The Bragg curve

P. Ronchese
Dipartimento di Fisica e Astronomia "G.Galilei"

Università di Padova

"Object oriented programming and C++" course

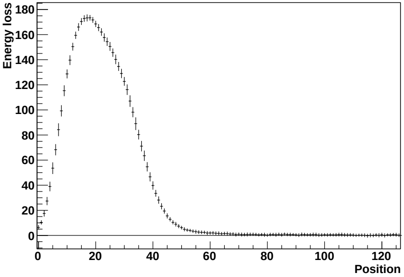
Energy measurement in α decay

Data to read and analyze have been collected in the nuclear physics lab. course.

Radioactive elements emit α particles, absorbed by a detector giving the energy lost by the particles while it goes through the material.

- The energy loss rises up to a maximum.
- The energy loss, after having reached the maximum, decreases to zero.
- This behaviour can be drawn on a plot, the “Bragg curve”
- The total energy is given by the sum of the energy losses in all the steps

Bragg curve



Both axes use arbitrary units, there's no special physical meaning in this plot.

- Energy loss at each step is computed from the measurements by subtracting a constant, called "background".
- The background can be estimated by averaging the measurements in the last part of the tail
- The plot above has been produced after background subtraction.

Events

The file `bragg_events.dat` contains the data,
written in binary form

For each event the following data are written:

- an event identifier (i.e. an `int`)
- the number, comprised between 120 and 128, of energy measurements,
- the list of energy measurements, each given as an `int` .

Events from 3 (simulated) different radioactive elements are stored, mixed with “background” events

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`".

Energies data dump - version 1

Read the binary file and produce a dump onto the screen

- Create an array of `int`s to contain energies.
- 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.

Energies data dump - version 2

Read the binary file and produce a dump onto the screen

- Create a `struct Event` to contain event data, with members corresponding to the data listed above.
- 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.

Energies data mean - version 1

Read the binary file and compute mean and r.m.s. energies

- Create functions to:
  - compute energy and energy squares sums for each point,
  - select events with total energy between 6000 and 7500,
  - compute energy mean and r.m.s. for each point.
- Modify the main function to hold sums and call statistical functions.

Energies data mean - version 2

Read the binary file and compute mean and r.m.s. energies for 3 Bragg curves and background

- Modify the version 1 to use `class`es:
  - create a `class` to contain event data,
  - create a `class` to compute statistics.
  - select events with total energy in the following ranges:
    - between 3000 and 5000,
    - between 6000 and 6499,
    - between 6500 and 6799,
    - between 6800 and 7200.
- Modify the main function to use the new `class`es.

Energies data mean - version 3

Read the binary file and compute mean and r.m.s. energies for 3 Bragg curves and background

- Modify the version 2 to use STL:
  - use a `std::string` to handle input file name
  - use a `std::vector` to store energies
- Modify the main function to use the modified `class`es.

Energies data mean - version 4

Read the binary file and compute mean and r.m.s. energies for 3 Bragg curves and background

- Modify the version 3 to use only `class`es in place of global functions:
  - create a `class` to read events,
  - create a `class` to dump events,
  - create a `class` to compute mean and rms energies: inside a `class` performing the general analysis steering create several (i.e. 3+1) instances of a `class` computing mean and rms energies, selecting different total energy ranges
- Create the `class`es as derivations of interfaces to get events and process them.
- Modify the main function to use the new `class`es.

Energies data mean - version 4b

Read the binary file and compute mean and r.m.s. energies for 3 Bragg curves and background

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

Energies data plot - version 1

Plot mean and r.m.s. energies
for 3 Bragg curves and background

- Modify the mean-version 4 to include graphic plots and allow multiple plots handling:
  - inside the general analysis steering `class` pair each object computing mean and rms energies with a `TH1F` object,
  - create, set and save plots in the same `class` , too.
- All other `class`es and the main function stay unchanged.

Energies data plot - version 2

**Plot mean and r.m.s. energies
for 3 Bragg curves and background**

- 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.

Energies data plot - version 3

Plot mean and r.m.s. energies
for 3 Bragg curves and background

- 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.

Energies data plot - version 4

Plot mean and r.m.s. energies for 3 Bragg curves and background

- Modify the version 3 to use a “dispatcher” to loop over events:
  - take the new versions of `AnalysisSteering` and `EventSource` from the particle analysis example,
  - modify `EventDump` and `ElementReco` to be `ActiveObserver`s,
- create a new analyzer `BGCalc` to compute background and printout it at execution end,
- create a `class TotalEnergy` to compute total energy, and declare it being a `LazyObserver`.
- Modify the main function to use the new `EventSource`.

Energies data plot - version 5

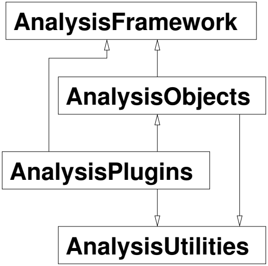
Plot mean and r.m.s. energies
for 3 Bragg curves and background

- Modify the version 4 to use a “dispatcher” to handle begin/end of analysis: take the new versions of `main` , `AnalysisInfo` and `AnalysisSteering` from the particle analysis example.
- Add a `class Constants` to contain the background mean and rms.
- Add a new function to `TotalEnergy` to compute energy after background subtraction, and use it to select and draw Bragg curves.
- In `ElementReco` get the energy ranges from a text file with name specified in the command line.

Energies data plot - version 6

Plot mean and r.m.s. energies
for 3 Bragg curves and background

- 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.