Laboratorio II – 1° modulo
Lezione XX
Rappresentazione dei dati mediante `ntuple`

Indice

- Rappresentazione dei dati mediante `ntuple`
  - La classe `TNtuple` di ROOT
- Esempio
- Salvare/Leggere `ntuple` da file
- Selezionare gli eventi
- Gestione delle `ntuple` in maniera più avanzata
- Commenti / osservazioni
- Esercizi

- Approfondimenti
  - ROOT in interattivo

Rappresentazione dei dati mediante `ntuple`

Fino ad ora abbiamo preso in considerazione il caso in cui vengano misurate solo due grandezze fisiche, `x` e `y`. Per esempio indice di rifrazione e lunghezza d'onda (legge di Cauchy), oppure tensione e tempo (circuito RC o RL), ecc…

Esempio di file di dati

x	y	errX	errY
0	1	0.25	0.05
0.25	0.987701	0.25	0.05
0.5	0.951106	0.25	0.05
0.75	0.891115	0.25	0.05
1	0.809204	0.25	0.05
...			

Ci sono però casi in cui per **descrivere** in maniera **completa** un fenomeno, è necessario osservare **svariate grandezze fisiche**, $x_1 \ldots x_n$, **nello stesso istante**. **E.g.:** per monitorare lo stato di salute di un paziente durante un'operazione chirurgica è necessario misurare:

- Temperatura corporea
- pressione sanguigna
- battito cardiaco
- quantità di CO_2 prodotta
- ecc…

Rappresentazione dei dati mediante `ntuple`

Tutti questi dati possono essere organizzati in una tabella in cui ogni colonna corrisponde ad una diversa osservabile fisica:

Fotografia del fenomeno all'istante "2"

Esempio di tabella di dati

Indx	Temperatura	Pressione (sistolica)	Pressione (diastolica)	Battito	CO_2
1	36.5	80	100	70	1.2
2	36.7	82	110	75	1.5
3	36.8	79	109	72	2.0
4	37.0	81	101	80	1.6
5	37.1	83	104	79	1.8
...	...				

Come rappresentare questi dati in un computer? Un modo molto valido è tramite le **`ntuple`**. Le `ntuple` non sono altro che un insieme di vettori, uno per ciascuna grandezza fisica osservata

Per implementare le `ntuple` ROOT mette a disposizione la classe `TNtuple`: `#include "TNtuple.h"`

```
TNtuple myNtuple; \\ Default constructor
TNtuple myNtuple (const char* name, const char* title,
        const char* varlist); \\ Il costruttore crea una ntupla
vuota dotata delle osservabili, i.e. colonne, definite tramite
varlist: "var1:var2:..." (notare il separatore "due punti")
```

Rappresentazione dei dati mediante `ntuple`

Da notare che in ROOT:

- ogni misura (i.e. ogni riga della tabella) è chiamata **event**
- ogni osservabile (i.e. colonna della tabella) è chiamata **branch**

Questi vettori di dati possono poi essere utilizzati per costruire gli istogrammi corrispondenti alle diverse osservabili, ma cosa ancora più importante è possibile **"interrogare"** la **`ntupla`** in maniera tale da **cercare** le **correlazioni** più disparate tra le osservabili

Riferendosi all'esempio precedente, è possibile controllare:

- se esiste una relazione tra temperatura corporea e battito cardiaco
- l'andamento della pressione solo per valori di CO_2 maggiori `k`
- l'andamento del battito cardiaco solo per valori di pressione diastolica maggiori di `1.2` volte quella sistolica
- e molto di più …

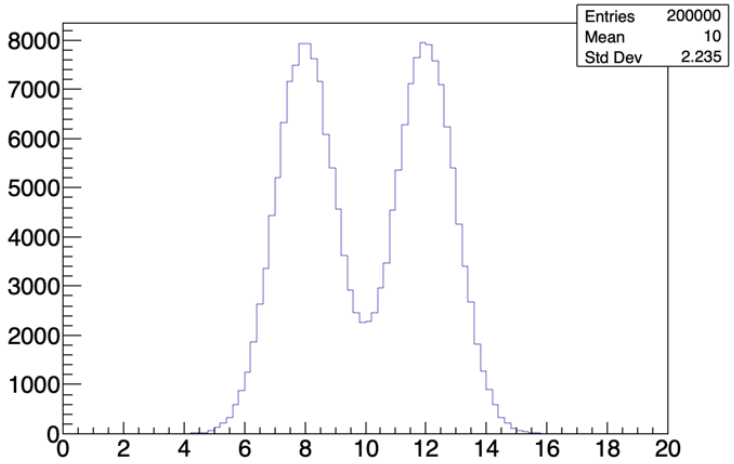
Esempio

Vediamo ora un semplice esempio con due sole osservabili

```
...
TNtuple* myNtuple = new TNtuple ("myNtuple","myNtuple","x:y");
// ... apertura del file di dati ...
while (!in.eof())
  {
    in >> a >> b;
    if (in.eof() == true)
      break;
    myNtuple->Fill(a,b);
  }
// ... altre operazioni ...
TCanvas* c1 = new TCanvas ("c1",
            "Marginal distribution of x")
TH1D* h1 = new TH1D("h1","h1",100,0,20);
myNtuple->Draw("x >> h1");
c1->cd();
h1->Draw();
```

File di dati

Indx	x	y
1	11.4725	7.01742
2	7.42488	9.61838
3	7.26657	7.96239
4	9.72206	8.65665
5	8.10054	8.64317
...	...	

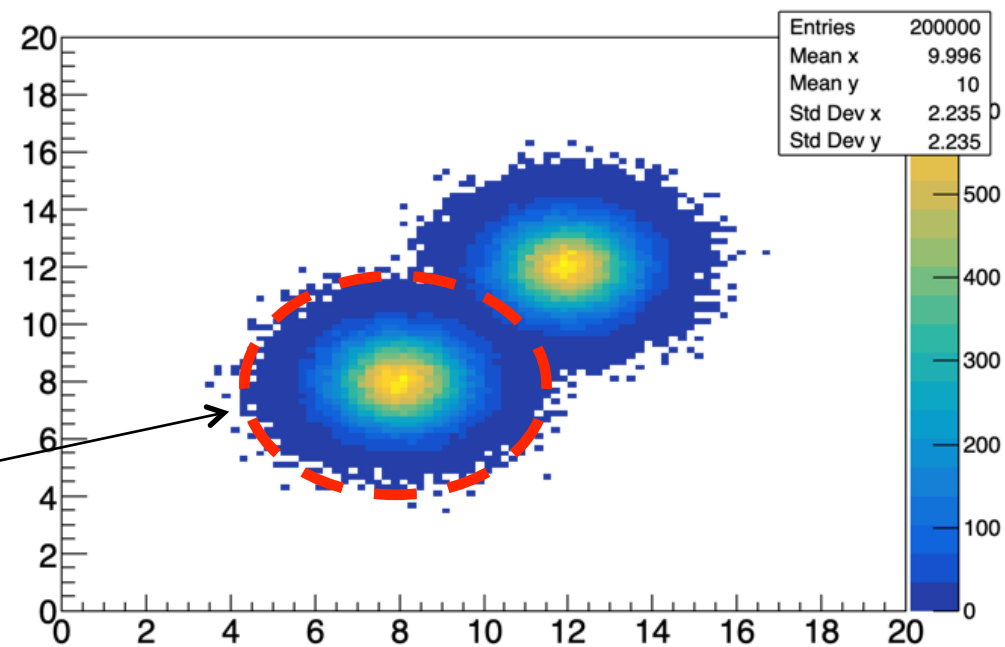


Esempio

Disegnamo ora entrambe le osservabili in uno *scatterplot*

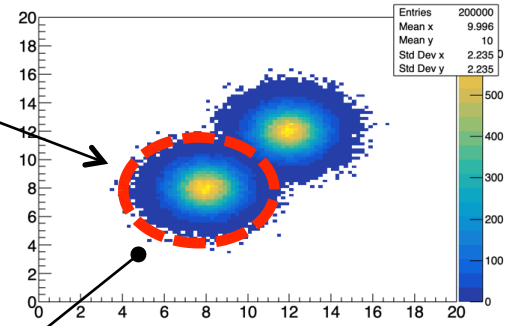
```
...
TCanvas* c3 = new TCanvas ("c3","Joint distribution");
TH2D* h3 = new TH2D("h3","h3",100,0,20,100,0,20);
// Equivalentemente:
// TGraph* myGraph = new TGraph(myNtuple->GetSelectedRows(),
// myNtuple->GetV2(), myNtuple->GetV1());
myNtuple->Draw("x:y >> h3");
c3->cd();
h3->Draw("gcolz");
// Oppure: myGraph->Draw();
...
```

E` possibile inoltre selezionare una regione come quella in figura e disegnare i dati che cadono in quella regione



Esempio

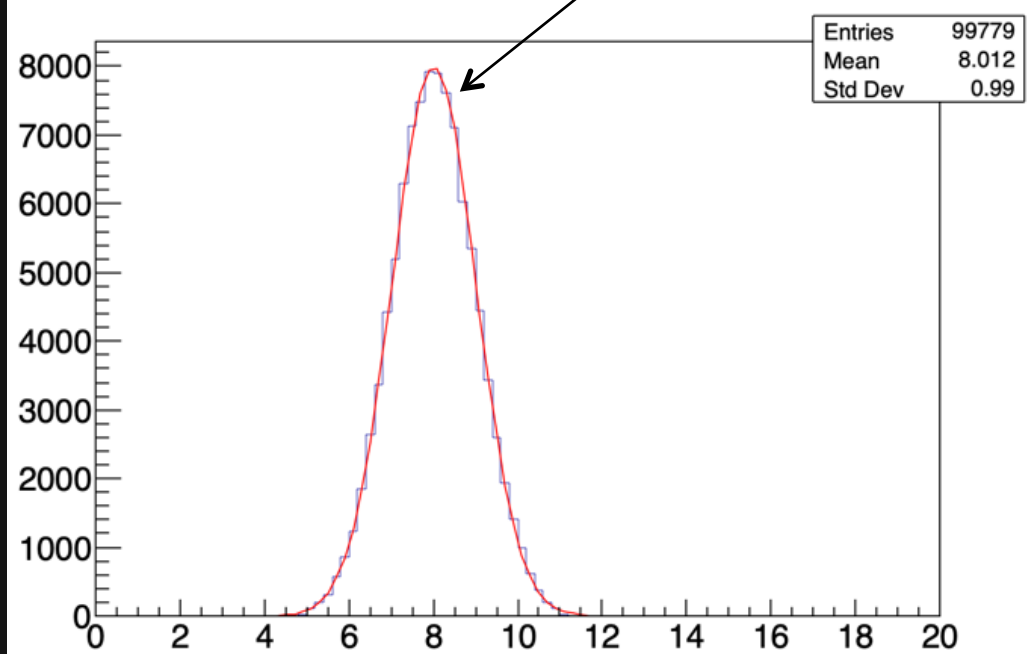
```
TCanvas* c4 = new TCanvas ("c4","Test event selection");
TH1D* h4 = new TH1D("h4","h4",100,0,20);
myNtuple->Draw("x >> h4","(y-8)*(y-8) + (x-8)*(x-8) < 10");
c4->cd();
h4->Draw();

TF1* myFunc = new TF1("myFunc","gaus",0,20);
h4->Fit("myFunc");
```



Per vedere velocemente il contenuto di una ntupla:

```
myNtuple->Scan();
```

```
**********************************
*    Row   *         x *         y *
**********************************
*        0 * 11.472499 * 7.0174198 *
*        1 * 7.4248800 * 9.6183795 *
*        2 * 7.2665700 * 7.9623899 *
*        3 * 9.7220602 * 8.6566495 *
*        4 * 8.1005401 * 8.6431703 *
*        5 * 8.7061300 * 8.4087295 *
*        6 * 7.5200901 * 8.8574399 *
*        7 * 8.0492696 * 7.0401701 *
*        8 * 7.9386801 * 7.7267398 *
*        9 * 7.2333998 * 7.2045898 *
*       10 * 6.1240201 * 7.5698800 *
*       11 * 7.1373000 * 9.5867099 *
*       12 * 8.2017097 * 9.2719802 *
*       13 * 8.4276504 * 7.9965300 *
*       14 * 8.1526298 * 7.0933098 *
*       15 * 7.4655399 * 6.8119702 *
*       16 * 6.4693298 * 8.3803100 *
*       17 *   8.59902 * 8.3508596 *
*       18 * 8.5648202 * 6.7160100 *
*       19 * 8.5223197 * 6.7617201 *
*       20 * 6.8700299 * 8.0060100 *
*       21 * 7.1858601 * 7.4433798 *
*       22 * 6.6160001 * 6.8647098 *
*       23 * 8.1190500 * 6.6103901 *
*       24 * 7.4528098 * 9.2352504 *
Type <CR> to continue or q to quit ==>
```



Salvare/Leggere `ntuple` da file

E` possibile salvare una `ntupla` in un file in formato ROOT. Per fare questo è necessario utilizzare la classe **`TFile`** di ROOT:

```
#include "TFile.h"
...
TFile* myFile = new TFile("nome_file.root","RECREATE");
TNtuple* myNtuple = new TNtuple ("myNtuple","myNtuple","x:y");
...
myFile->cd();
myNtuple->Write();
...
```

Inoltre è ovviamente possibile leggere una `ntupla` da file tramite le istruzioni:

```
...
TFile* myFile = new TFile("nome_file.root","READ");
TNtuple* myNtuple = (TNtuple*) myFile->Get("myNtuple");
...
```

Cast al tipo “puntatore a `Tntuple`”

Selezionare gli eventi

In riferimento all'esempio presentato precedentemente, la selezione degli eventi era stata implementata in questo modo:

```
myNtuple->Draw("x >> h4","(y-8)*(y-8) + (x-8)*(x-8) < 10");
```

Un altro modo per implementare la selezione degli eventi è tramite l'utilizzo del tipo **TCut** messo a disposizione da ROOT:

```
#include "TCut.h"
...
TCut cut = "(y-8)*(y-8) + (x-8)*(x-8) < 10";
myNtuple->Draw("x >> h4",cut);
...
// Altro esempio
TCut cut = "(E1 + E2) < 2625 &&  (E1 + E2) > 2605";
myNtuple->Draw("E1:E2 >> h5",cut);
...
```

Gestione delle `ntuple` in maniera più avanzata

In ROOT esiste una struttura dati chiamata **`TTree`** che è una generalizzazione della struttura `TNtuple` ed è legata al concetto di "persistenza dei dati". Senza entrare qui in dettaglio nelle potenzialità dei `TTree` vediamo come possono essere utilizzati come `ntuple`

```
#include "TTree.h"
...
Ttree myTreeOut; // Default constructor
TTree* myTreeOut = new TTree("myTree","A simple Tree");
// Costruttore: TTree(const char* name, const char* title

double px, py, random;
int ev;
myTreeOut->Branch("px",     &px,     "px/D");
myTreeOut->Branch("py",     &py,     "py/D");
myTreeOut->Branch("random", &random, "random/D");
myTreeOut->Branch("ev",     &ev,     "ev/I");
```

Ogni **`Branch`** è definito mediante un nome, un puntatore ad una variabile ed il tipo della variabile (e.g. **`D`** = `double`; **`I`** = `integer`; ecc…)

Ogni grandezza (i.e. colonna) della `ntupla` è definita mediante un **`Branch`**

Gestione delle `ntuple` in maniera più avanzata

Per riempire un `TTree` bisogna prima assegnare un valore alle variabili associate ai branch e poi eseguire il metodo `Fill()`

```
...
for (unsigned int i = 0; i < NENTRIES; i++)
  {
    px     = rand_TAC_gaus(0,1);
    py     = rand_TAC_gaus(0,1);
    random = rand_range(0,1);
    ev     = i;
    myTreeOut->Fill();
  }
myTreeOut->Scan();
...
// Salva i dati in un file (TFile)
TFile* myFileOut = new TFile(FILE_NAME,"RECREATE");
myTreeOut->Write();
myFileOut->Close();
```

Gestione delle `ntuple` in maniera più avanzata

Per leggere, interrogare e plottare un `TTree` è necessario associare delle variabili ai branch ed eseguire il metodo `Fill()` degli istogrammi

```
...
TFile* myFileIn = new TFile(FILE_NAME,"READ");
TTree* myTreeIn = (TTree*) myFileIn->Get("myTree");
myTreeIn->SetBranchAddress("px",      &px);
myTreeIn->SetBranchAddress("py",      &py);
myTreeIn->SetBranchAddress("random",  &random);
myTreeIn->SetBranchAddress("ev",      &ev);
unsigned int nEntries = myTreeIn->GetEntries();
for (unsigned int i = 0; i < nEntries; i++)
  {
    myTreeIn->GetEntry(i);
    if ((random > 0.5) && (ev > 10))
    {
      h1->Fill(random);
      h2->Fill(px,py);
    }
  }
```

Apre il file ed estrare il `TTree`

Associa le variabili precedentemente definite:
`double px, py, random;`
`int ev;`
ai branch

Estrae una "entry" (i.e. una riga della tabella) alla volta

Tramite l'`if` viene implementa la selezione dei dati (corrisponde a `TCut`)
Vengono poi riempiti gli opportuni istogrammi

Commenti / osservazioni

- Come è stato illustrato nelle slide precedenti le `ntuple` sono, in ultima istanza, dei **contenitori di dati**. E` possibile quindi usare una `ntupla`, in alternativa al file di testo, per contenere le coppie di misure ($x_i$, $y_i$) con i relativi errori ($\sigma_{xi}$, $\sigma_{yi}$) per poi istogrammarli con, per esempio, un `TGraphError`

Esempio di `ntupla`

Indx	x	y	σ_x	σ_y
1	0	1	0.25	0.05
2	0.25	0.987701	0.25	0.05
3	0.5	0.951106	0.25	0.05
4	0.75	0.891115	0.25	0.05
5	1	0.809204	0.25	0.05
...	...			

- Inoltre è opportuno ricordare che il concetto di `ntupla` non è legato a ROOT. ROOT semplicemente la implementa a suo modo mediante la classe `TNtupla`, e più in generale mediante la classe `TTree`. Altri linguaggi implementano il concetto di `ntupla` a loro modo. Le `ntuple` sono infatti un potente strumento per l'analisi dati che è trasversale rispetto particolare linguaggio di programmazione

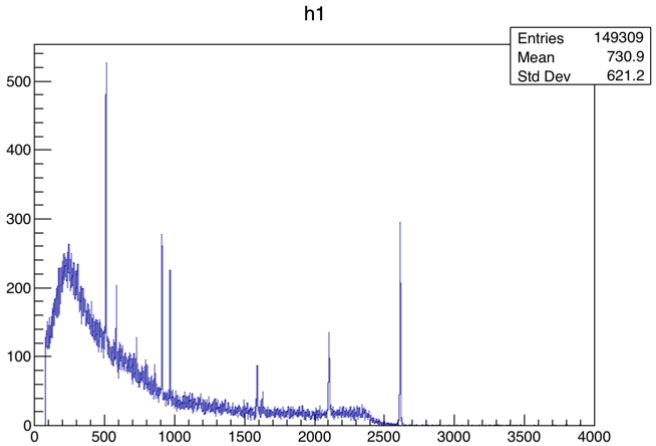
Esercizi

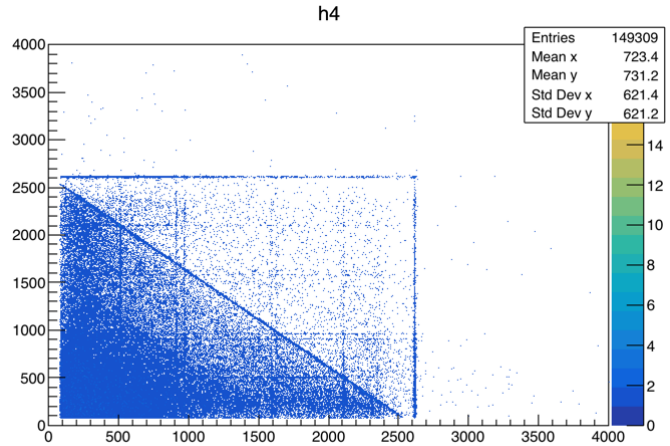
Esercizio 1: Aprire il file `data.root` e caricare in memoria la `TNtupla` in esso contenuta, denominata "myNtuple", le cui osservabili sono `E1` ed `E2`. Produrre quindi i seguenti grafici:

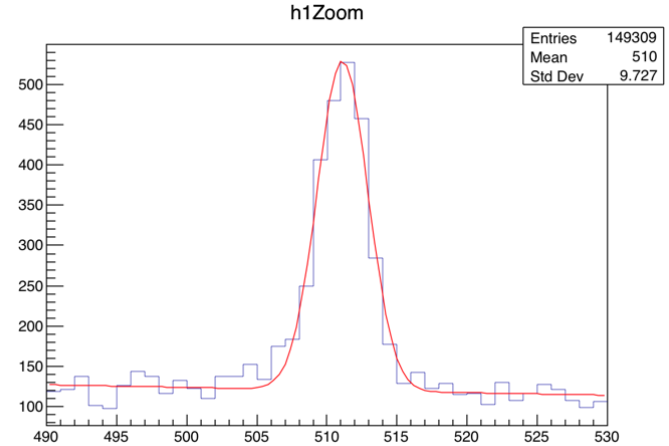
- disegnare gli istogrammi di `E1` ed `E2` utilizzando un binning da `0` a `4000`, con `4000` bins
- ridisegnare l'istrogramma di `E1` restringendone gli estremi intorno ad uno dei picchi; eseguire quindi il fit del picco con una funzione data dalla somma di una gaussiana e di una retta
- disegnare lo *scatterplot* `E1:E2`
- selezionare i dati che si addensano intorno alla diagonale principale del grafico precedente e disegnare `E1+E2`
- dopo ave applicato la selezione disegnare l'istogramma di `E1`
- creare un `TFile` denominato `graphs.root` e salvare i canvas prodotti

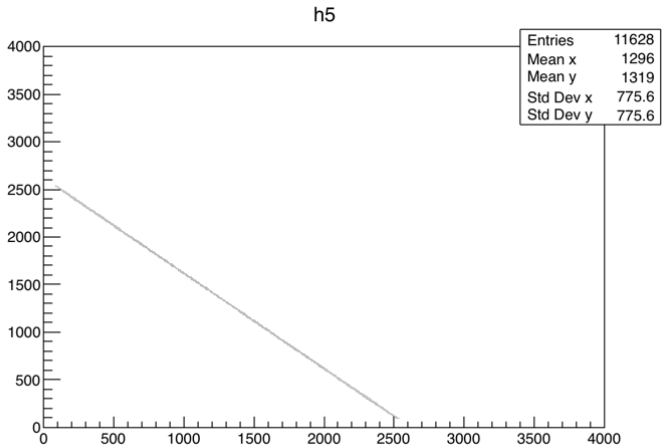
Esercizi

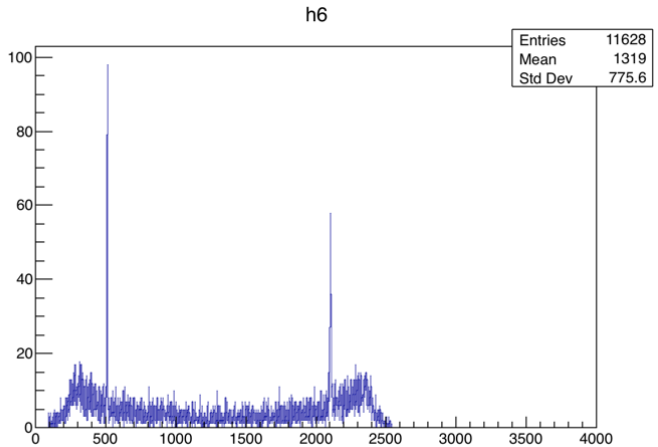
Vengono riportati in questa slide i plot richiesti nell'esercizio 1











Interpretazione dei plot dell'esercizio 1

I dati dell'esercizio proposto riguardano una misura di radioattività effettuata con due rivelatori in coincidenza temporale: **`E1`** ed **`E2`** sono le energie misurate da ciascun rivelatore

1. I picchi negli spettri `E1` ed `E2` corrispondono ad eventi in cui il decadimento radiativo è avvenuto mediante emissione di un raggio gamma che è stato poi assorbito, con una interazione di tipo **fotoelettrico**, da uno dei due rivelatori

2. Gli eventi sulla diagonale corrispondono ad interazioni di tipo **Compton** in cui il raggio gamma ha interagito in uno dei rivelatori ed è stato poi *scatterato* nel secondo rivelatore, dove ha rilasciato la sua energia. Infatti sulla diagonale **`E1+E2`** è costante e corrisponde all'energia del raggio gamma (`2615 keV` dal Tl^{208} nella catena del Th^{232})

3. I due picchi che appaiono nell'ultimo plot corrispondono ad un'energia di `511 keV` e `2615 - 511 = 2104 keV`. Questi sono eventi di tipo **produzione di coppie** in cui il raggio gamma ha prodotto la coppia e^+e^-. L'elettrone è stato assorbito da uno dei rivelatori. Il positrone si è annichilato con un elettrone producendo due raggi gamma da `511 keV`, ciascuno assorbito da un rivelatore

Esercizi

Esercizio 2: Viene fornita una `ntupla` in formato `TTree` nel file `hiddenSignal.root`. La `ntupla` è composta da due branch. Scrivere un programma che legge il `TTree` e seleziona il "segnale nascosto" nei dati, cioè seleziona quegli eventi la cui distribuzione è più densa. Suggerimenti:

- visto che la `ntupla` ha solo due branch è consigliabile prima produrre lo *scatterplot* delle due osservabili
- osservare lo *scatterplot* e cercare di individuare il segnale
- implementare la selezione degli eventi in maniera tale da estrarre solo la regione contenente il segnale

Esercizi

Esercizio 3: Creare una classe che gestisce il formato dati `ntuple`. In particolare la classe deve avere il seguente header file:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

#include "TH1D.h"
#include "TH2D.h"
#include "TGraphErrors.h"

class myNtuple
{
 public:
  myNtuple  (int n) : fields(n) {};
  ~myNtuple ();

  void AddEntry      (double* vals);
  bool LoadFromFile (std::string fileName);
  bool Write2File    (std::string fileName);

  void Fill1DHisto (unsigned int varIndx, TH1D* histo);
  void Fill2DHisto (unsigned int varIndxX,
       unsigned int varIndxY, TH2D* histo);
  void Fill1DGraph (unsigned int varIndxX,
       unsigned int varIndxY, unsigned int
                                varIndxErrX,
       unsigned int varIndxErrY, TGraphErrors*
                                  graph);
  std::vector<double*>* GetNtuplePtr ()
    { return &ntuple; };
  int GetEntries () { return ntuple.size(); };

  private:
   int fields;
   std::vector<double*> ntuple;
  };
```

Esercizi

- **`Write2File`** salva la `ntupla` in un file in formato testo
- **`LoadFromFile`** legge la `ntupla` da un file in formato testo

Esempio di file di dati

```
x          y
-4.99992   -3.68462
2.55605    -0.413499
0.327672   -2.81041
-4.52955   1.78865
...
```

- La struttura dati è gestita in memoria mediante un `vector` di array: `std::vector<double*> ntuple`, in particolare il vettore è istanziato nello *stack* e consta di una sequenza di puntatori ad array allocati nello *heap*
- **`fields`** contiene il numero di "colonne" della `ntupla`, cioè il numero di osservabili, che corrisponde quindi alla lunghezza degli array
- Il metodo **`AddEntry`** permette di inserire una nuova "riga" nella `ntupla`, cioè deve istanziare un nuovo array nella memoria dinamica e aggiungere il relativo puntatore nel vettore `ntuple`

Esercizi

- I metodi **`Fill1DHisto`**, **`Fill2DHisto`**, e **`Fill1DGraph`**, servono per riempire un istogramma 1D, un istogramma 2D, oppure un grafico con i dati della `ntupla`

Il `main` program deve leggere il file di dati `hiddenSignal.txt` e svolgere la stessa analisi dell'esercizio 2 … si dovranno ottenere quindi gli stessi risultati

Approfondimenti

- **ROOT in interattivo**

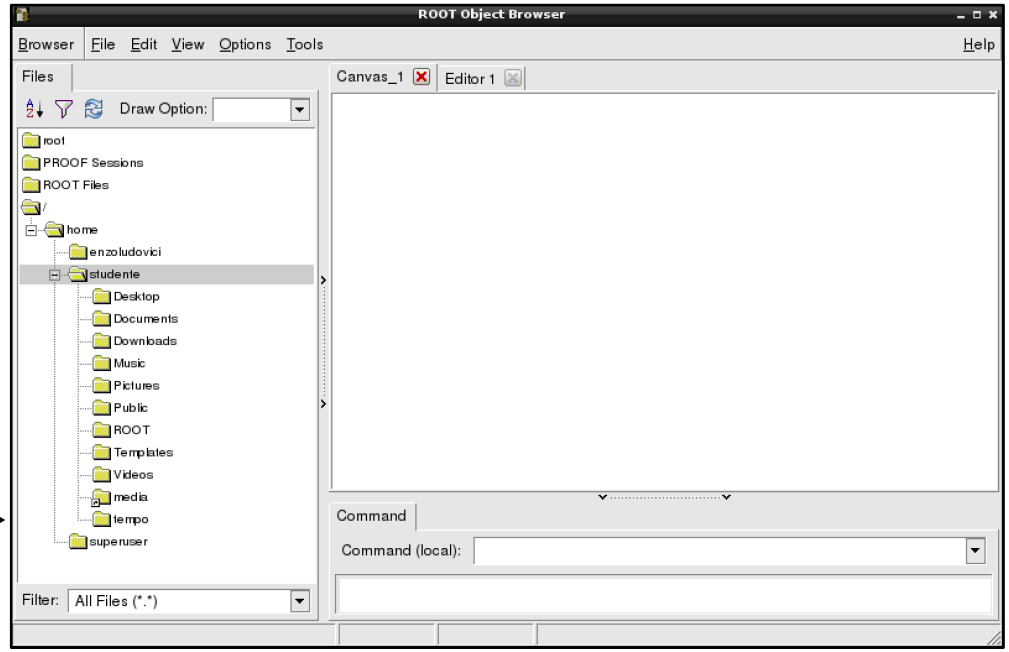
ROOT in interattivo

Scrivendo semplicemente `root` al *prompt* del terminale si entra nell'ambiente interattivo di ROOT. Dall'ambiente interattivo è possibile eseguire varie operazioni tra cui scrivere codice `C/C++` che viene immediatamente interpretato dall'interprete `C/C++` di ROOT chiamato `CINT`

Per cominciare è possibile scrivere il comando **`.help`** che stampa a schermo alcuni suggerimenti di base

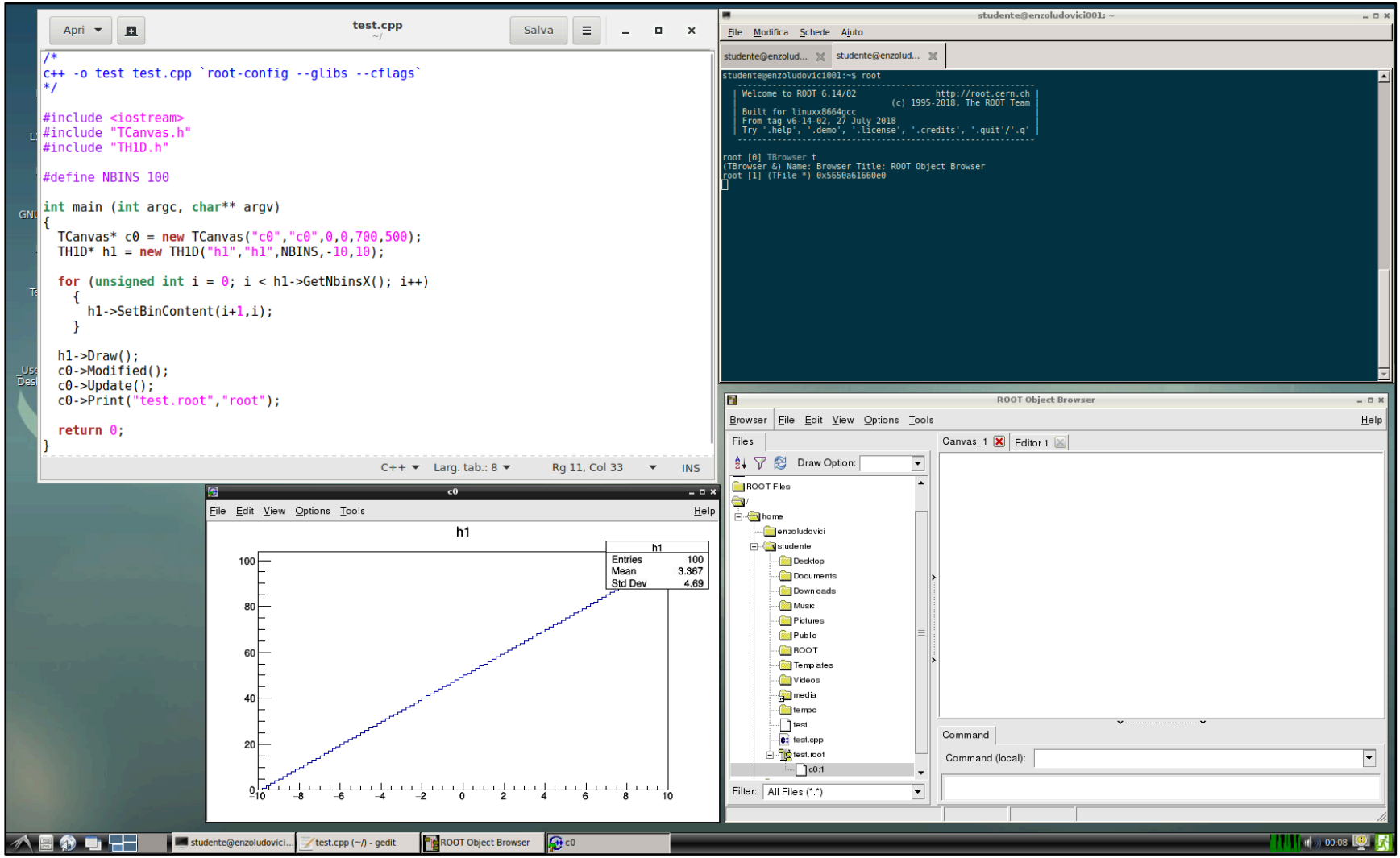
```
studente@enzoludovici001:~$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.14/02                    http://root.cern.ch |
  |                                   (c) 1995-2018, The ROOT Team |
  | Built for linuxx8664gcc                                        |
  | From tag v6-14-02, 27 July 2018                                |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'     |
   ------------------------------------------------------------------

root [0]
```

Tipicamente però è utile entrare nell'ambiente interattivo di ROOT per lanciare l'applicativo `TBrowser`. Questo applicativo consente di **aprire** ed **esplorare file** salvati in formato ROOT (i.e. con estensione `root`) che possono contenere anche svariati istogrammi (magari suddivisi in sotto-cartelle). Dalla riga di comando scrivere quindi **`TBrowser t`**



ROOT in interattivo

Dall'interno di `TBrowser` è possibile aprire i file con estensione `root` semplicemente cliccando su di essi. Qui di seguito è riportato un banale esempio



ROOT in interattivo

Da ROOT in interattivo è anche possibile eseguire degli *script* scritti in `C++`. Come vedrete dall'esempio lo *script* non è un programma `C++` completo, mancano infatti gli `include`, ma `CINT` è in grado di capire cosa gli serve e li include automaticamente. Inoltre ROOT istanzia automaticamente una `TApplication` per gestire il `TCanvas` in interattivo

Per eseguire uno *script* l'istruzione è : `.x nome_del_file.C`

```
{
  TCanvas* c0 = new TCanvas("c0","c0",0,0,700,500);
  TH1D* h1    = new TH1D("h1","h1",100,-10,10);

  for (unsigned int i = 0; i < h1->GetNbinsX(); i++)
    {
      h1->SetBinContent(i+1,i);
    }

  h1->Draw();
  c0->Modified();
  c0->Update();
  c0->Print("test.root","root");
}
```

```
   ------------------------------------------------------------------
  | Welcome to ROOT 6.14/04                    http://root.cern.ch |
  |                                  (c) 1995-2018, The ROOT Team |
  | Built for macosx64                                            |
  | From tags/v6-14-04@v6-14-04, Aug 23 2018, 17:00:44            |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'    |
   ------------------------------------------------------------------

root [0] .x test.C
Info in <TCanvas::SaveAs>: ROOT file test.root has been created
root [1]
```