photons with matter

A photon with 1 GeV of energy enters a cubic volume of aluminum.
What it is going to happen?

Physics tells us that 3 processes must be taken into account:
- *Compton scattering;*
- *Photoelectric effect;*
- *Pair production;*

and we know exactly the cross sections of these processes for different energies and for a series of materials, as for the subsequent interactions of the produced electrons and positrons (bremstralhung, dE/dx).

What if the material is lead or uranium or if we change the photon initial energy? The answers to these questions will come from a simulation program we are going to write as a tutorial on ROOT/GEANT.

A very simple calorimeter

We want to study these interaction of photons and electrons/positrons within a volume of a material by:

- Tracking all the particles produced by Compton, photoelectric and pair creation processes and by all the subsequent interactions;

- Recording the total energy deposed in a given volume of material.

We need to store the 3D coordinates of all the interaction points and the energy deposed at a given point.
To do this we will build a class with these variables: a TObjArray of T**Vector3** objects and a Double_t for the lost energy.

The TVector3 class

The Vector3 class is a simple implementation of a 3-dimensional vector and consists of three Double_t variables representing the cartesian coordinates and of a set of methods representing operations on a 3D vector.

We rewrite it from scratch, call it Vector3 and use it to review the following concepts:

- Classes and objects
- Constructors and destructors
- Pointers and references.
- Use of memory: stack and heap
- Overloading and inheritance.

Classes and Objects

An object is just simply a miniature complete program with data and functions (called member functions or methods) to use these data and it represents a concept from the real world.
Examples: a vector, an histogram, a particle, a detector...

A class is the description of the data and the member functions that an object of the same class refers to.
The writer of the class decides exactly how the object can be used and this is independent of the application in which it is used (or reused).

To use an object, the user sends a message to the object (i.e. calls a member function) and gets a reply in return: the reply can then be used in any other expression.

Objects are new data types that can be used as any other variable, but they offer much more possibilities.

Use of a simple class Vector3

Class with only data

```
class Vector3
{
        Double_t fX;
        Double_t fY;
        Double_t fZ;
};
```

and its use:

```
Vector3 a,b;
a.fX=3;
a.fY=4;
a.fY=5;
b.fX=a.fY;
b.fZ=a.fX-a.fZ;
```

Syntax for class definition:
- object name
- . member selector operator
- internal variable

This very simple class is very similar to a C-like structure to group together in one single unit variables related to each other.

The . operator is very important and its use allows to access the class data members individually, although this is not recommended, as we'll see…

Member Functions

Class with data and member functions: *getters & setters*

```
class Vector3
{
        Double_t fX, fY, fZ;

        Double_t GetX() { return fX; }
        Double_t GetY() { return fY; }
        Double_t GetZ() { return fZ; }

        void SetX(Double_t X) { fX = X; }
        void SetY(Double_t Y) { fY = Y; }
        void SetZ(Double_t Z) { fZ = Z; }

};
```

The syntax to use a member function relies on the selection operator .:

```
Vector3 a,b;
a.SetX(3);
b.GetZ();
```

Important points

• Given a class, there can be as many independent objects as the user needs: an object is then much different from a regular subroutine with local variables (internal data) and various entry points (functions).

• Collections of objects (arrays) are possible: each component of the array is independent from the others and has its own data members and functions.

• Distinguish between abstraction and encapsulation:
given a class to be designed, it is very important to design it with internal complexity and external simplicity.
- Encapsulation: hiding of internal complexity
- Abstraction: presenting a simple user interface.

Example: a pbar generator object. Internally based on a complex physical model, but externally usable by someone who does not care about this model, but simply wants a pbar source!

Headers and implementation files

It is customary to separate the class definition (abstraction)
From the member functions definition (implementation).

Vector3.h

```
class Vector3
{
        Double_t fX, fY, fZ;

        Double_t GetX();
        Double_t GetY();
        Double_t GetZ();

        void SetX(Double_t X);
        void SetY(Double_t Y);
        void SetZ(Double_t Z);
};
```

Vector3.C

```
#include "Vector3.h"

Double_t Vector3::GetX() {return fX;}
Double_t Vector3::GetY() {return fY;}
Double_t Vector3::GetZ() {return fZ;}

void Vector3::SetX(Double_t X) {fX = X;}
void Vector3::SetY(Double_t Y) {fY = Y;}
void Vector3::SetZ(Double_t Z) {fZ = Z;}
```

Class design is very important

Before writing an OO program, classes need to be planned according to the problem to be solved.

Question to ask yourself are:

- What concept or abstraction the object represents?
- What data members does it hold?
- Which data are needed when an object of this type is created?
- Which member functions are to be provided, i.e. which messages can be sent?

Try to answer for our Vector3…

Try to answer for our simple calorimeter…

Constructors and Destructors

Two member functions are very important and on them Relies the high degreee of autonomy of objects:

- constructor
- destructor

Constructors and destructors are member functions with a special role and have the same name of the class (with a leading ~ for the destructor). They do not have a return value and cannot therefore be called by the user.

A constructor is used to establish initial values for data members, while the destructor is called whenever an object is deleted from memory.

Vector3: C & D

Class with data and member functions:
getters & setters, constructor & destructor

```
class Vector3
{
        public:

        Double_t fX, fY, fZ;

        Vector3();
        Vector3(Double_t X, Double_t Y, Double_t Z);
        ~Vector3();

        Double_t GetX();
        Double_t GetY();
        Double_t GetZ();

        void SetX(Double_t X);
        void SetY(Double_t Y);
        void SetZ(Double_t Z);
};
```

Vector3: C & D

Implementation file

```
#include "Vector3.h"

Vector3::Vector3()
{
        SetX(0); SetY(0); SetZ(0);
}
Vector3::Vector3(Double_t X, Double_t Y, Double_t Z)
{
        SetX(X); SetY(Y); SetZ(Z);
}
Vector3::~Vector3()
{
}

Double_t Vector3::GetX() {return fX;}
Double_t Vector3::GetY() {return fY;}
Double_t Vector3::GetZ() {return fZ;}

void Vector3::SetX(Double_t X) {fX = X;}
void Vector3::SetY(Double_t Y) {fY = Y;}
void Vector3::SetZ(Double_t Z) {fZ = Z;}
```

Public and Private

It is possible to access objects functions with the operator . :

```
x=a.GetX();
```

and also data members with the same syntax:

```
x=a.X;
```

But this is a very bad thing to do and can easily introduce errors and inconsistencies…

To prevent this, the class designer can inhibit the user access to data members and only grant this via class methods (like getters and setters): this is achieved by the keywords public and private and is usually referred to as data hiding.

A bad class example

Example with a four-momentum of a particle of given mass:

```
class FourMomentum
{
        Double_t fE, fM, fP;
        void SetEnergy(Double_t Energy);
};

Void FourMomentum::SetEnergy(Double_t Energy)
{
        fE = Energy;
        fP = sqrt(fE*fE – fM*fM);
}
```

If all is accessible, an user could write then:

```
FourMomentum a;
a.fM=1;
a.fE=2;
a.fP=3; ???
```

But it would violate $p^2=m^2$!

Setting energy via the method, always guarantee consistency.

Why it is safe not to access data?

- The function SetEnergy() sets the energy but also the momentum of the particle to guarantee consistency: leaving to the user to change freely this parameters is a potential source of errors that are left behind by changing data members (private) only via the provided methods (public).

- There is some redundancy as only 2 variables out of fE, fM and fP are independent. Keeping 3 or 2 variables and which 2 is a design problem the user does not need to know, so it is an implementation issue that must be hidden or encapsulated. The user only wants to be able to set the energy, which is part of the model or the abstraction.

Vector3: hiding data

Public and Private scope in a class

```
class Vector3
{
        public:
        Vector3();
        Vector3(Double_t X, Double_t Y, Double_t Z);
        ~Vector3();

        Double_t GetX();
        Double_t GetY();
        Double_t GetZ();

        void SetX(Double_t X);
        void SetY(Double_t Y);
        void SetZ(Double_t Z);

        private:
        Double_t fX, fY, fZ;
};
```

C++ defaults to private and so far our Vector3 class has not been useful at all!

A third level of protection, called protected, is available. (see inheritance)

Pointers and References

Variables and objects use memory and, in order to access any data type, it is necessary to know where it is located in memory. This can be done in two ways:

- static: the compiler/linker decide and manage addresses.

- dynamic: the user manage addresses at execution time.

ROOT and OO programs are dynamic and some kind of pointer that holds the memory address has to be used. Two types:

pointers: a pointer holds the address of an object.

references: a reference is another name for the same object.

Pointers

Pointers hold addresses of data types: to form an address the address operator & is used, while to read the memory content at a given address the dereference operator * can be used:

```
Vector3 a,b;
Vector3 *aPtr = &a;
*aPtr = b;
```

To access methods via a pointer to an object, because of operator precedence (. takes priority over *), the syntax is:

```
x = a.GetX();
y = (*aPtr).GetY();
```

which can be written with a shortcut:

```
y = aPtr->GetY();
```

References

A reference is just simply another identifier for an object or a variable. It is a type of pointer that must be initialized when it is declared with the variable it references to:

```
Vector3 &a=b;
```

This makes a a reference to b and from now on the two are equivalent:

```
b=7;  is equivalent to a=7;
```

The C++ knows that a is a pointer that needs to be dereferenced to be, but this is done automatically and all the changes to b reflects to a.

Once defined, a reference cannot be reassigned a new value and something like `b=&c;` generates an error.

Calling functions

Because C++ copies arguments of functions by value, references and pointers are handy in function calls. They are equivalent, but references make the code a lot clearer avoiding the use of & and * symbols.

pointer

```
void Change(Int_t *var)
{
        *var = 7;
}

Int_t value = 2;
Change(&value);
cout << value << endl;
```

reference

```
void Change(Int_t &var)
{
        var = 7;
}

Int_t value = 2;
Change(value);
cout << value << endl;
```

Passing pointers to functions also save time when the argument is an object with many data.

Stack and Heap Objects

Declaring a static variable, places it on the stack with a scope limited to the local program module {}:

```
Vector3 a(0.,0.,1.);
```

To give an object more persistency, it can be placed on the heap by declaring a pointer to it and by reserving some memory with the new operator:

```
Vector3 *a = new Vector3(0.,0.,1.);
```

The object is initialised by calling its constructor and can be deleted freeing the memory with the delete operator:

```
delete a;
```

Stack objects are local, while heap objects are global.

Working on Stack and Heap

Declaring a dynamic variable via its pointer, always brings two items in existence:

- The heap object;
- The pointer to it, which is defined on the stack.

Heap objects get deleted by the user, while stack objects are deleted when they go out of scope or by resetting the ROOT stack with `gROOT->Reset();`

```
Vector3 a(1.,2.,-3.);
Vector3 *b = new Vector3(1.,2.,-3.);
gROOT->Reset();
```

This deletes only the first object a, but the second cannot be accessed as the pointer b has been removed from the stack as well. This is a memory leak: heap object exists, but cannot be used! But ROOT has a solution for this(s. ee inheritance)

Overloading

Overloading means to reuse the same symbol or function for two distinct operations and makes the meaning of an operator or of a function context depending.

It is already present in non OO languages, like basic and fortran! i.e.: 1+2 vs 1.+2., the code called is very different.

- Function overloading:

  calling the same function with different number and types of arguments (tipical for constructors).

- Operators overloading:

  use of algebrical operators (+,-,,/) for non basic data types and of [] and () operators (see collections).*

Function overloading

Typical function overloading is with constructors:

```
Vector3();
Vector3(Double_t X, Double_t Y, Double_t Z);
```

A default is required, i.e. a function without arguments and also function with different types and/or different numbers of arguments:

```
Vector3(Double_t X, Double_t Y=0, Double_t Z=0);
Vector3(Double_t X, Double_t Y=0);
Vector3(Double_t X);
```

But all these methods must be present in the class definition. C++ is able to choose the right implementation of the function called by finding the exact match!

Operator overloading

How to overload the addition operator + to be able to add two Vector3 objects with a very short and intuitive command:

```
Vector3 a,b,c;
c = a + b;
```

Prototype of the operator + function in header file:

```
Vector3 operator + (Vector3 &, Vector3 &);
```

Function implementation:

```
Vector3 operator + (Vector3 &a, Vector3 &b)
{
        return Vector3( a.GetX() + b.GetX(), a.GetY() +
                        b.GetY(), a.GetZ() + b.GetZ());
}
```

Inheritance

Inheritance means that a class inheriths a few of its characteristics (i.e. data members and methods) from some other classes, the baseclass. This is one of the OO philosophy features, as it allows, if well implemented, to *never write the same code twice!*

Our Vector3 class does not do very much and we could add to it some functionality by inheriting it from the Tobject class (mother class for all the ROOT classes).

```
Class Vector3 : public TObject {
…
};
```

This adds to Vector3 objects all the data members and methods of the TObject class, for instance:

```
Vector3 a;
a.Print();
a.Dump();
```

A Vector3 now IS_A Tobject: *the baseclass constructor is called before the derived class constructor executes.*

Protected access

The public keyword specifies how the member class access is modified: should we have defined the TObject data members as public (very bad programming…), to block the access to the same class while using it as a baseclass for a Vector3 we would write:

```
Class Vector3 : private TObject {
…
};
```

Because Tobject has instead private data members, the Vector3 class cannot either access them internally. To allow this, there is a third type of data protection – protected – which still keeps the user to access private data members, but allow subclasses to do it.

When inheriting as public, everything is left the way it is. When inheriting from a base class, security can only be raised and not lowered: each member takes the higher priority of:
- its original security
- the security specified when inheriting.

More on inheritance

- Overloading and inheritance

When calling `a.Print();` , the Print() method of Tobject is called because a Vector3 IS_A Tobject. Suppose we add a Print() method to out Vector3 class to change the printing format (make it more fancy):

```
Vector3::Print(){…};
```

Now a call to `a.Print();` will generate a call to the Vector3 own Print() method, which overloads the base class one.

- Pointer conversion

A Vector3 object now contains embedded objects of class Tobject. Given the inheritance tree, it is possible to write:

```
Vector3 *vector = new Vector3(6.,5.,1.);
Tobject *object = vector;
```

Conversion in the opposite direction is permitted, but with an explicit cast:

```
vector = (Vector3*)object;
```

ROOT memory management: named objects

Accessing objects on the heap requires a pointer that can be deleted from the stack when it goes out of scope: this generates a memory leak, i.e. memory is allocated but cannot be accessed!
ROOT allows to locate objects in memory without having pointers to them by means of the concept of object naming.
Once an object inherits from Tnamed:

```
class Vector3: public Tnamed {
...
```

we can name it with:

```
Vector3 *a = new Vector3(1.,2.,3.);
a->SetName("dubna");
```

and print its contents:

```
a->Print();
```

ROOT and user objects

Suppose now the pointer a goes out of scope. Within CINT, the memory leak is not a serious problem now because we named the object and can still retrieve the memory contents by creating another pointer (note the casting to Vector3):

```
Vector3 *b = (Vector3*) gROOT->FindObject("dubna")
```

and write:

```
b->GetX();
b->Print();
dubna->Print();
```

Note that the shortcut:

```
dubna->GetX();
```

does not work. This is because, as far as CINT is concerned, the object called "du bna" is just a Tobject (see the cast).

RTTI methods for classes

With a command like the last one, CINT interrogates a class to see what type it really is and to see how to handle it. ROOT classes have methods to reply so that ROOT can understand them and this feature is known as Run Time Type Information (RTTI).

To fully use the named object features we need to have ROOT understanding better our simple Vector3 and we would have to extend ROOT by creating a dictionary.

A dictionary contains this extra member functions for user classes that tell ROOT what class an object is, so that the naming trick also works for it. It also contains advanced methods for I/O, like streamers, that we will see later on…

Virtual functions

Consider the following code:

```
Vector3 *vector = new Vector3(6.,5.,1.);
Tobject *object = vector;
vector->Print();              calls Vector3::Print()
object->Print();              calls Tobject::Print()
```

where we want to make use of our overloaded and fancy print method, without calling the Tobject one. But with the above code it wont happens, as the object does not know that it is embedded within vector! This is undesirable, as we want to always print with our method. The solution is to declare the method as virtual in the baseclass:

```
class TObject {
        public:
        virtual void Print();
}
```

This technique is called polymorphism ans is at the heart of ROOT and OO coding. Dramatic use in `Tobject->Draw();`!

ROOT classes used in HEP

ROOT has been developed by physicists (mainly by Rene Brun, Fons Rademakers and Valery Fine) and it was originally thought as a tool for physicists in the HEP community for analysing and displaying data..

Now ROOT is much more widely known and used and its role became more general: it is now used in other fields, like in mathematical finance and in banking/trading (Rquant class).

Its main role is still for the physicist, though, and the classes that it provides can be grouped as follows:

- classes for data analysis;
- classes for Physics computations and simulations;
- classes for programming graphical interfaces;

Analysis

The first classes to be studied to start playing with ROOT are the ones to create histograms, graphs and trees/ntuples.

Since these classes play a central role in many ROOT applications in HEP, well see how to use them in the context of analyzing the data coming from our simple calorimeter:

- TH1F, TH2F
- TGraph
- TFile
- Tntuple
- Ttree

Wel l see also how to fit an experimental histogram (e.g. a dE/dx distribution for electrons) with a parametrical function (e.g. a Landau distribution).

Histograms

ROOT supports histograms in 1D, 2D and 3D and also profile histograms.
Histograms are created with the constructor:

```
TH1F *h1 = new TH1F("h1","h1 title", 100, 0, 5);
TH2F *h2 = new TH2F("h2","h2 title", 100, 0, 5, 100, -1, 1);
```

With the following parameters (for the TH1 object):
- histogram's name (TH1F inherits from Tnamed)
- histogram's title
- number of bins
- x minimum
- x maximum

The bin size is fixed by default, but variable bin sizes are supported with the constructor

```
TH1F *h1 = new TH1F("h1","h1 title", 100, xbins);
```

Where xbins is an array of low edges for each bin.

Filling Histograms

Histograms are filled using the Fill() method that is overloaded to handle different cases:

```
h1->Fill(x);
h1->Fill(x,w);
h2->Fill(x,y);
h2->Fill(x,y,w);
```

The Fill() method computes the bin number corresponding to the given x and y argument and increments this bin by the given weight (the default weight is 1).

During filling some statistical parameters are incremented to compute the mean value and the root mean square.

A random number filling method is available to fill histograms according to a distribution:

```
h1->FillRandom("gaus",10000);
```

Drawing Histograms

Histograms can be drawn in various formats and with many options. The basic command is:

```
h1->Draw();
```

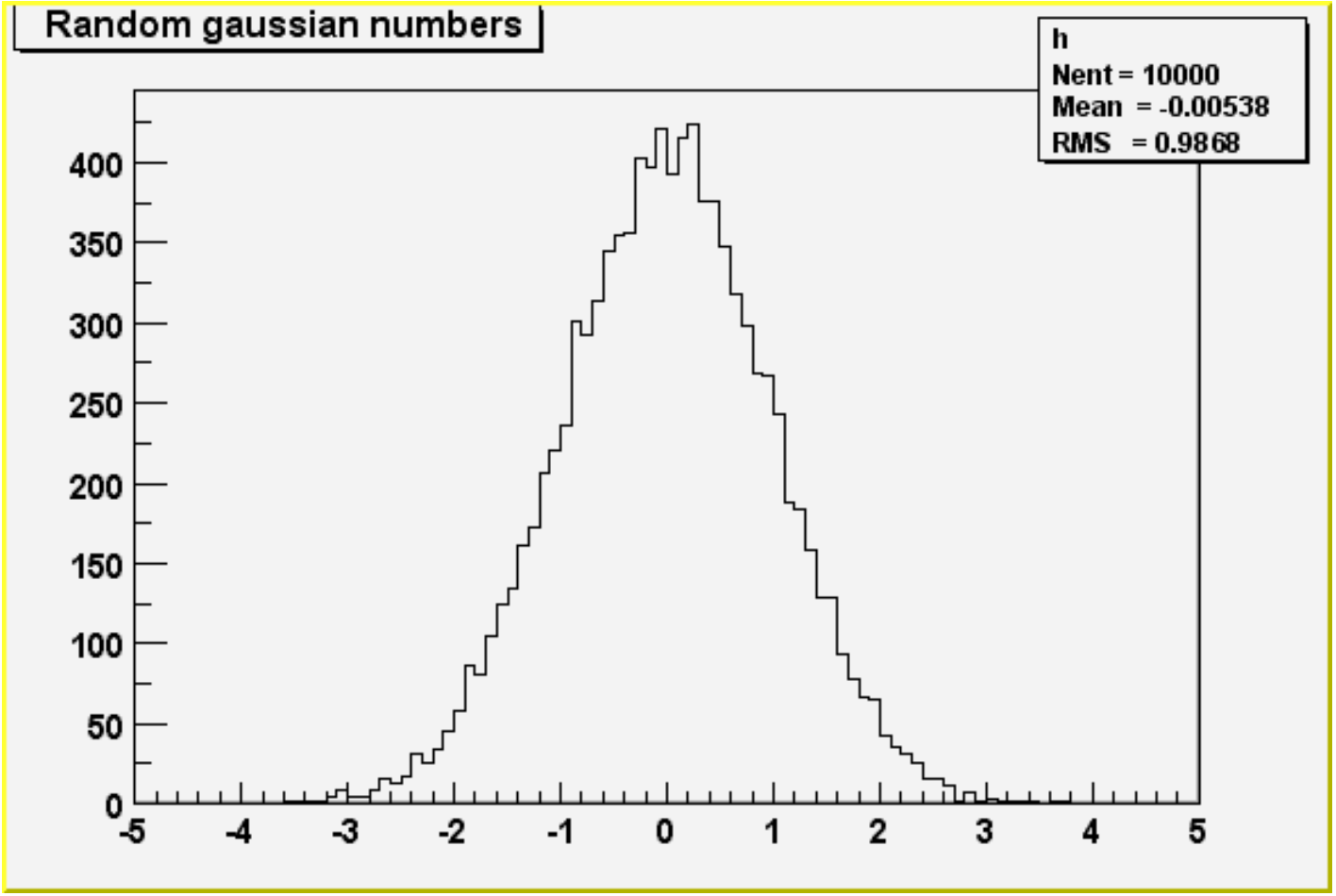
The histogram is drawn inside a Tpad and within a Tcanvas.
These are graphical objects used to store
Multiple histograms in the same view:

```
Tcanvas *c1 = new Tcanvas("c1","Canvas with hitos");
c1->Divide(1,2);
c1->cd(1);
h1->Draw();
c1->cd(2);
h2->Draw();
```

It is possible to superimpose histograms with:

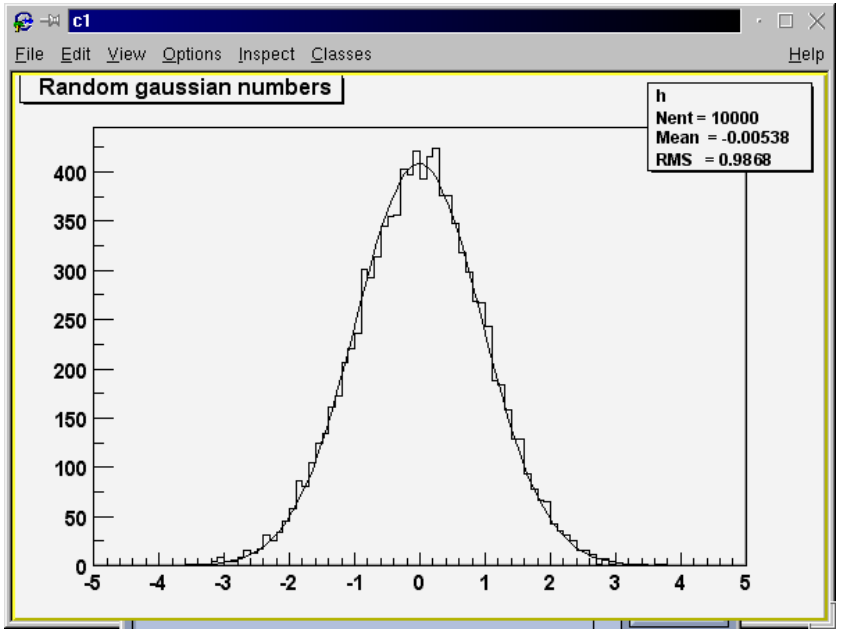
```
h1->Draw();
h2->Draw("same");
```

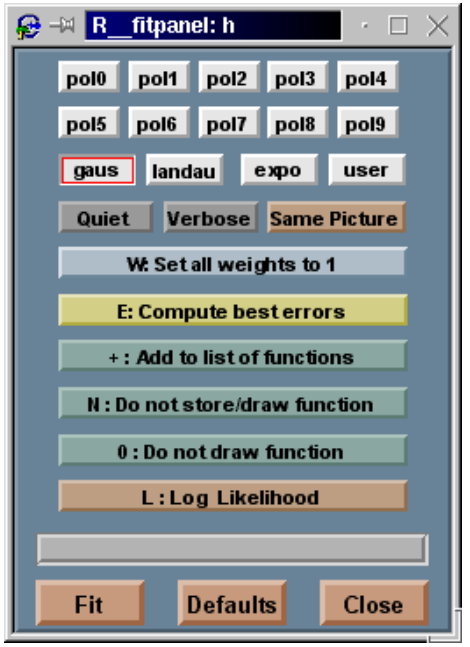
A sample 1-D histogram



Fitting Histograms

ROOT provides a graphical interface to fit 1D histograms according to well known and very common functions: Gaussian, Landau, Polynomia and user functions.





To fit an histogram with a user function, it is necessary to write a separate function that holds the function and the parameters in an array.

Graphs

A graph is a graphical object made of two arrays X and Y and it holds the x,y coordinates of N points.

First the data vectors with N elements are defined and the the constructor is invoked:

```
Int_t N=10; Double_t x[N], y[N];
Tgraph *g1 = new Tgraph(N, x, y);
```

The graph is drawn with:

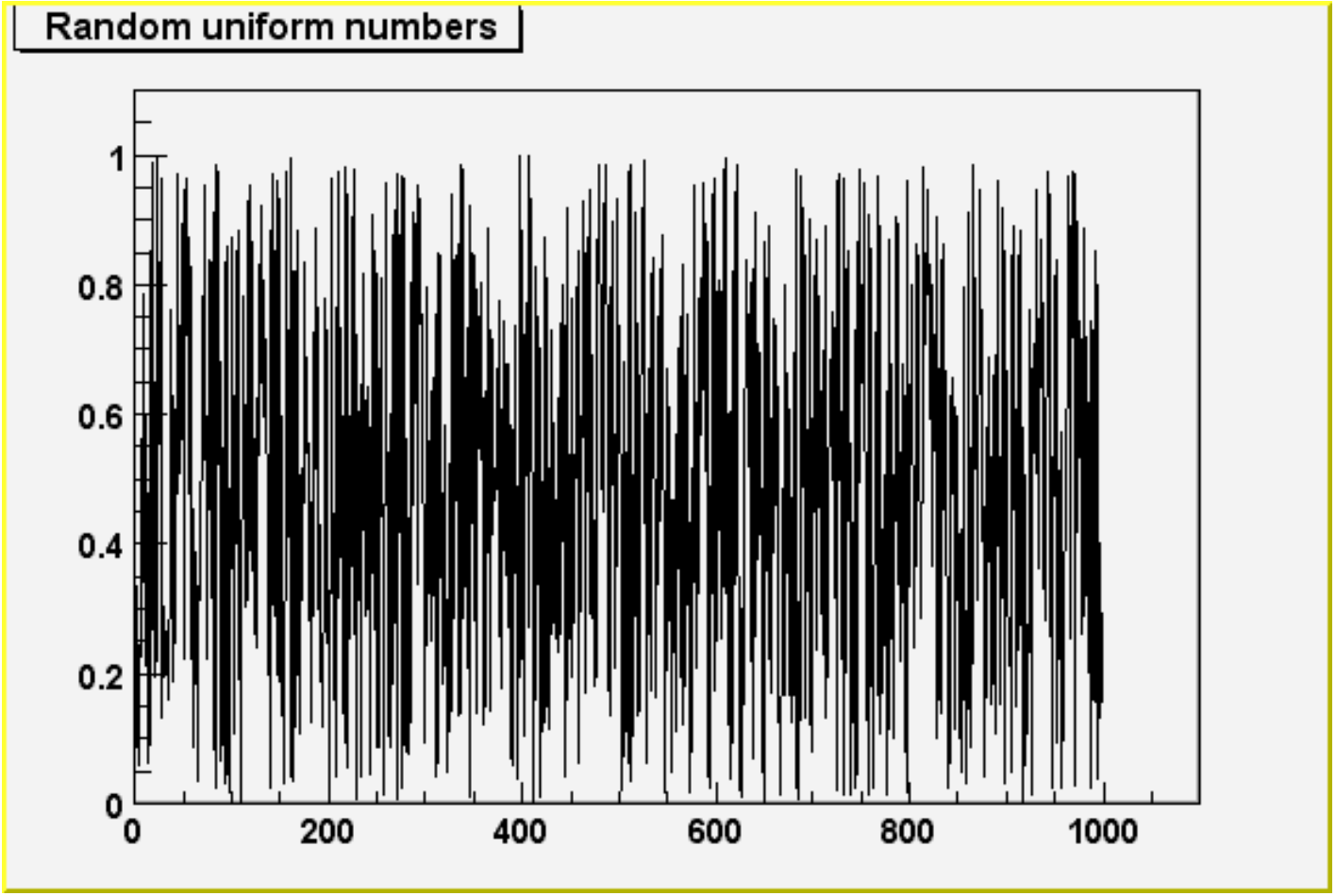
```
g1->Draw();
```

and many graphical options are available:

```
g1->Draw("AL");   axis with polyline between points
g1->Draw("A*");   star at each point
g1->Draw("AB");   bar chart
```

Graphs can be superimposed by leaving out the "A" option for the second graph or using a MultiGraph.

A sample Graph



MultiGraphs

Using a MultiGraph is a cool way to plot various graphs on the same canvas.

A TMultiGraph object is simply a collection of Tgraph objects (think of it like of an array of Tgraphs)..

Assuming to have two independent graphs:

```
TGraph *g1 = new Tgraph(N, x1, y1);
TGraph *g2 = new Tgraph(N, x2, y2);
```

we can plot them on the same sheet:

```
TMultiGraph *mg = new TMultiGraph();
mg->Add(g1);
mg->Add(g2);
mg->Draw("ALP");
```

ROOT files

ROOT objects can be saved to files for future analysis or simply for storage. A ROOT file is very similar to a UNIX file directory, with directories and objects organized in a user-decided structure and in a machine independent format (very important!).

By opening a ROOT file (in overwrite mode) with the constructor:

```
Tfile f("Test.root","recreate");
```

this file becomes the current directory and all the objects created from now on are saved into the file by invoking the respective Write method (which is a method defined for Tobject). For instance for an histogram called h, we will write it to the above Tfile by:

```
h->Write();
```

This method is not very efficient for writing MANY objects.

A sample ROOT file

We have an histogram filled with random numbers generated according to a Gaussian distribution and save it into a ROOT file:

```
Tfile f("Histos.root","recreate");
char name[10], title[20];
for(Int_t k=0; k<15; k++){
  sprintf(name,"h%d",k);
  sprintf(title,"Histo %d",k);
  TH1F *h = new TH1F(name, title, 100, -4, 4);
  h->FillRandom("gaus",1000);
h->Write();
f.Close();
```

The file can then be reopened and its contents loaded in memory for display or further use:

```
Tfile f("Histos.root");
TH1F *g = (TH1F*)f.Get("h3");
g->Draw();
```

Useful facts to know about ROOT files

• It is possible to graphically browse through a ROOT file using a Tbrowser object:

```
Tfile f("Test.root");
Tbrowser browser;
```

• A ROOT file is always compressed by default using a gzip-like compression algorithm. The default level of compression is 1, a compromise value for speed a compression efficiency. It can be changed by:

```
f.SetCompressionLevel(n)   n=0: no compression, n=9: best
```

• By closing a ROOT file, all the objects in the present directory are saved to the file and removed from the memory. Any reference to them results in an error!

```
f.Close();
h->Draw();   ERROR!
```

Saving objects to files

A Tfile provides persistency, but it is not very efficient in terms of optimizing disk space and access speed. This is particularly true when saving many objects of the same class.

Writing millions of objects to a Tfile involves writing an object header to the file which is always the same, with a great waste of time and space.

ROOT provides a solution to this problem (very common in HEP) with the Ttree class (and its simplified version Tntuple class).
A TTree provides a structure to organize data, clever enough not to duplicate the object header and optimized for data access.

These structures are filled with data and written to a Tfile, instead of writing data directly to the Tfile.
A Ttree is a true OO database for storing data and is widely used in the HEP community for saving data.

Ntuples

Ntuples are very common data structures in the HEP community to store experimental data and their use stems from the days of PAW and fortran programming.

An ntuple can be thought as a spreadsheet table in two dimensions and it is a structure optimized to collect and store numerical data into a ROOT file. It is built with the constructor:

```
Tntuple *nt = new Tntuple("ntuple","A simple table","x:y");
```

and filled (within a loop where x and y are evaluated) with:

```
ntuple->Fill(x,y);
```

It can be viewed and browsed with:

```
ntuple->Print();
ntuple->Scan();
```

Working with data in ntuples

```
root [3] ntuple->Scan()
*************************************
*    Row   *         x *         y *
*************************************
*        0 *         0 * 0.1079150 *
*        1 *         1 * 0.5874750 *
*        2 *         2 * 0.3189729 *
*        3 *         3 * 0.1464375 *
*        4 *         4 * 0.2992230 *
*        5 *         5 * 0.0407005 *
*        6 *         6 * 0.1521224 *
*        7 *         7 * 0.9564707 *
*        8 *         8 * 0.4798908 *
*        9 *         9 * 0.5815624 *
*       10 *        10 * 0.9404398 *
*       11 *        11 * 0.2410030 *
*       12 *        12 * 0.8393390 *
*       13 *        13 * 0.3083940 *
*       14 *        14 * 0.4706811 *
*       15 *        15 * 0.4858036 *
*       16 *        16 * 0.9754050 *
*       17 *        17 * 0.2558995 *
```

The Scan() method allows to browse through all the ntuple entries.

It is also possible to make cuts on variables with commands like:

```
ntuple->Scan("x","y>0.5");
```

It is possible to directly draw histograms of data entries in ntuples by invoking the Draw() method for the ntuple. A default histogram called "ht emp" is drawn on a default canvas:

```
ntuple->Draw("x");
```

Trees

A Tree is a key concept in ROOT and it also give the name to ROOT, along with Tbranch, Tleaf…

A Ttree object is a generalization of the ntuple concept and is a structure to hold objects, both ROOT objects and user-defined objects.

A Ttree is divided in branches, one branch for each object and when it is filled the branc buffers are filled first and written to the file when the buffer is full.

Ttrees are also optimized for access to data by using a hierarchy of branches that can be accessed individually.

Important: a Ttree is not editable! Once written, no data cannot be updated and it can only be read.

TBranches

The Ttree basic entry is a Tbranch object, that creates a "column"(ve ry loosely speaking) in our table. A Tbranch can hold an entire object, a list of variables or even an array of objects.
Tbranches can be built in three ways:

1) with a list of variables from a C-like structure (see ntuples)

```
tree->Branch("Data Branch",&data,"x/D:y/D");
```

2) with an object (like a TVector3):

Most common

```
TVector3 *v = new TVector3();
tree->Branch("Data Branch","TVector3", &v,64000,0);
```

3) with an array of objects (like a TClonesArray of TVector3s):

```
TClonesArray ca;
tree->Branch("Data Branch","TVector3", &ca,64000,0);
```

TTrees of objects

The Tbranch method stores objects constructed according to a given class in a branch:

```
TVector3 *v = new TVector3();
tree->Branch("Data Branch","TVector3", &v,64000,1);
```

Parameters to the Branch() method are:
- branch name
- the class name
- the address of a pointer to an object
- the buffer size (in bytes)
- a flag that indicates the split mode (in split mode=1 the data members (leaves) are saved in separate sub-branches, while in split mode=0 the object is saved as an entity and not broken down to elementary types).

The tree is then filled (within a loop where v contains data):

```
tree->Fill();
```

The 5 steps to build a tree

There is traditionally a 5 steps recipe to build a tree, fill it with data and store it into a ROOT file:

1) Create the file
2) Create the Ttree
3) Add Tbranches to the Ttree
4) Fill the Ttree
5) Write the Ttree to the TFile

How to read a tree written to a file?

```
TFile f2(filename);
T2=(TTree *)f2.Get("T2");

TBranch *gb = T2->GetBranch("fEvent");
gb->SetAddress(&fEvent);
for(Int_t ev=0;ev<T2->GetEntries();ev++) {
      gb->GetEvent(ev);
}
```

Collection classes most used

In C++ arrays contain collections of basic data types or objects and can be static or dynamic:

```
TVector3 v1[10];
TVector3 *v2 = new TVector3[10];
```

ROOT provides objects to go beyond C++ arrays in terms of memory access speed, storage efficiency and special features, like automatic resizing on demand:

- TObjArray
- TClonesArray
- TList
- TIterator

} *Beyond the scope of these lessons...*

TObjArray and TClonesArray are very common and very good alternatives to regular arrays.

TObjArrays

- TObjArray

  It holds an array of objects, in any number, but can be resized at runtime if new objects need to be added.

```
TObjArray a;
a.AddLast(TVector3 c);     automatically expands
```

  The [] operators are overriden and C-like array syntax is allowed:

```
a[25];        is equivalent to        a.At(25);
```

- TClonesArray

  It is more efficient in holding many identical objects by optimizing the way data are stored into memory.
  This is very useful for repetitive data analysis tasks, where the same objects are created and deleted many times in a loop.

Physics

ROOT was originally designed to be an analysis program, i.e. as a tool for displaying real or montecarlo data and to analyze them.

Over the years and after the support of many happy users, ROOT has been extended to include new classes that allow ROOT to be used as a Physics computing environment and as a simulation tool.

The most important ones are:

- *TVector3*
- *TLorentzVector*
- *TPhaseSpace*
- *TParticlePDG*
- *TGeant3*

TVector3

A TVector3 object represents a three dimensional vector that can be used in many ways. All the basic operations to 3D vectors are provided as class methods.

```
TVector3 a(1.,2.,3.);
TVector3 b(4.,5.,6.);

Double_t m  = v.Mag();      // get magnitude (m=Sqrt(x*x+y*y+z*z)))
Double_t m2 = v.Mag2();     // get magnitude squared
Double_t t  = v.Theta();    // get polar angle
Double_t ct = v.CosTheta(); // get cos of theta
Double_t p  = v.Phi();      // get azimuth angle
Double_t pp = v.Perp();     // get transverse component
Double_t pp2= v.Perp2();    // get transvers component squared

s = v1.Dot(v2);             // scalar product
s = v1 * v2;                // scalar product
v = v1.Cross(v2);           // vector product

v2 = v1.Unit();             // get unit vector parallel to v1
v2 = v1.Orthogonal();       // get vector orthogonal to v1
```

Our simple Vector3 class has been inspired by TVector3!

TLorentzVector

A TLorentzVector object represents a 4D vector used for relativistic and kinematics computations, both in spacetime (x,y,z,t) and in momentum space (px,py,pz,E). The metric is always (-,-,-,+).
It is implemented as a TVector3 and a Double_t variable.

```
TLorentzVector x(1.,2.,3.,4.);
TLorentzVector y(TVector3(5.,6.,7.),8.);

x.X() // access the x component
x.T() // access the y component
x.Px() // access the x component
x.E() // access the y component

v.SetVect(TVector3(1,2,3));
v.SetXYZT(x,y,z,t);
s  = v1.Dot(v2);     // scalar product
s  = v1*v2;          // scalar product
s2 = v.Mag2();   or    s2 = v.M2();
s  = v.Mag();          s  = v.M();
TVector3 b;
v.Boost(bx,by,bz);
v.Boost(b);
b = v.BoostVector();   // b=(x/t,y/t,z/t)
```

Both notations can be used!

TPhaseSpace

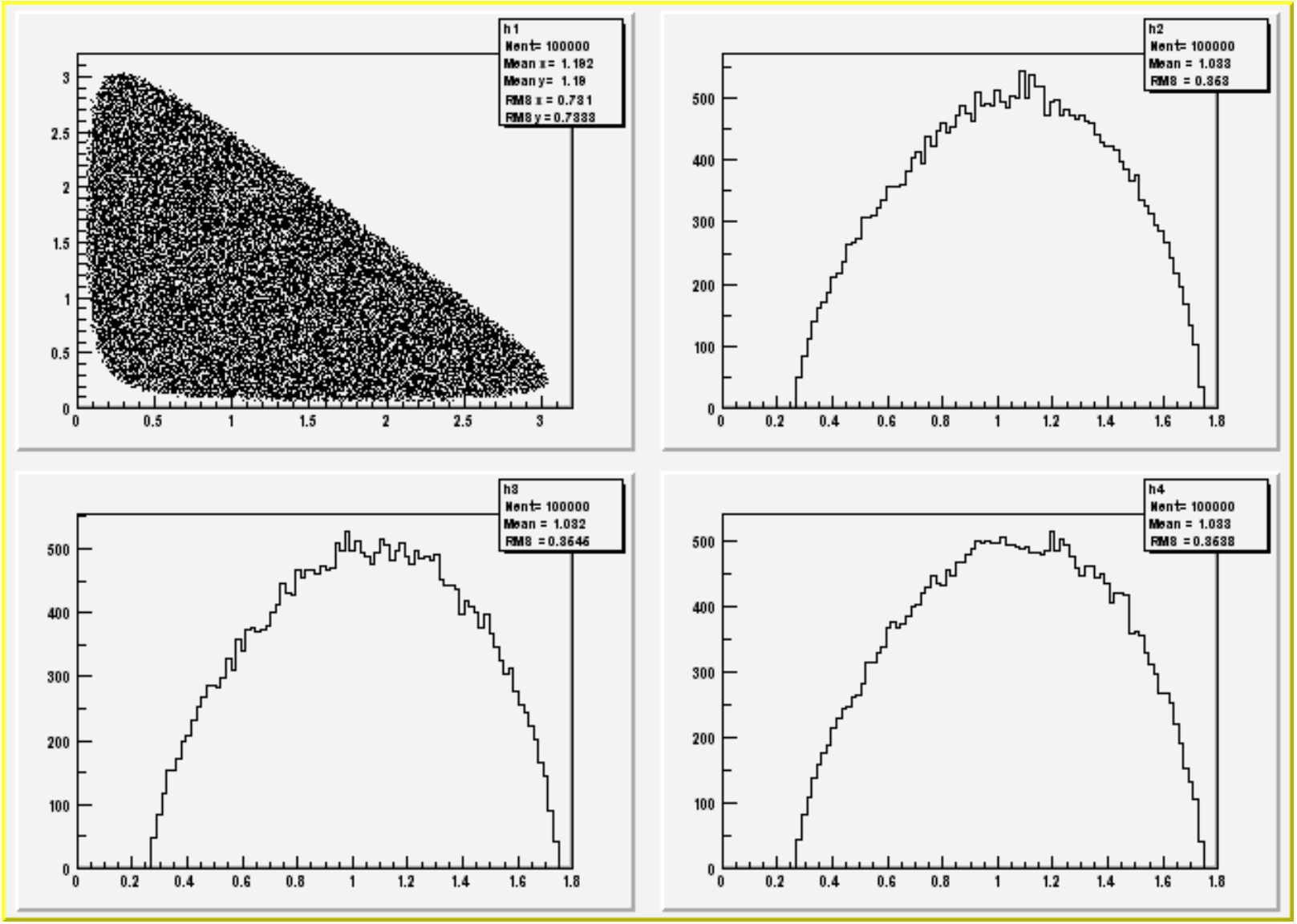
```
TLorentzVector P(0.,0.,0.,1.876);
TLorentzVector p1,p2,p3,p12,p13,p23;
Int_t n=3;
Double_t m[3]={.139,.139,.135};
Double_t m12,m13,m23;

TGenPhaseSpace S;
S.SetDecay(P,n,m,"default");

Double_t w;
for(int i=0;i<100000;i++)
{
        w=S.Generate();
        p1=S.GetDecay(0);
        p2=S.GetDecay(1);
        p3=S.GetDecay(2);
        p12=p1+p2;
        p13=p1+p3;
        p23=p2+p3;
        m12=p12.M2();
        m13=p13.M2();
        m23=p23.M2();
}
```

A phasespace generator for N bodies events (by V. Filippini) is available in the latest ROOT release (3.00).

It generates kinematics events for decays of a particle into N particles with given masses and initial momentum (example: pbar p -> pi+ pi- pi0).

The event is generated in the center of mass frame, but the decay products are finally boosted using the betas of the original particle.

Dalitz Plots



TParticlePDG

Interesting class to access the PDG database for retrieving information on particles properties, quantum numbers, decay channels and branching ratios. The all PDG booklet is provided with ROOT in a ASCII file called “ dg_table.txt”.

A TDatabasePDG object is used to store the database in memory, where it can be accesses by creating particle objects with the TParticlePDG class. Particles are retrieved with the GetParticle() method and data can be inspected with other methods:

```
TDatabasePDG *pdg = new TDatabasePDG();
pdg->ReadPDGTable("/usr/local/root/etc/pdg_table.txt");

TParticlePDG *part1 = new TParticlePDG();
part1 = pdg->GetParticle("rho0");
Double_t m = part1->Mass();
Double_t w = part1->Width();

part1->PrintDecayChannel(part1->DecayChannel(0),"");
```

Graphical Interfaces

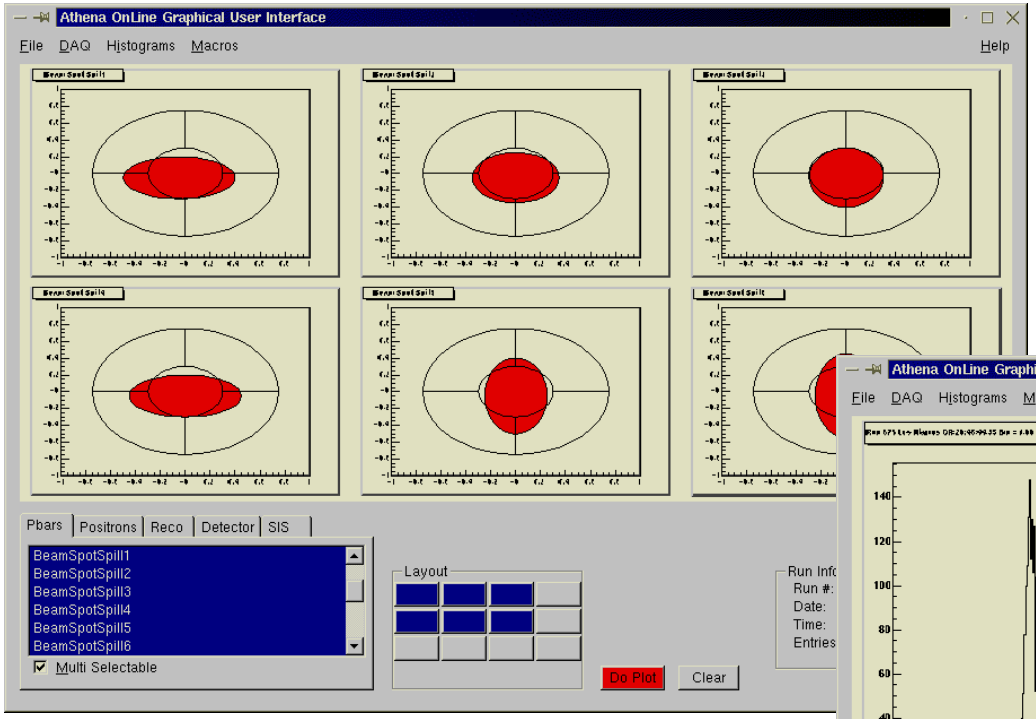
Most operating systems have a graphical interface for the user to interact (see windowz and Linux/X11). Consequently modern programs also have a graphical interface with menus, panels and buttons for running different routines and different tasks.

ROOT provides a very extensive libraryu of these classes based on the class Xclass (that mimics the windowz look and feel).

It is beyond the scope of this seminar to go into details here, but we only mention one class, the TGTextButton, which creates a button that the user can click to execute a method:

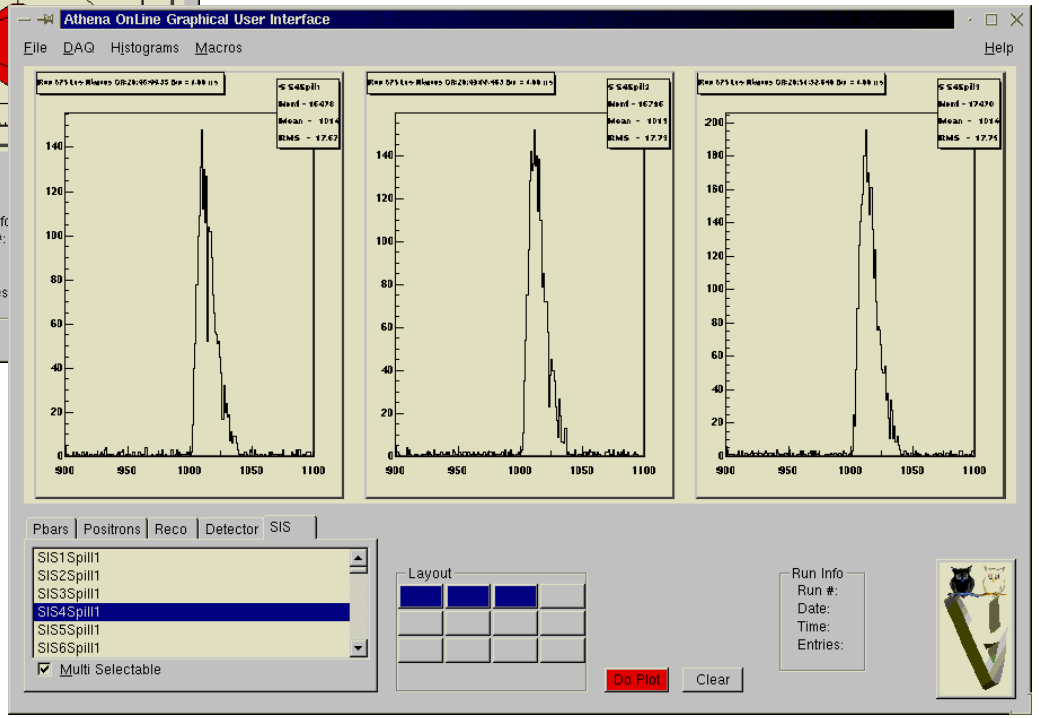
Clicking on the buttons will execute two different methods.

Example: the Athena OnLine GUI



Six beamspots as seen on the BCs.

Three hot pbars peaks at different shots as seen on the HPDs.



Extending ROOT

Creation of a ROOT dictionary allow to access the ROOT advanced I/O functions by adding extra member functions to user classes (streamers) and to add RTTI data to users programs.

RTTI means Run Time Type Information and it is a system to find out which class an object belongs, its baseclasses, its datamembers and methods, the methods signatures and other informations needed to make advanced object browsers and to use the automatic documentation generation.

To add user classes to ROOT it is necessary to add two calls to functions that links the classes to the dictionary:

- `ClassDef(ClassName, ClassVersionID);`
- `ClassImp(ClassName);`

Creating a dictionary

A ROOT dictionary is created with the rootcint program:

rootcint automatically generates new methods based on the user's class definition to supply RTTI functions and I/O functions.
Care must be taken if the class has dynamic variables, as a streamer function must be written manually.

A sample Makefile:

```
C00     = TDubnaEvent
C01     = TDubnaMC
C02     = TDubnaDisplay

ALINC   = $(C00).h $(C01).h $(C02).h
Dic.o:          $(ALINC)
         rootcint     -f  Dic.C -c -I$(GEANT) $(C00).h $(C01).h
                          $(C02).h TGeant3.h- THIGZ.h-
         $(CXX)           Dic.C -c -I$(GEANT) -o Dic.o $(CXXFLAGS)
```

Executables and ROOT scripts

Very simple way to transform a ROOT script into a real compiled executable program by using the TROOT class.

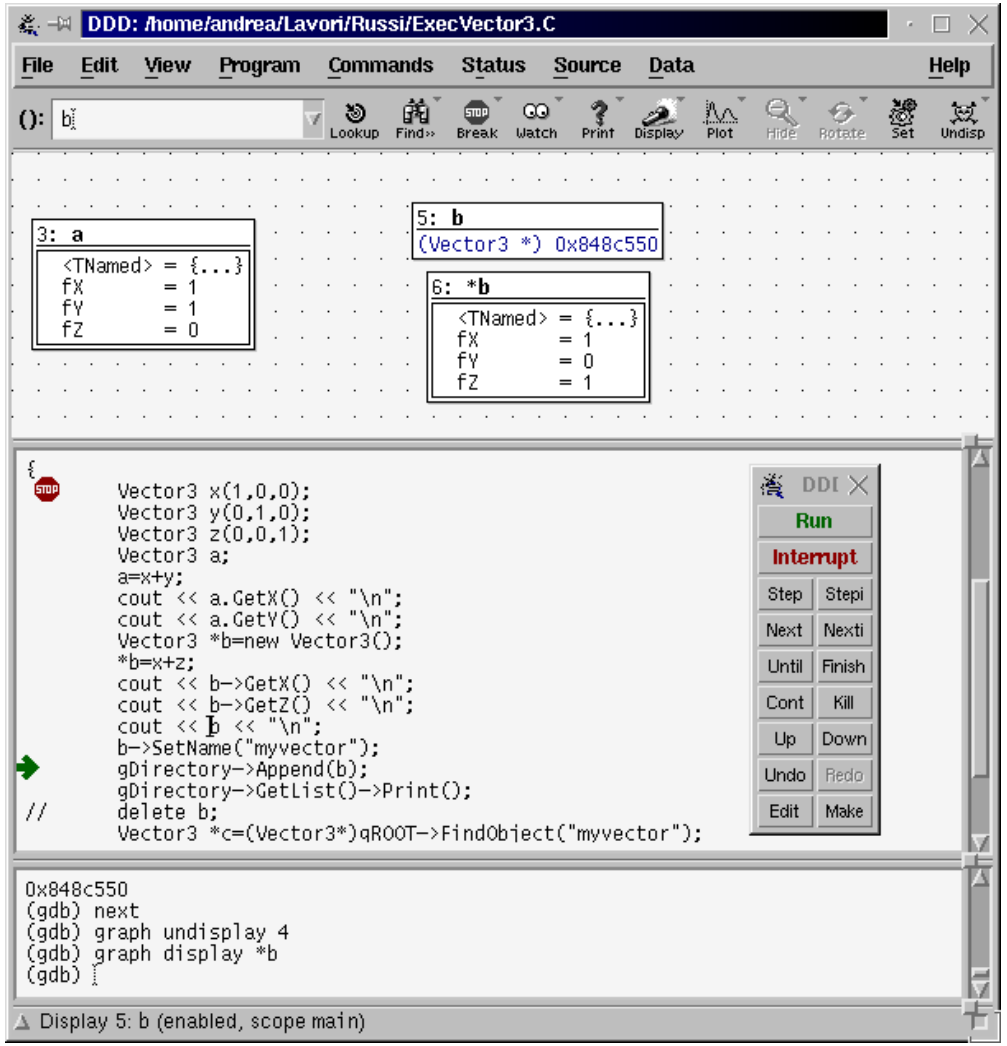
```
#include "TROOT.h"
#include "Vector3_F.h"

extern void InitGui();
VoidFuncPtr_t initfuncs[] = {InitGui,0};

TROOT root("Rint","ROOT Interface",initfuncs);

int main(int argc, char **argv)
{
        Vector3 x(1,0,0);
        Vector3 y(0,1,0);
        Vector3 z(0,0,1);
        Vector3 a;
        a=x+y;
        return 0;
}
```

The original script:

```
Vector3 x(1,0,0);
Vector3 y(0,1,0);
Vector3 z(0,0,1);
Vector3 a;
a=x+y;
```

Working with a real programs makes easier to debug it and allows to create CINT independent programs, that only use ROOT as an external library.

Debugging ROOT



*A debug session of a ROOT program with TVector3 objects: a is on the stack, *b on the heap!*

Having a ROOT executable program, it is much more easy to debug it by using standard debuggers, like gdb or ddd.

CINT has its own debug, but it is very basic and lacks a graphical interface.

The GEANT program

GEANT is a simulation program developed at CERN and used for particle detector simulations.

Tasks you are able to do with GEANT:
- *creating a detector geometry*
- *filling volumes of detector with any element*
- *tracking a variety of Standard Model particles through the detector, with any initial position and momentum*
- *retrieving data from the detector*

GEANT is originally written in Fortran 77 and many subroutines are available: some are hidden to the user, but other have to be written and customized.

A ROOT interface to GEANT has been developed by the Alice and Athena Collaborations which allow to call the fortran routines from within ROOT and uses ROOT objects.

The TGeant3 class

The TGeant3 class is a complete interface to the Geant3.21 package in fortran 77.

It works with a minimal CERNLIB on most Unix systems, but also uses the HIGZ package for graphics, so that a Thighz class is also provided.

The methods of the class directly call functions in the CERNLIB library libGeant321.lib, but for memory storage the ROOT containers are used, with a considerably higher computing speed: TGeant3 is almost two times faster than the old Fortran version!

It is an alternative to Geant 4 *(still in development)* and at the moment is not officially included in ROOT, as only Alice and Athena make use of it.

GEANT flow chart

GEANT comprises many routines, but the user only need to know a few of them.

The basic flow of a GEANT program can be summarized in three steps: *initialization*, *event processing* and *termination*.

- *Initialization* is done in the Uginit() function and involves the definition and positioning of volumes and materials/media and the loading in memory of physical data (like cross sections).

- *Event processing* is the core of GEANT: particles are tracked through the media and interactions are generated according to the media, energies and cross sections involved. Functions used are Gukine() and Gustep().

- *Termination* is done with the user routine Uglast().

GEANT scheme

- *Initialization*
  - UGINIT
    - Creating media
    - Creating volumes
    - Positioning volumes
- *Event processing*
  - Kinematics
    - GUKINE
  - Tracking
    - GUSTEP
- *Termination*
  - UGLAST

If a subroutine's name begins with GU GEANT always call it automatically, otherwise it needs to be called by the user.

Creation of Materials and Media : Uginit()

The first step is to define material and media through which track particles. GEANT distinguishes between materials and media: a material is defined by specifing the physical properties of what fills the volume of the geometry (looking into the GEANT database), while a medium is a material with a specific property, like a magnetic field.

The routines used to create aluminum are:

```
Gmate();
Medium(Nalu, "ALU", 9, 0, 3, 10, 0, 0, 0, 0, 0, NULL, 0);
```

The (important) arguments are:
- tracking medium number assigned `Nalu`
- tracking medium number `"ALU"`
- material number `9`
- magnetic field (kilogauss) `3`
- max field value `10`

Creation of Volumes : Uginit()

Volumes are created by specifying one of several predetermined shapes: box, cylinder, sphere and cone are the main ones.

The routine used to create a volume is Gsvolu():

```
Gsvolu("CUBE", "BOX", Nalu, size, 3);
```

The arguments are:
- the volume name;
- the volume type;
- the name of the media or material previously defined;
- a 3D array specifying the volume half-lengths along the 3 axis (e.g. size={0.5,0.5,0.5} is a cube of 1cm x 1cm x 1cm);
- number of parameters in the above array:
  BOX: x, y, z
  CYL: inner radius, outer radius, half-length.

Positioning of Volumes : Uginit()

Volumes are positioned with respect to a mother volume. The origin of the mother volume is at the center of it (volumes are defined using half distances) and the volume that contains everything sets up the mother reference system.

The routine used to position a volume is Gspos():

```
Float_t size0[] = {50,50,50};
Gsvolu("CHAN", "BOX", Nvac, size0, 3);   // primary volume
Float_t size1[] = { 0.5, 0.5, 5};
Gsvolu("CUBE", "BOX", Nalu, size1, 3);   // Cube
Gspos("CUBE", 1, "CHAN", 0, 0, 0, 0);    // Position
```

with arguments:
- daughter volume name
- number of daughters
- mother volume name
- X,Y and Z position of the daughter center in the mother reference
- rotation matrix (0 if not rotated).

Kinematics Initialization: Gukine()

The initial configuration for all the particles involved in the simulation is set up in the Gukine() method, which interfaces to the Gsvert() and Gskine() fortran functions.

These functions track a given particle from a given vertex and with a given momentum in space. A photon, generated in {0,0,0} with 1 GeV/c momentum along z, is generated with:

```
Float_t v[3] = { 0, 0, 0 };
Int_t nv = Gsvert(v,0,0,0,0);

Float_t p0[3] = { 0, 0, 1 };
Gskine(p0 ,1 ,nv ,NULL ,0);
```

Different particles have different (PDG) code numbers:
1: photon (22)
2: positron (-11)
3: electron… (11) (different from standard GEANT codes)

Interactions generation: Gustep()

GEANT tracks particles by calculating the probability of all possible (known) physics events and then, using a random numbers generator, decides what happens to the particle next.

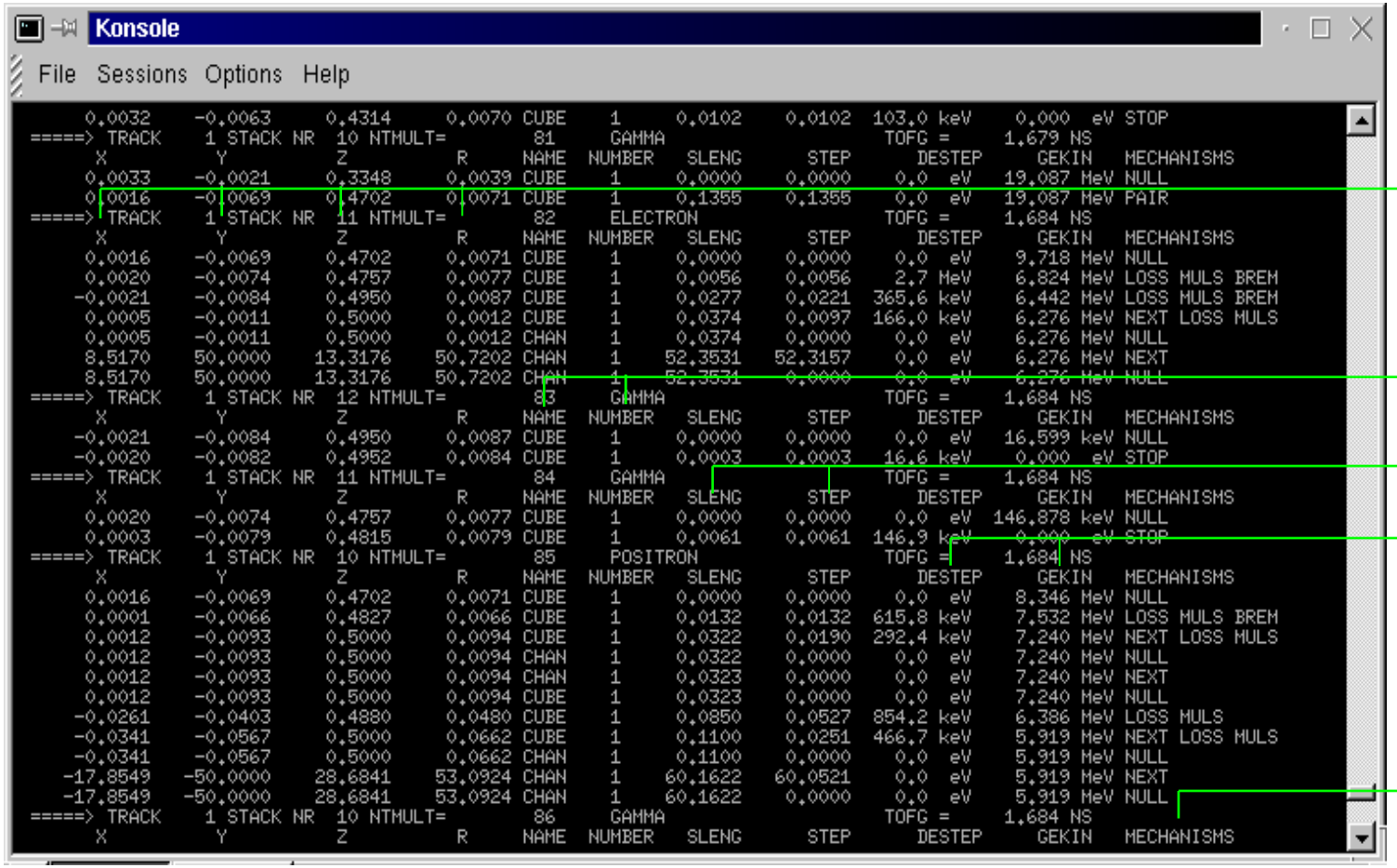
Each time something happens (I.e. the particle interacts) GEANT calls Gustep() and that step is done. The user function Gustep() is called whenever a particle looses energy.

Main operations performed by Gustep(): store TVector3 interaction points and energy loss in volume.

```
Gpcxyz();              // prints tracking and physics parameters
TLorentzVector x;
TrackPosition(x);      // return the current position of the track being
                          transported
Edep();                // return the energy lost in the current step
```

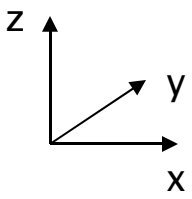
Understanding tracking

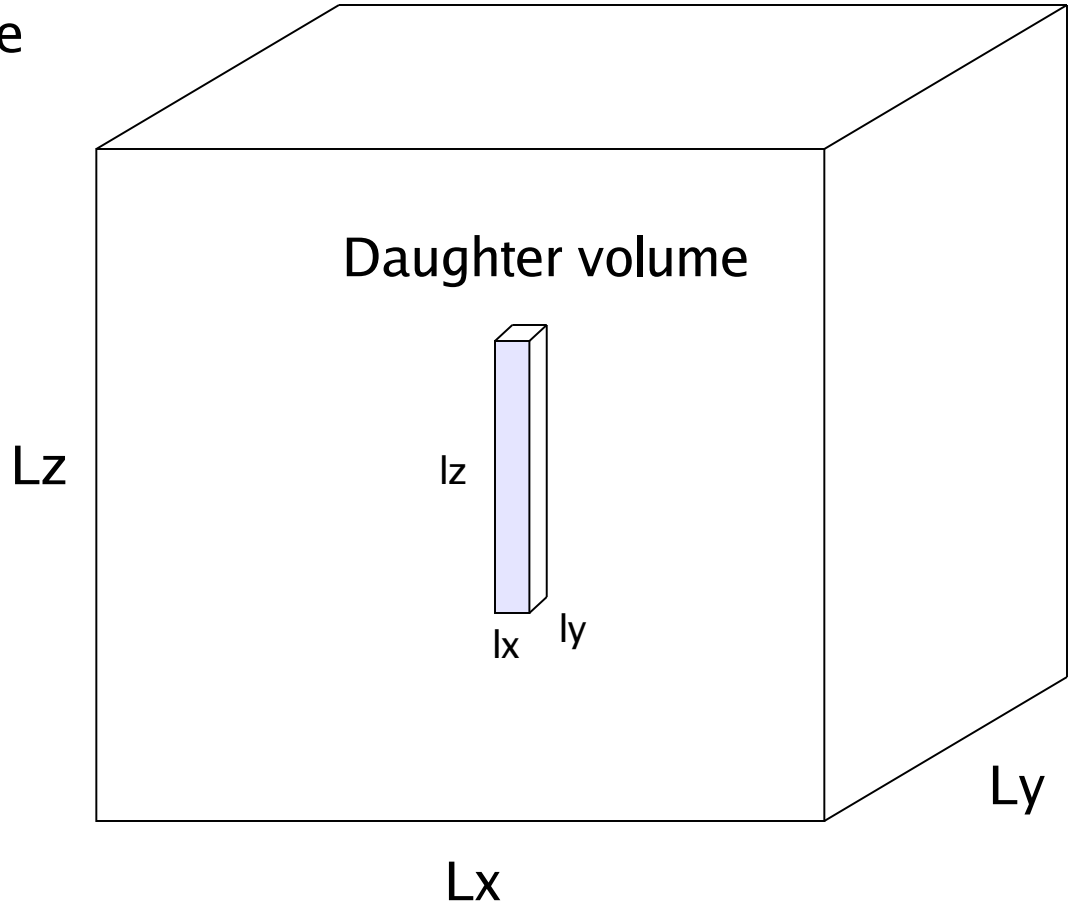
The function Gpcxyz() prints out tracking information and physical quantities relevant for understanding the interactions generated by GEANT. We can have a look at a typical output:



Geometry Definition

Mother space and reference system.





Photon prop. direction: z axis.

Volume materials:

- *aluminum*
- *lead*
- *uranium*

Not to scale

User classes

To use GEANT via ROOT for building our simple calorimeter we need to extend ROOT by defining a few new classes and by adding them to the ROOT dictionary.

There are many ways to do this and for this example we add three new classes:

- TDubnaEvent:
  Stores in memory information from tracking.

- TDubnaMC:
  Deals with the GEANT 3.21 interface.

- TDubnaDisplay:
  Graphically displays the detector's volumes and the tracks of the simulated particles.

The TDubnaEvent class

We want to study the interaction of photons within materials by:

- Tracking all the particles produced by Compton, photoelectric and pair creation processes and by all the subsequent interactions (bremstralhung);

- Recording the total energy deposed in a given volume of material.

Our class must contain a TObjArray of TVector3 objects to store the 3D coordinates of all the interaction points and a Double_t variable to store the energy deposed at a given point.

The TDubnaEvent class (.h)

```
class TDubnaEvent : public TObject {
 private:
  TObjArray    *fMCPoint;
  Double_t      fADC;

 public:
  TDubnaEvent();
  virtual ~TDubnaEvent();
  void DeleteEvent();

  void  AddMCPoint(TVector3 *v) { fMCPoint->Add(v); }
  Int_t GetMCNumPoint() const   { return fMCPoint->GetLast()+1; }

  TVector3 *GetMCPoint(Int_t n) const
  {
     return (TVector3*) fMCPoint->At(n);
  }
  Double_t  GetAdc() { return fADC; }
  void SetAdc(Double_t ADC) { fADC += ADC; }

  ClassDef(TDubnaEvent,1)
};
```

The TDubnaEvent class (.C)

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include "TVector3.h"
#include "TDubnaEvent.h"
ClassImp(TDubnaEvent);

TDubnaEvent::TDubnaEvent()
{
fMCPoint = new TObjArray();
fADC=0;
}

TDubnaEvent::~TDubnaEvent()
{
DeleteEvent();
Delete fMCPoint;
}

void TDubnaEvent::DeleteEvent()
{
if (fMCPoint) fMCPoint->Delete();
fADC = 0;
}
```

The TDubnaMC class

Deals with the interface to GEANT and contains the following methods:

- Uginit():

  Defines and positions the volumes and fills them with a material (magnetic field can be ON and OFF):
  - CHAN: mother volume
  - CUBE: interaction region

- Gukine() and Gustep():

  Defines the initial photon kinematics and generates step by step interactions and produced particles. Store coordinates and energies for display.

- Uglast():

  Terminates computation and release memory.

The TDubnaMC class (.h)

```
class TDubnaMC : public TGeant3 {
 private:
  TDubnaEvent   *fEvent;

 public:
  TDubnaMC() {}
  TDubnaMC(TDubnaEvent **e);

  virtual ~TDubnaMC() {}
  virtual void UgInit(Bool_t field);
  virtual void UgEvent() { Gtrigi();  Gtrig();  Gtrigc(); }
  virtual void UgLast()  { Glast(); }
  virtual void GuKine();
  virtual void GuStep();
  virtual void GuOut();

  ClassDef(TDubnaMC,1)
};
```

The TDubnaDisplay class

Deals with graphical visualization of the simulated interactions and tracks by using the information we stored in TDubnaEvent.

It contains a Tcanvas and Tpad for the display and a copy of the geometry that is used for the display only: the geometry defined in Uginit() is only used for the actual computation. This is a f eature" of the present version of TGeant3: you have a double version of the geometry, but this might change in the future.

The class then loops over the Tobjarray of TVector3 objects to get all the interaction point coordinates and links them with TPolyLine3D objects to draw tracks in different colors.

The viewpoint can be changed and also a X3D display is automatically generated for interactive inspection.

The TDubnaDisplay class (.h)

```
class TDubnaDisplay : public TNamed {
 private:
  TCanvas    *fCanvas;
  TPad       *fPad;
  TGeometry  *fDubnaGeom;

  Bool_t fViewMCData;
  TDubnaEvent **fEv;

 public:
  TDubnaDisplay() {}
  TDubnaDisplay(TDubnaEvent **e);
  virtual ~TDubnaDisplay();
  virtual void DrawView(Float_t theta, Float_t phi,
                        Float_t psi, Option_t *title=NULL);
  virtual void DrawAllViews();
  virtual void SetView(Float_t theta, Float_t phi, Float_t
                       psi, Option_t *title=NULL);
  virtual void DrawViewX3D();

 ClassDef(TDubnaDisplay,0)
};
```

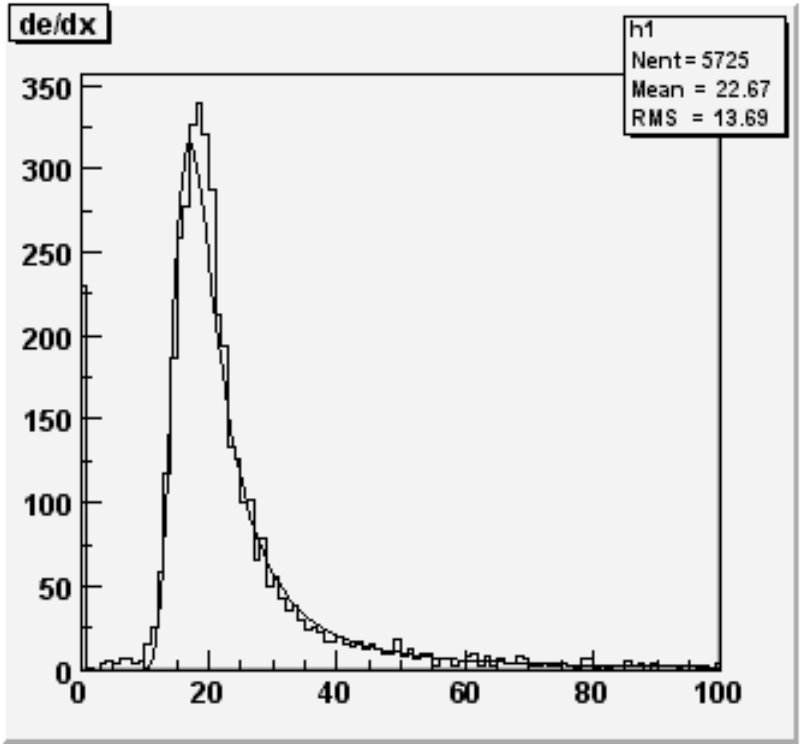
dE/dx distributions

To check if we have done things correctly, we can follow for instance all the electrons produced by a 1 GeV photon hitting on uranium and measure the dE/dx distribution.

We can do this by using two methods of TGeant3:

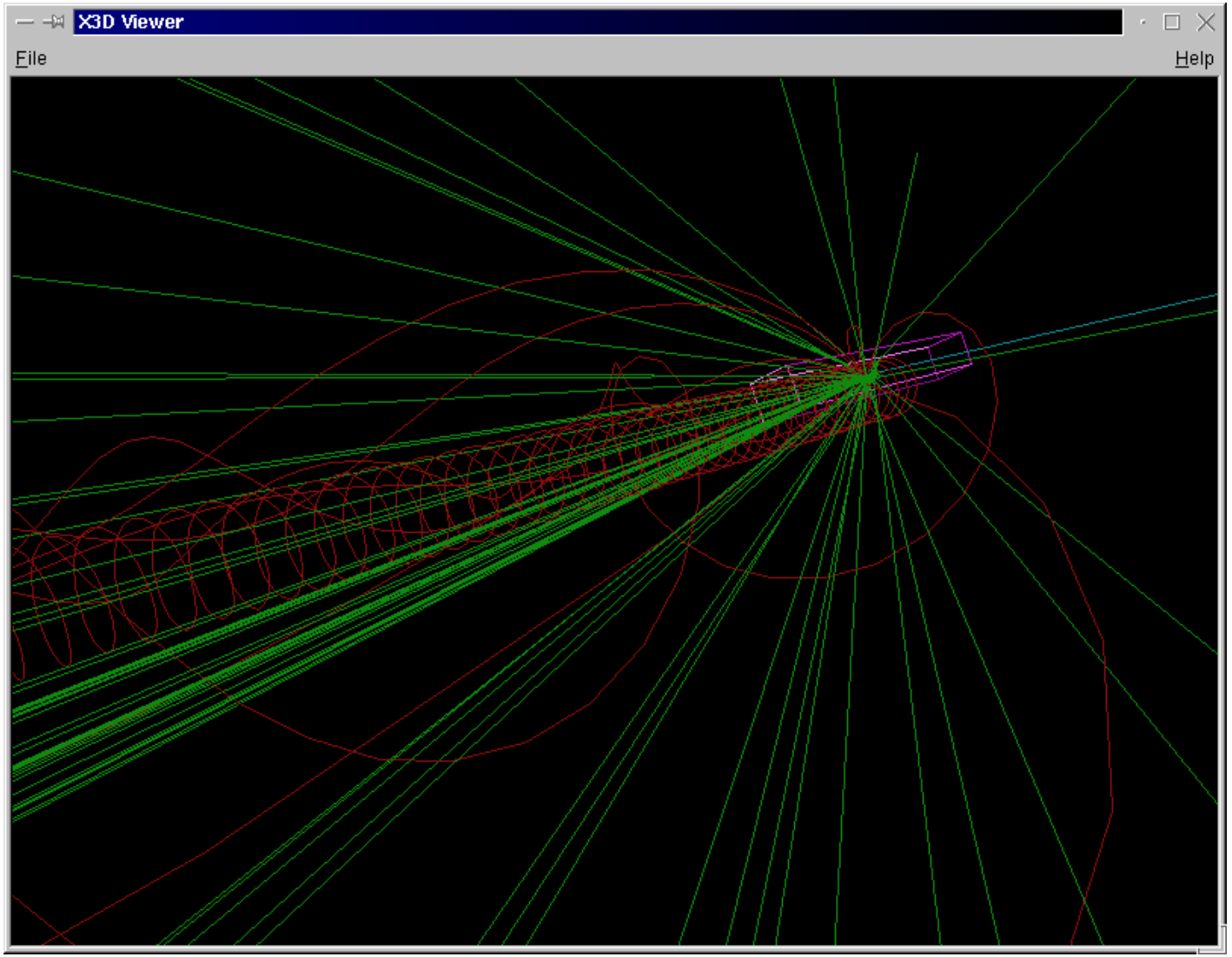
```
Edep()/TrackStep()
```

and by histogramming the result.



This is the resulting distribution fitted with a Landau!

A TDubna Event in magnetic field



The Athena Project Software

The Athena software is structured on different levels:

- Data Acquisition (LabView)

  Detectors readout and data storage (raw data files)

- OnLine software (ROOT)

  Raw files decoding, data cooking and experiment real time monitor

- OffLine software (ROOT/TGeant3)

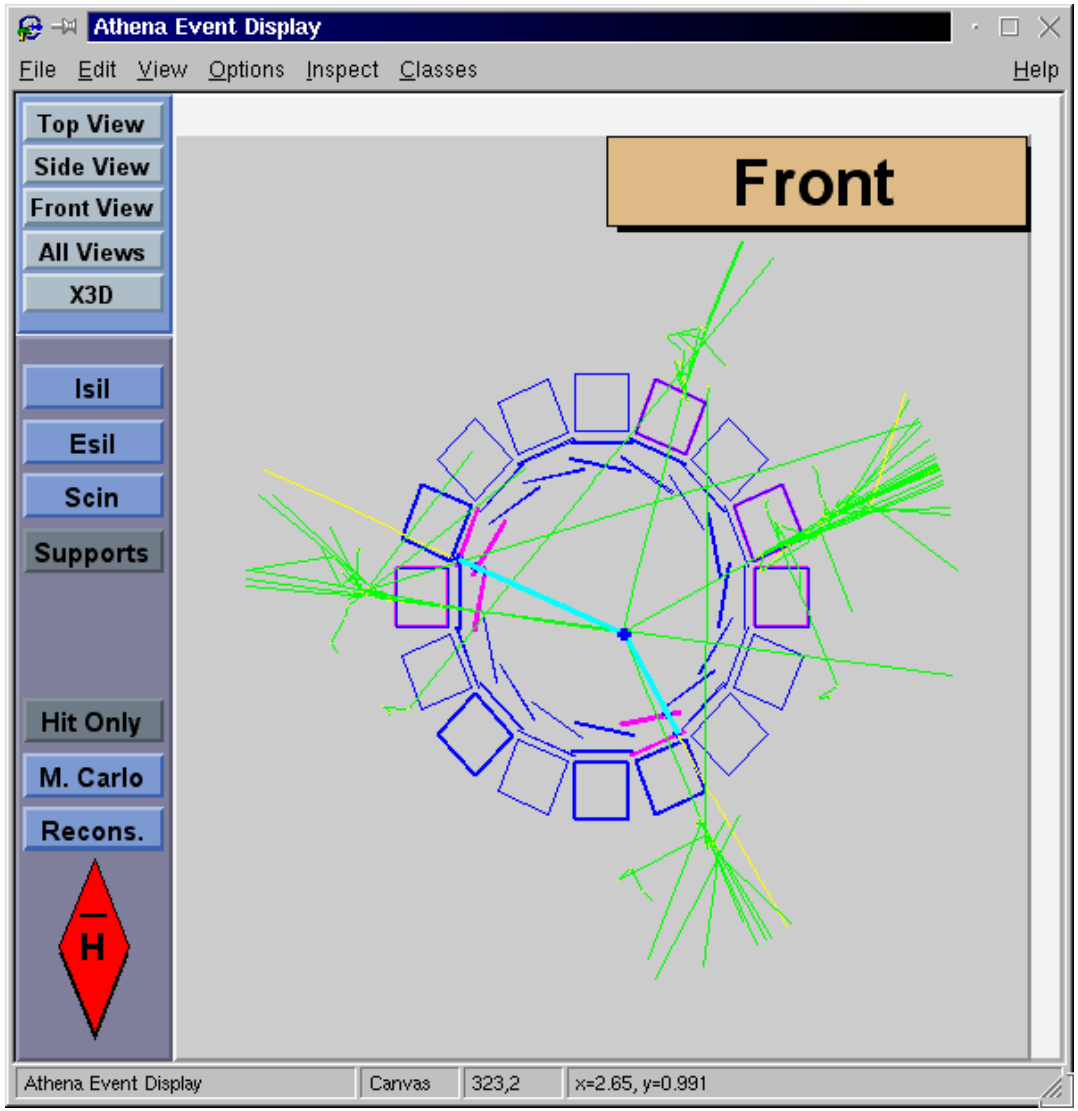
  Central detector event display, reconstruction, pattern recognition and MC.

The Athena OffLine software

The Athena OffLine software is based on various classes:

- TAthenaEvent
  base class for tracking and hit detectors and for reconstruction;
- TAthenaEventCsI
  base class for tracking in Crystals;
- TAthenaEventSil
  base class for tracking in Silicons;
- TAthenaMC
  Monte Carlo interface to TGeant3;
- TAthenaRealData
  Interface to data from OnLine monitor;
- TAthenaDisplay
  Detector display class.

The Athena Event Display



Monte Carlo simulation of an anti-hydrogen candidate event in the Athena detector.

The Athena OffLine software is ROOT based and extensively uses the TGeant3 class.

Also the reconstructed vertex is shown.

Summary

ROOT is a powerful tool for the modern HEP physicist both for analysis and for simulations.

The TGeant3 class is a full interface to routines of the CERN library that are well known, tested and stable. Using the ROOT/GEANT interface there is no need for writing new Physics code and commercial software is not required (like LHC++/Objectivity).

Two things to remember when doing Physics with ROOT or any other OO system:

- Quantum Mechanics is more difficult and you know it already!

- Hamming's Rule:
"The purpose of computing is insight, not numbers."

Where to go from here...

- The ROOT web site:

  http://root.cern.ch

- The Alice Software and Computing group:

  http://AliSoft.cern.ch/offline/

- The Athena Web site:

  http://athena.web.cern.ch/athena/

- My web page from where to download these slides and the source code of TGeant3 and of the TDubna tutorial:

  http://www.pv.infn.it/~fontana/TDubna