Data Analysis in High Energy Particle Physics

Gennady Pospelov

Scientific Computing at MLZ
October, 24 2013, ILL, Grenoble, France

- Introduction
- Data acquisition in HEP
- ROOT: common framework for data analysis
- Conclusion
The instruments used are detectors installed on the accelerator.

The accelerator boosts beam of particles before they are made to collide with each other.

Detector is built around collision point.

Detector observes and records the results of collisions (signals in sub-systems caused by secondary particles).

Signals are used to reconstruct the collision picture.

https://twiki.cern.ch/twiki/bin/view/AtlasPublic/
Pertinent differences between HEP and scattering experiments:

- Number of read out channels, data rate
- Size of collaboration
- Absence of “users”

Main factors determine the evolution of the software:

- Data rate -> performance, scale issues
- Size of collaborations -> easier to build community project
- Absence of users -> no need to oversimplify the software
Data acquisition in HEP

Beam crossing every 25 ns
- Size of ATLAS event 25 Mb
- 40Mhz, 100 PB/sec

Level 1 Trigger
- Massive parallel system of detector electronics
- Filters out garbage events
- 100kHz, 160 Gb/sec

Level 2,3 triggers
- Cluster of 2300 machines
- Select events of interest
- 200 Hz, 300 Mb/s

Disk storage
- Deals with 300 Mb/sec, 1Pb/year
Data analysis in HEP

To make analysis possible raw data has to be processed
- to reconstruct objects (electrons, muons, gammas)
- to preselect events of interest and prepare compactified datasets which fits into user laptops

It is done using the detector software framework
- on ATLAS it deals with data acquisition, reconstruction and simulation
- C++/python, 6M lines of code

As soon as compact datasets are prepared, no detector framework is required. This is the point where the analysis starts.
A possible scenario

- Download required dataset using ATLAS metadata system.

- Write analysis scripts to finalize data processing, make fits and prepare plots for a publication.
- No detector software framework is used. Data is processed using ROOT framework.

Final data analysis is always done in the same way around the HEP world: using a single framework

The ROOT framework
ROOT analysis framework

ROOT is a framework for data processing used by all HEP experiments and beyond.
Maintained by CERN (http://root.cern.ch)

- Read/write complex data structures
- Process data with powerful mathematical and statistical tools
- Show result with histograms, scatter plots, functions

- Development started in 1994 (after PAW)
  *initiated by René Brun and Fons Rademakers*

- Currently developed by a dedicated software team (about 12 staff members) and a large user community (>100k)

- Professional approach to the software development
  - 14 platforms, 13 different compilers
  - Hourly builds, continuous integration
  - Static analysis, performance monitoring

![ROOT Downloads per year](image)
ROOT analysis framework

**Key features**
- object oriented, C++ based, multi-platform, open-source
- Classes grouped into several libraries
- Python, Ruby bindings
- C++ interpreter

**Way of usage:**
- in interactive sessions using Python or C++ interpreters
- running Python or C++ macros in corresponding interpreter
- in compiled C/C++ application with ROOT libraries linked to run at full speed

**Further in this talk:**
- ROOT graphics system
- Input and output
- Data analysis using trees
ROOT graphics

ROOT graphics is build around graphics pad concept

- linked list of primitives of any kinds
  
  ![Diagram of linked list of primitives]

- Add elements to the pad by using **Draw** method

  ```cpp
  hist = TH1F("name","title", 10, 0.0, 1.0)
  hist.Draw()
  ```

- Later access to primitives and interaction with them

  ```cpp
  c1.GetObject("hist").GetXaxis().SetTitle("X")
  ```

- Saving canvas in a variety of graphics formats

  - gif, jpg, png, eps, ps, pdf, root, C++

- Graphics canvas is interactive (mouse, keyboard events)
- Graphics can be processed in batch mode
basic shapes
**ROOT graphics**

- **basic shapes**
- **High level primitives**

Text can have different fonts and colors.

Sizes and angles can be adjusted.

Some equations:

\[ e^+e^- \rightarrow Z^0 \rightarrow ll, qq \]

\[ |a \cdot b| = |a| |b| \cos \theta \]

\[ i(\gamma^\mu + m\gamma^5) = 0 \Rightarrow (\gamma + m^2) \gamma^\mu = 0 \]

\[ L_{em} = e_{\mu} A_{\mu} , \quad j_{\mu} = \gamma_{\mu} I , \quad M = \sum A_{\mu} \gamma^\mu \]
basic shapes | High level primitives | Objects for 2 variables visualization

**ROOT graphics**

October 24, 2013, ILL, Grenoble
3 variables visualization techniques
ROOT graphics

3 variables visualization techniques

4 variables visualization techniques

October, 24, 2013, ILL, Grenoble

Data analysis in HEP, Gennady Pospelov
ROOT graphics

Access to ROOT objects in the browser

Standard histogram browser

Full-fledged event displays

Histogram browser on iPad
1D, 2D, 3D, profile histograms

```
h1 = TH1D("name","title", nbins, xmin, xmax)
h1.Fill(x)
h1.Draw()
```

Main features
- fixed/variable bin size, automatic binning
- formula evaluation, slicing, fitting
- histograms math (arithmetic, integration …)
- multiple drawing options
ROOT histograms advanced

THStack – collection of histograms

TH2Poly – polygonal bins of arbitrary shape
with partitioning algorithm to speed-up bin filling

TKDTreeBinning – multi-dimensional histogram with automatic binning
automatic space partitioning, every bin will have the same entries or the same content

THnSparse – multi-dimensional histogram for sparse data
10 dimensions with 10 bins per dimension, $10^{10}$ will fit in memory
General requirements for data format
- Public and well documented
- Machine independent
- Quick data access
- Extensibility
- Data life cycle of 10-20 years

<table>
<thead>
<tr>
<th>ASCII files</th>
</tr>
</thead>
<tbody>
<tr>
<td>- not an option</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary file formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>- allow to save complex structures and achieve maximum compression</td>
</tr>
<tr>
<td>- need to use a program to decode a binary file</td>
</tr>
</tbody>
</table>

How to assure accessibility from other programs, in 10 years, when specifications have changed?
ROOT input/output system

ROOT solution: ROOT’s files
Data format includes both the data and its description

ROOT files
- hierarchical sequential and direct access object store
- machine independent, compressed-binary format
- automated tool to generate data description when saving data, and when reading it back
- content can be structured into directories
- optimized access for high data rate

Any ROOT object can be saved and read to/from file

```c
TFile f("mydata.root","new");
TH1D h1("name", "title", nbins, xmin, xmax);
h1.Write();
```

Any user object too

```c
class MyData
{
  float t;
  string name;
  vector<int> points;
}
TFile f("mydata","new");
MyData data;
data.Write();
```
Data analysis using trees

**ROOT tree** is designed to store large quantities of same type objects inside **ROOT file**

- Extent the concept of tuple’s (ordered list of elements) to complex data structures

<table>
<thead>
<tr>
<th>Meas #</th>
<th>Status</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>status 1</td>
<td>data 1</td>
</tr>
<tr>
<td>2</td>
<td>status 2</td>
<td>data 2</td>
</tr>
<tr>
<td>3</td>
<td>status 3</td>
<td>data 3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Main features**
- Tree uses hierarchy of branches to store objects
- Branches are automatically compressed
- Branches can be switched on/off for quick access
- Tree can span over a number of files

```
class Status {
    double temp;
    double pressure;
    string date;
    string comments;
};
```

```
class Data {
    bool success;
    double x[1024], y[1024];
    double intensity[1024][1024];
    Monitor monitor;
};
```

A Tree provides a convenient interface to visualize its content.
A given tree writes the results of measurements stored in two data structures.

Status status;
Data data;

TTree tree("MyTree", "title");
tree.Branch(status);
tree.Branch(data);

while(measurements) {
    // user fills his structures here
    status.temp = 21.0

    // tree writes content of structures
    tree.Fill();
}

The tree reads the stored data and lets the user draw them quickly.

Tree tree = file->Get("MyTree")
tree.Draw("x:y:intensity","temp<20.");
To give ROOT a try

Project web-page
http://root.cern.ch/drupal

First steps and gentle introduction
http://root.cern.ch/drupal/content/discovering-root
http://root.cern.ch/drupal/content/documentation (really nice)

Image galleries
http://root.cern.ch/drupal/image

Tutorials

User’s guide
http://root.cern.ch/drupal/content/users-guide

The Reference Guide for all major ROOT classes
http://root.cern.ch/root/html534/ClassIndex.html
Different detectors in HEP uses different data acquisition systems to collect raw data, different formats and different detector software to treat the raw data.

However, final data analysis in HEP is always done in the same way, using ROOT.

ROOT consist of a set of libraries for different purposes (graphics, input/output, data analysis and fit).

- libraries can be used in C++ standalone applications
- interactively from C++ command line interpreter
- from Python

ROOT files, histograms and trees are of particular interest since they allow processing and showing the results in a very convenient manner.