Persistency: writing information on an external file

Luciano Pandola
INFN-LNGS

Queen's University, Belfast (UK), January 24, 2013

Based on a presentation by G.A.P. Cirrone (INFN-LNS)

Introduction: data analysis with Geant4

- For a long time, Geant4 did not attempt to provide/ support **any data analysis** tools
  - The focus was given (and is given) to the central mission as a Monte Carlo simulation toolkit
  - As a general rule, the user is expected to provide her/his own code to output results to an appropriate analysis format
- A few basic classes for data analysis have recently been implemented in Geant4 (version 9.5)
  - Support for histograms and (very limited) ntuples
  - Output in **ROOT**, **XML**, **HBOOK** and **CSV** (ASCII)
  - Appropriate only for easy/quick analysis: for advanced tasks, the user must write his/her own code and to use an external analysis tool

Introduction: how to write simulation results

- Formatted (= human-readable) **ASCII files**
  - Simplest possible approach is comma-separated values (.csv) files
  - The resulting files can be opened and analyzed by <u>tools</u> such as: Gnuplot, Excel, OpenOffice, Matlab, Origin, ROOT, PAW, …
- **Binary files** with complex anlysis objects (Ntuples)
  - Allows to control what plot you want with modular choice of conditions and variables
    - Ex: energy of electrons knowing that (= cuts): (1) position/ location, (2) angular window, (3) primary/secondary …
  - <u>Tools</u>: Root , PAW, AIDA-compliant (PI, JAS3 and OpenScientist)

ASCII files

Output stream (G4cout)

- **G4cout** is a **iostream** object defined by Geant4.
  - The usage of this objects is exactly the same as the ordinary **std::cout** except that the output streams will be handled by **G4UImanager**
  - **G4endl** is the equivalent of **std::endl** to end a line
- Output strings may be displayed on another window or stored in a file
- One can also use the file streams (**std::ofstream**) provided by the C++ libraries

Output on screen – an example

```
void SteppingAction::UserSteppingAction(const G4Step* aStep)
{

  evtNb = eventAction -> Trasporto();

  G4String particleName = aStep -> GetTrack() -> GetDynamicParticle() -> GetDefinition() -> GetParticleName();
  G4String volumeName = aStep ->GetPreStepPoint() -> GetPhysicalVolume() -> GetName();
  G4double particleCharge = aStep -> GetTrack() -> GetDefinition() -> GetAtomicNumber();
  G4double PDG=aStep->GetTrack()->GetDefinition()->GetAtomicMass();


  G4Track* theTrack = aStep->GetTrack();
  G4double kineticEnergy = theTrack -> GetKineticEnergy();
  G4int trackID = aStep -> GetTrack() -> GetTrackID();
  G4double edep = aStep->GetTotalEnergyDeposit();
  G4String materialName = theTrack->GetMaterial()->GetName();
```

```
G4cout      << "Energy deposited--->" <<  " " <<  edep << " "
      << ”Charge--->" <<  " " << particleCharge << " "
      << ”Kinetic Energy --->" << "  " << kineticEnergy << " "
                 << G4endl;
```

Output on screen – an example

```
---> Begin of Event: 0
Energia depositata---> 9.85941e-22 Carica--->  6 Energia Cinetica--->  160
Energia depositata---> 8.36876 Carica--->  6 Energia Cinetica--->  151.631
Energia depositata---> 8.63368 Carica--->  6 Energia Cinetica--->  142.998
Energia depositata---> 5.98509 Carica--->  6 Energia Cinetica--->  137.012
Energia depositata---> 4.73055 Carica--->  6 Energia Cinetica--->  132.282
Energia depositata---> 0.0225575 Carica--->  6 Energia Cinetica--->  132.259
Energia depositata---> 1.47468 Carica--->  6 Energia Cinetica--->  130.785
Energia depositata---> 0.0218983 Carica--->  6 Energia Cinetica--->  130.763
Energia depositata---> 5.22223 Carica--->  6 Energia Cinetica--->  125.541
Energia depositata---> 7.10685 Carica--->  6 Energia Cinetica--->  118.434
Energia depositata---> 6.62999 Carica--->  6 Energia Cinetica--->  111.804
Energia depositata---> 6.50997 Carica--->  6 Energia Cinetica--->  105.294
Energia depositata---> 6.28403 Carica--->  6 Energia Cinetica--->  99.0097
Energia depositata---> 5.77231 Carica--->  6 Energia Cinetica--->  93.2374
Energia depositata---> 5.2333 Carica--->  6 Energia Cinetica--->  88.0041
Energia depositata---> 3.9153 Carica--->  6 Energia Cinetica--->  84.0888
Energia depositata---> 14.3767 Carica--->  6 Energia Cinetica--->  69.7121
Energia depositata---> 14.3352 Carica--->  6 Energia Cinetica--->  55.3769
```

To write a new ASCII file: a recipe - 1

- Add to the include list of your class the **`<fstream>`** header file
  - This will allow to use the C++ libraries for stream on file
- Put into the class declaration (file .hh) an ofstream (=output file stream) object (or pointer):

```
std::ofstream myFile;
```

  - In this way, the file object will be visible in all methods of the class
- Open the file, in the class constructor, or into a specific method:

```
myFile.open("filename.out", std::ios::trunc);
```

  - To append data to an existing file, you must specify **`std::ios::app`**

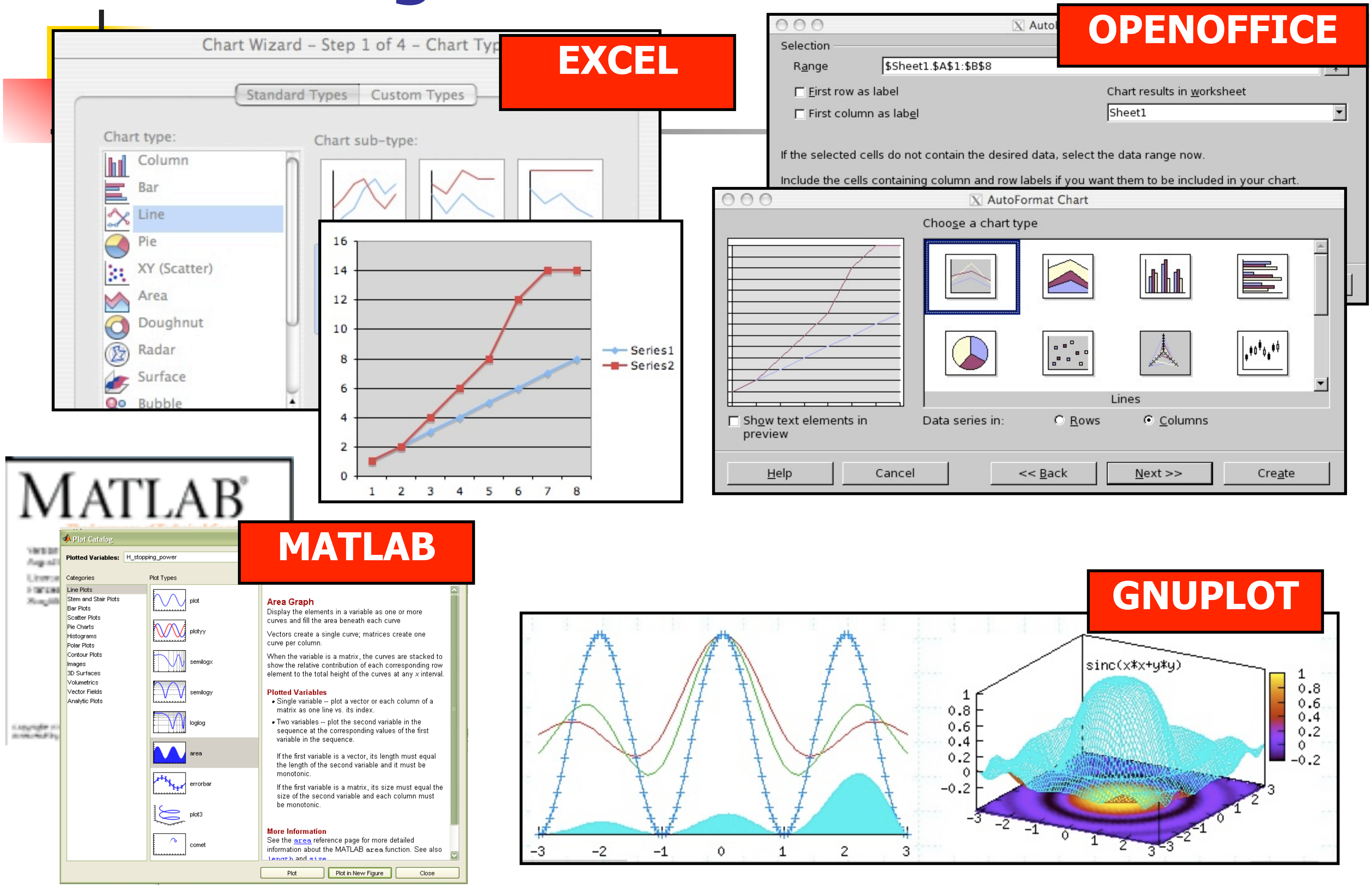
To write a new ASCII file: a recipe - 2

- Inside a regularly called method (e.g. inside a virtual method of an User Class), where appropriate, write your data (i.e. **G4double**, **G4int**, **G4String**,…) to the file, in the same fashion of **G4cout**:

```
if (myFile.is_open()) // Check that file is opened
  {
     myFile << kineticEnergy/MeV << " " << dose << G4endl;
     …
  }
```

  - This could be for instance the **EndOfEventAction()** of the **G4UserEventAction** user class
- Finally close the file, in the class destructor, or into a specific method: **myFile.close();**

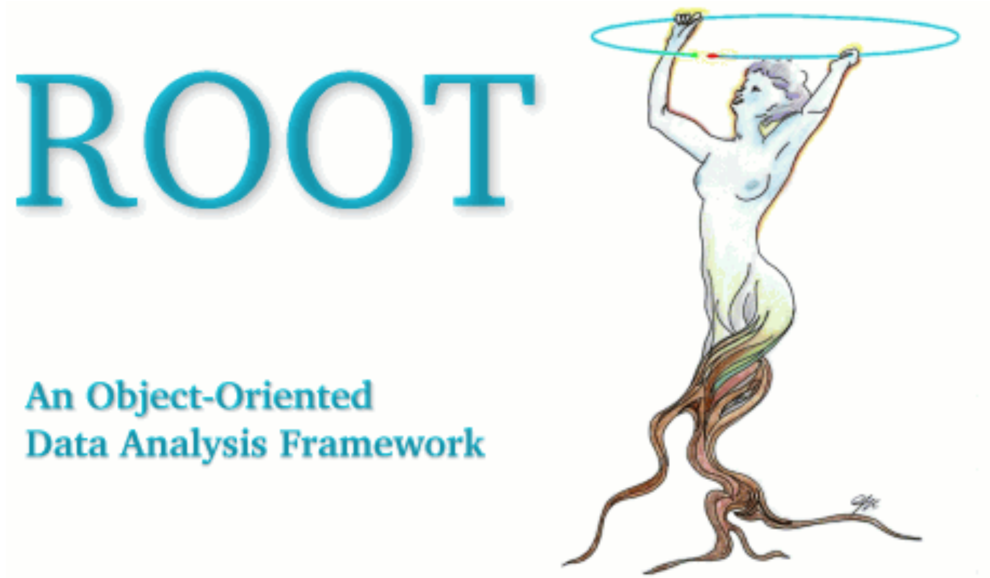
Plotting with tools



ROOT files

ROOT

- ROOT is an Object Oriented Data Analysis Framework.
- It is heavily used in High Energy and Particle Physics
- **Advanced support** for data analysis, storage and display
- Freely available
  - http://root.cern.ch/



How to compile ROOT in a Geant4 application - 1

- First of all, the compiler must know where to find the ROOT includes (.hh) and the ROOT libraries
- Easily managed by the cmake build
  - The CMakeLists.txt file must be edited like

```
find_package(ROOT)
if(ROOT_FOUND)
 include_directories(${ROOT_INCLUDE_DIR} ${Geant4_INCLUDE_DIR}
      ${PROJECT_SOURCE_DIR}/include)
 message(STATUS "ROOT found. Analysis enabled")
else()
 message(STATUS "ROOT not found. EXIT")
 return()
endif()
target_link_libraries([myexec] ${Geant4_LIBRARIES}
           ${ROOT_LIBRARIES})
```

How to compile ROOT in a Geant4 application - 2

- When launching cmake, one must specify where to find the configuration of the ROOT module
  - **-DCMAKE_MODULE_PATH**=/…/…/
  - Geant4 provides the cmake configuration of several modules (ROOT, AIDA, CLHEP, Pythia, HepMC) in the build/Modules directory
- Then add in the class header (.hh file) of specific user class(es) devoted to analysis the required ROOT include files
  - Histrograms, graphs, ntuples, etc.
  - See next slide

How to compile ROOT in a Geant4 application - 3

Mandatory headers →

```
#include "TROOT.h"
#include "TFile.h"
```

NTuples, 1-D(float) & 3-D (double) histograms →

```
#include "TNtuple.h"
#include "TTree.h"
#include "TH1F.h"
#include "TH3D.h"
```

Graphic al objects →

```
#include "TCanvas.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TLegend.h"
#include "TLegendEntry.h"
#include "TLegend.h"
#include "TStyle.h"
```

Using ROOT objects for analysis - A recipe 1

- Declare the pointers to the **ROOT** objects in your class header (.hh):
  - `TFile  *theTFile; // ROOT file`
  - `TH1F *histoEnergyDepositedPerEvent;  // 1-D histogram`
  - `TNtuple  *kinFragNtuple; // ntuple`
- Create an instance for each object in the class constructor, or in a specific method:

```
theTFile = new TFile("myFileName", "RECREATE");
```

This will create the file `myFileName.root` containing an image of ROOT variables.
The option "RECREATE" means that an existing file will be overwritten!

Using ROOT objects for analysis - A recipe 2

- An instance of each defined object can be created, in the class constructor or in a specific method called once, via the "**new**" operator:

```
// Histogram containing the energy deposited in the FIRST slice of the
// detector, at each event;
histoEnergyDepositedPerEvent = new TH1F("EnergyPerEvent",
                                        "Energy, Counts",
                                        400,
                                        50.0,
                                        70.0);

kinFragNtuple  = new TNtuple("kinFragNtuple",
                  "Kinetic energy by voxel & fragment",
                  "i:j:k:A:Z:kineticEnergy");
```

Using ROOT objects for analysis - A recipe 3

- Now you have to fill each ROOT object with the appropriate values
  - …from the appropriate place, e.g. EndOfEventAction
  - Data are temporarily written to memory, then flushed to file

```
/////////////////////////////////////////////////////////////////////////////////////
// FillKineticFragmentTuple create an ntuple where the voxel indexs, the atomic number and mass and the kinetic
// energy of all the particles interacting with the phantom, are stored
void HadrontherapyAnalysisManager::FillKineticFragmentTuple(G4int i,
                                                            G4int j,
                                                            G4int k,
                                                            G4int A,
                                                            G4double Z,
                                                            G4double kinEnergy)

{
  kinFragNtuple -> Fill(i, j, k, A, Z, kinEnergy);
}
```

← Fills one raw of the ROOT Ntuple

Using ROOT objects for analysis - A recipe 4

- At the end of the simulation (or at the end of a run) **write and finalize** the ROOT file.
  - This can be done e.g.
    - At the EndRunAction
    - In the destructor of the analysis class
    - At the end of the main program

```
//////////////////////////////////////////////////////////////////////////////////
// Flush data & close the file
void HadrontherapyAnalysisManager::flush()
{
  if (theTFile)
    {
      theTFile -> Write();
      theTFile -> Close();
    }
}
```

It's a good programming practice to check that a pointer is not NULL before using it

This will finalize and close the ROOT file, and it frees the memory

Graphics at run-time

- It is possible to create a ROOT Application Environment that interfaces to the windowing system
  - This will allow to use and display ROOT objects at run-time
    - For instance, you can see how the histrogram looks after 1000 simulated events and update it every 1000 events
- A unique **`TApplication`** object must be instantiated (for example in the main) so that ROOT will load the graphic libraries

```
TApplication myapp("myapp",0,0);
```

- Crate a ROOT **`TCanvas`** and draw the histograms (graphs, or whatever ROOT object) on it