A particle physics experiment

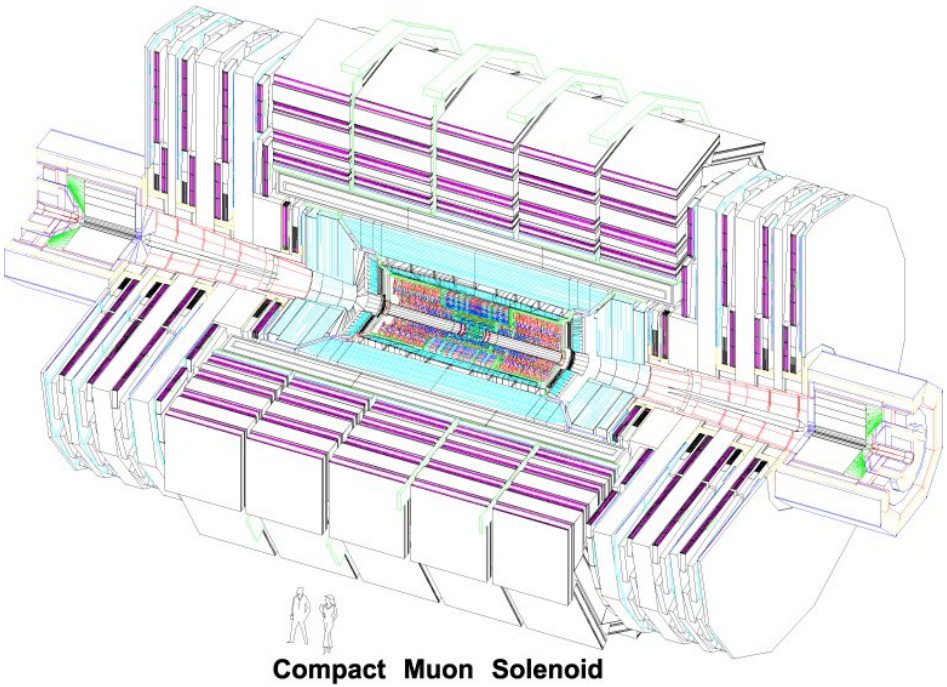
P. Ronchese
Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

The CMS experiment

Data to read and analyze have been collected by the CMS experiment at LHC



Protons coming from opposite directions collide at the center of the detector. The energy released by the collisions allows the creation of hundreds new particles.

Some of them decay almost instantaneously, producing other ones that are observed by the detector.

Particles properties

The particles produced in the collision have different properties and interact in different ways with the detector:

- electrons and photons (i.e. electromagnetic radiation) are absorbed after passing through the innermost part of the detector,
- hadrons, i.e. all particles sensitive to the strong nuclear force, are absorbed by the outermost part of the detector, shown as white slabs in the picture,
- muons, particles similar to electrons with a mass $\sim$ 200 times bigger than electrons, can go through all the detector and can be seen by dedicated ionization chambers shown as violet strips in the picture.

Particles masses

The data analyzed in the example correspond to selected particles produced in proton-proton collisions:

- hadrons:
  - pions (π) , having mass $M_\pi = 0.13957\text{GeV}/c^2$
  - kaons (K) , having mass $M_K = 0.49368\text{GeV}/c^2$
  - pions and kaons interact in the same way so that they are essentially indistiguishable
- muons, having mass $M_\mu = 0.10566\text{GeV}/c^2$

All particles can have positive or negative electric charge.

Invariant mass

The mass of a decaying particle can be determined by the masses and momenta of its decay products plus the light speed in vacuum c:

$$M = \frac{\sqrt{E_{\text{tot}}^2 - p_{\text{tot}}^2 c^2}}{c^2} \quad ; \quad E_{\text{tot}} = \Sigma_i E_i \quad ; \quad p_{\text{tot}} = |\Sigma_i \vec{p}_i|$$

The single particle energies can be computed in an analogous way:

$$E_i = \sqrt{p_i^2 c^2 + M_i^2 c^4}$$

Usually masses and momenta are measured with units GeV/c^2 and GeV/c respectively, so that in the previous formulas $c = 1$ can be assumed:

$$M = \sqrt{E_{\text{tot}}^2 - p_{\text{tot}}^2} \quad ; \quad E_i = \sqrt{p_i^2 + M_i^2}$$

Coordinates

The following coordinates are defined:

- the x axis is horizontal and orthogonal to the proton beams,
- the y axis is vertical
- the z axis corresponds to the beam axis

Particle momenta $\vec{p}$ are given in a spherical coordinate system, so that:

- $p_t = \sqrt{p_x^2 + p_y^2}$
- θ is the angle between $\vec{p}$ and the z axis
- φ is the angle between the projection of $\vec{p}$ on the xy plane and the x axis

A "pseudorapidity" quantity η is used in place of θ :

- $\eta = \frac{1}{2} \ln\left(\frac{|p|+p_z}{|p|-p_z}\right) = -\ln\left(\tan\frac{\theta}{2}\right)$
- $\theta = 2\arctan\left(e^{-\eta}\right)$

Events selected

The file `particle_events.dat` contains data relative to pions, kaons and muons coming from the decay chain of two instable particles, eventually mixed with other particles coming from other processes:

- $B^0 \rightarrow K^{\pm}\pi^{\mp}$
- $B^{\pm} \rightarrow K^{\pm}J/\psi$; $J/\psi \rightarrow \mu^+\mu^-$

The file `particle_bkgsim.dat` contains data relative to simulated hadrons and muon coming from background processes.

Data format

Data are written in binary form

Each event contain:

- 1 event identifier (i.e. an `int`)
- 1 number of particles (an `int`)
- for each particle 5 numbers:
  - 3 `float`s:
    - p_t
    - η
    - φ
  - 2 `int`s
    - charge (+1,-1)
    - muonId (0 for hadrons, 1 for muons)

The ROOT library

ROOT is a library to handle histograms

ROOT provides functions to:

- create,
- fill,
- store and retrieve,
- draw

histograms. A lot of other functionalities are provided, but their description goes beyond the scope of this course.

Histogram creation

ROOT histograms are object of type `TH1F` .

Creation of an histogram

```
TH1F* h=new TH1F(name,title,
                 nbin,xmin,xmax);
```

- Name and title are given as C-strings.
- The name must be unique, no two histograms may have the same name; spaces and symbols should be avoided.
- The title can be the same for several histograms.
- `nbin` is the number of bins.
- `xmin` is the lower limit of the histogram range.
- `xmax` is the upper limit of the histogram range.
- Each bin contains the entries for an interval with a width `(xmax-xmin)/nbin` .
- The histogram is created empty, i.e. all bins are empty.

Histogram fill

Histograms are filled by calling a function `Fill` .

```
float x;
...
h->Fill(x);
```

The function `->Fill` increases by 1 the content of the bin whose interval contains `x` :

- `x` values lower than `xmin` are classified as “underflow”,
- `x` values higher than `xmax` are classified as “overflow”.

Histogram operations

Operations can be performed over single bins

- `int n=h->GetNbinsX();` gives the number of bins.
- `float c=h->GetBinContent(i);`
  gives the content of bin `i` :
  - `i=0` gives the underflow content,
  - `i=n+1` gives the overflow content.
- `float e=h->GetBinError(i);`
  gives the error on content of bin `i` ;
- `h->SetBinContent(i,c);`
  set the content of bin `i` at `c` ;
- `h->SetBinError(i,e);`
  set the error on content of bin `i` at `e` ;
- `int i=h->FindBin(x);`
  gives the bin whose interval contains `x` .

Histogram storing

ROOT histograms are saved onto files, accessed through objects of type `TFile`.

Store of an histogram

```
TDirectory* currentDir=gDirectory;
TFile* file=new TFile(name,mode);
h->Write();
delete file;
currentDir->cd();
```

- ROOT has its own way to of handling transient (memory resident) and persistent (file resident) histograms.
- The working area should be saved before opening a file and then restored.
- The file name and open mode are given as C-strings.
- The `delete` instruction removes the object and not the file (of course)

Histogram files

- The open mode control access for reading or writing.
- The following options are available:
  - `"CREATE"` or `"NEW"` : create a new file and open it for writing; if the file already exists it's NOT opened;
  - `"RECREATE"` : create a new file and open it for writing; if the file already exists it's overwritten;
  - `"UPDATE"` open an existing file for writing; if the file does not exist it's created;
  - `"READ"` (default) open an existing file for reading.

Histogram retrieving

Retrieve of an histogram

```
TDirectory* currentDir=gDirectory;
TFile* file=new TFile(f_name,mode);
currentDir->cd();
TH1F* h = dynamic_cast<TH1F*>(
          file->Get(h_name)->Clone() );
delete file;
```

- ROOT handle object with different types and a common interface `TObject` .
- All objects are written and read through that interface.
- When an object is read from file it's type must be specified.
- A copy must be done to use the histogram after closing the file.

Compilation

ROOT headers and libraries must be included

- The `ROOTSYS` environment variable must be set.
- The headers are to be looked for in `${ROOTSYS}/include`.
- The libraries are to be looked for in `${ROOTSYS}/lib`, this path must be added to the `LD_LIBRARY_PATH` environment variable.
- All the compilation flags are provided by the command `${ROOTSYS}/bin/root-config --cflags --libs`.
- It's worth add `${ROOTSYS}/bin` to the `PATH` environment variable.

```
~> c++ -Wall `root-config --cflags` \
?  -o prog prog.cc `root-config --libs`
~> setenv LD_LIBRARY_PATH \
?  ${LD_LIBRARY_PATH}":${ROOTSYS}/lib"
```

Histogram drawing

Histograms can be drawn by using an interactive tool.

```
~> root -l f_name.root
root[1] h_name->Draw();
root[2] ...
...
root[...] .q
~>
```

- The file name can be given in the command line
- The histograms can be accessed through pointers equal to their names
- The list of histograms contained in a file can be obtained with the command "`.ls`".

Particles data dump - version 1

Read the binary file and produce a dump onto the screen

- Create 5 arrays:
  - : to contain particles p_T ,
  - : to contain particles η ,
  - : to contain particles φ ,
  - : to contain particles charge,
  - : to contain particles type.
- Create functions to:
  - read an event from file,
  - dump an event onto the screen,
- Create a `main` function to loop over the file and dump the events.

Particles data dump - version 2

Read the binary file and produce a dump onto the screen

- Create 2 `struct`s:
  - `Event` : to contain event data,
  - `Particle` : to contain particle data.
- Create functions to:
  - read an event from file,
  - dump an event onto the screen,
  - free the memory used by the event.
- Create a `main` function to loop over the file and dump the events.

Particles p_T mean - version 1

Read the binary file and compute p_t mean and r.m.s. for muons and hadrons

- Create functions to:
  - compute p_t and p_t^2 sums,
  - compute p_t mean and r.m.s.
- Modify the `main` function to hold sums and call statistical functions.

Particles p_T mean - version 2

Read the binary file and compute p_t mean and r.m.s. for muons and hadrons

- Modify the version 1 to use `class`es:
  - create a `class` to contain event data,
  - create a `class` to compute statistics.
- Modify the `main` function to use the new `class`es.

Particles p_T mean - version 3

Read the binary file and compute p_t mean and r.m.s. for muons and hadrons

- Modify the version 2 to use STL:
  - use a `std::string` to handle input file name,
  - use a `std::vector` to store particle pointers.
- Modify `PtStatistic` to use the modified `Event`.

Particles p_T mean - version 4

Read the binary file or simulate events
and compute p_t mean and r.m.s. for muons and hadrons

- Modify the version 3 to use only `class`es in place of global functions:
  - create a `class` to read or simulate events,
  - create a `class` to dump events,
  - create a `class` to compute statistics.
- Create the `class`es as derivations of interfaces to get events and process them.
- Create a `class` to simulate events.
- Modify the `main` function to use the new `class`es.

Particles p_T mean - version 4b

Read the binary file or simulate events
and compute p_t mean and r.m.s. for muons and hadrons

- Modify the version 4 and call “compute” automatically inside functions returning mean and RMS
- `mutable` variables must be used

Event discrimination - version 1

Build a likelihood discriminator to
classify events as “signal” and “background”.

- Build a set of discriminating variables:
  - $\Sigma_i p_{t,i}$, $p_{t,\text{min}}$, $p_{t,\text{max}}$,
  - $\Delta\varphi_{\text{min}}$, $\Delta\varphi_{\text{max}}$,
  - $\Delta\eta_{\text{min}}$, $\Delta\eta_{\text{max}}$.
- Create and fill histograms of all the variables for signal and background events.
- For each variable compute the probabilities $P_{j,\text{sig}}$ and $P_{j,\text{bkg}}$ that the variable stays inside a defined interval, for a signal or background event.
- Compute the discriminating variable:

$$D = \Pi_j P_j = \frac{\Pi_j P_{j,\text{sig}}}{\Pi_j P_{j,\text{sig}} + \Pi_j P_{j,\text{bkg}}} \ .$$

- Create and fill histograms of the discriminating variable for signal and background events.

Event discrimination - version 2

Build a likelihood discriminator to
classify events as “signal” and “background”.

- Modify the version 1 to encapsulate the following operations for a generic set of variables:
  - compute the discriminating variable,
  - fill and save on file the variables histograms,
  - read from file the variables histograms.
- Modify the concrete discriminator `class` to use the generic one by inheritance.

Event discrimination - version 3

Test the likelihood discriminator
with different subsets of variables.

- Modify version 2 to compare the discriminating performances with different subsets of variables:
- Create several discriminator `class`es, setting different variables as “active”.
- Create a corresponding histogram for each discriminator.
- For each event compute the discriminator for all the variable choices and fill the corresponding histograms.

Event discrimination - version 4

Test the likelihood discriminator
with different subsets of variables.

- Modify version 3 to compute the variables only once for all the discriminators.
- Move the variable declaration and computation to a new small `class` .
- Include in the discriminator `class` a pointer to this new `class` .
- Provide in the constructor of the discriminator `class` the choice to create a new set of variables or share the ones in another object.

Mass reconstruction - version 1

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Create classes to select B^0 , $B^{\pm}$ events:
  - B^0 events have at least 2 hadrons with opposite charge; the K can be taken as the leading-p_t hadron and the π as the leading-p_t hadron with opposite charge w.r.t. the K .
  - $B^{\pm}$ events have 2 muons with opposite charge and an hadron.
- Create classes to:
  - convert coordinates from spherical to cartesian,
  - reconstruct the masses.
- Add an analyzer to handle those classes and fill histograms.

Mass reconstruction - version 2

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 1 to use a "factory" to create a data source.
- Create a `class AnalysisInfo` to handle command line parameters, with functions to:
  - look for keys among the parameters,
  - return value following a key.
- Use a `class SourceFactory` to create data source with a `create` function:
  - taking an `AnalysisInfo` as argument,
  - returning an object to read or simulate events according to the command line parameters.
- Move the code to create data sources from the `main` function to the factory.

Mass reconstruction - version 3

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 2 to use a “factory” to create analyzer objects:
  - create a `class AnalysisFactory` to create analyzers and implement a mechanism to create analyzers according to command-line parameters,
  - modify `EventDump` and `MassHisto` to be handled by `AnalysisFactory` ,
  - add “builder” `class`es to register the available analyzers and create them on request.
- Add an `AnalysisInfo*` parameter to `AnalysisSteering` and save a copy to be used later.
- Move the code to create analyzers from the `main` function to the factory.

Mass reconstruction - version 4

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 3 to use a “dispatcher” to loop over events:
  - declare `JPsiKCandidate` , `HHCandidate.h` and `Cartesian` as `LazyObserver`s and `Singleton`s,
  - declare `EventDump` , `ParticleStatistic` and `MassHisto` as `ActiveObserver`s, and remove the calls to `update` functions,
  - declare `PtStatistic` as an `ActiveObserver` too,
  - remove the `process` function from `AnalysisSteering` and rename it to `update` in all analyzers,
  - add a `run` function to `EventSource` .
- Move the event loop from the `main` function to the `run` function in `EventSource` .

Mass reconstruction - version 5

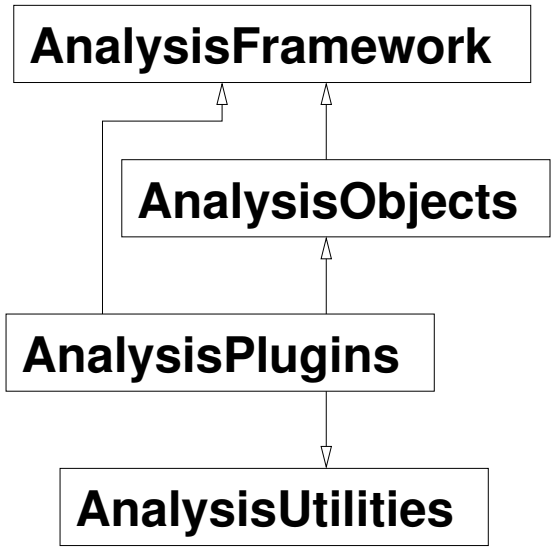
Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 4 to use a “dispatcher” to handle begin/end of analysis:
  - declare in `AnalysisInfo` an `enum AnalysisStatus` with values `begin` and `end` ,
  - declare `AnalysisSteering` as an `ActiveObserver` of `AnalysisInfo::AnalysisStatus` ,
  - in `AnalysisSteering` implement a function `update` calling in its turn `beginJob` or `endJob` .
- Replace in `main` the analyzers loop calls to `beginJob` and `endJob` with notifications of `AnalysisInfo::begin` or `AnalysisInfo::end` .

Mass reconstruction - version 6

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 5 to organize the code in packages:
  - create 4 packages: `AnalysisFramework` , `AnalysisPlugins` , `AnalysisObjects` , `AnalysisUtilities` ,
  - move all source files, including `main.cc` , into those packages, avoiding circular dependencies,
  - compile each package into a library but `AnalysisPlugins` , where each analyzer is to be compiled to a distinct library.

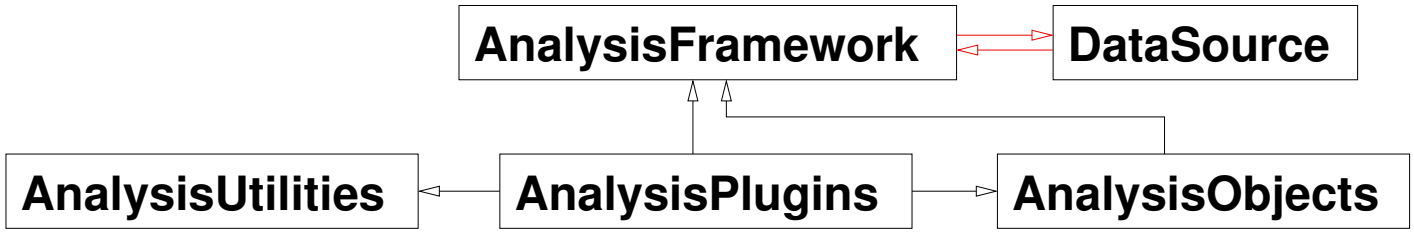


- Produce the executable by using a dummy source code.

Mass reconstruction - version 6b

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

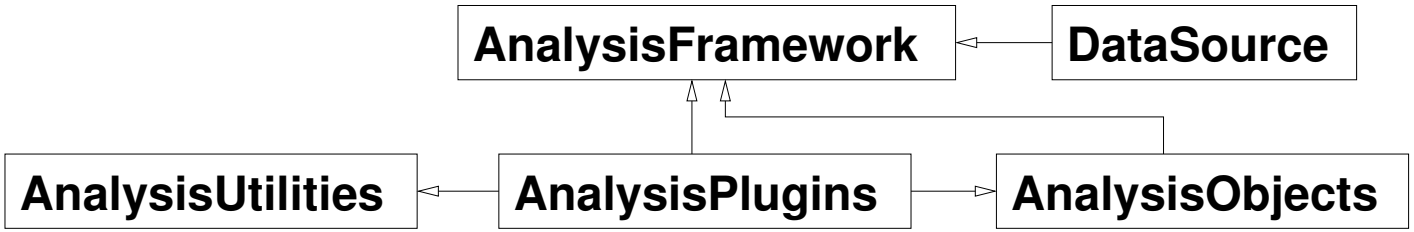
- Modify the version 6 and move the concrete event sources to a dedicated package.
- A circular dependency occurs.



Mass reconstruction - version 7

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 6b to remove the circular dependency.
- Implement a mechanism to create event sources in a similar way as analyzers.



Mass reconstruction - version 8

Reconstruct the masses of the B^0 , $B^{\pm}$ and J/ψ .

- Modify the version 7 and move the event data structure to a dedicated package.
- Create an `EventBase` interface and make `Event` to inherit from it.