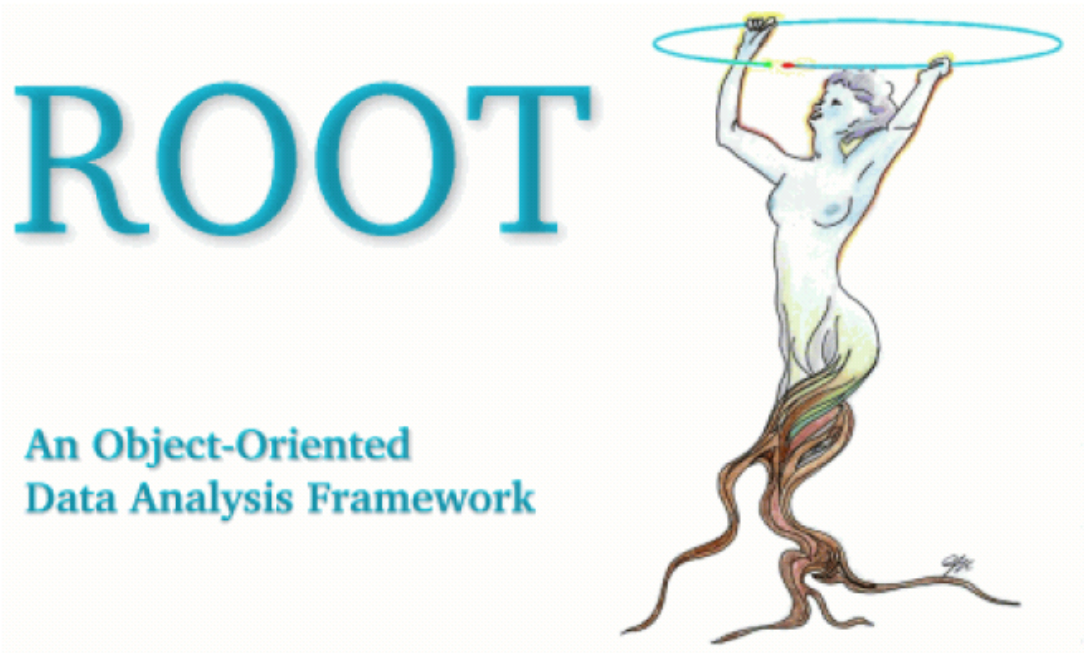
Manuale d'uso delle librerie ROOT

(per il corso di Laboratorio di Calcolo A/B)



Indice

Capitolo 1

Manuale d'uso delle librerie ROOT

ROOT è un package completo per la programmazione scientifica (comprende, tra l'altro, grafica 2D e 3D, gestione di istogrammi, algoritmi di best-fit,...) e viene distribuito per la maggior parte dei sistemi operativi oggi in uso.
ROOT può essere utilizzato sia come ambiente di lavoro interattivo che come libreria. Nel corso di Laboratorio di Calcolo utilizzeremo quasi esclusivamente la seconda possibilità: una breve introduzione all'uso interattivo è fornita nel capitolo 4.

ROOT fornisce, in maniera naturale, un'editore grafico che permette di modificare gli attributi dei grafici realizzati (colori, forma dei punti, ecc...). L'utilizzo di questo editore, basata su pannelli di comando autoesplicativi (e quindi di facile utilizzo), non sarà trattato in questo manuale.

La documentazione completa di ROOT è disponibile sul sito *http://root.cern.ch* mentre le istruzioni per installare ROOT sono disponibili sul sito del corso di Laboratorio di Calcolo (*http://www.fisica.unige.it/labc*).

1.1 Generalità dell'uso delle librerie grafiche di ROOT

La libreria di ROOT è una collezione di classi C++; data una classe TClass la creazione di un oggetto della classe e la chiamata ad un suo metodo (funzione membro della classe) è esemplificata di seguito (notare che ogni classe ha un header file separato)

```
#include <TClass.h>
int main(){
  TClass oggetto(...);
  oggetto.metodo(...);
  return 0;
}
```

la creazione di un oggetto di una della classe può avvenire anche tramite definizione ed allocazione dinamica del puntatore:

```
#include <TClass.h>
int main(){
  TClass *oggetto = new TClass(...);
  oggetto->metodo(...);
  ...
  delete oggetto;
```

```
  return 0;
}
```

Per compilare, in ambiente Linux, si usa il comando `rootlib`

```
> g++ sorgente.cpp -o eseguibile ‘rootlib‘
```

composto dalle seguenti istruzioni

```
#!/bin/sh
ROOTLIB="-Wall -O -fPIC"
ROOTLIB=$ROOTLIB" "‘root-config --cflags‘
ROOTLIB=$ROOTLIB" "‘root-config --glibs‘
ROOTLIB=$ROOTLIB" -lMinuit"
echo $ROOTLIB; export ROOTLIB
```

dove i comandi `root-config --cflags` e `root-config --glibs` definiscono rispettivamente le opzioni standard di compilazione e link necessarie a ROOT (se si utilizza TMinuit la relativa libreria va aggiunta esplicitamente).

Nel seguito saranno illustrate le principali classi di ROOT e i metodi più utilizzati. La descrizione non è completa né per quanto riguarda la documentazione sulle classi né per quanto riguarda la descrizione dei loro metodi ma risponde alle esigenze del corso.

1.1.1 Tipi

Le classi di ROOT utilizzano tipi di variabile ridefinite allo scopo di avere maggiore portabilità tra sistemi operativi diversi; ad es.

Char_t	→	char
Int_t	→	int
Double_t	→	double
Float_t	→	float
...		

Nel manuale completo di ROOT troverete i tipi elencati nella colonna di sinistra, in questa versione semplificata supporremo completa corrispondenza tra le due colonne (approssimazione più che soddisfacente per i nostri scopi) e utilizzeremo i tipi “standard”.

1.1.2 Oggetti globali

ROOT possiede una serie di variabili globali (elenchiamo qui solo quelle che useremo):

`gStyle` puntatore allo stile corrente (oggetto della classe TStyle).

`gPad` definito dopo la creazione di una TCanvas, puntatore alla pad corrente (oggetto della classe TPad)

I metodo di questi puntatori globali saranno descritti nelle relative classi.

1.2 Inizializzazione e controllo della finestra grafica

`TApplication`

Ogni programma che utilizza ROOT come libreria deve creare uno ed un solo oggetto della classe TApplication che gestisce l'interfaccia grafica e assicura le necessarie inizializzazioni.

`TApplication(const char *nome, int* argc, char **argv)`

Crea un'applicazione con identificatore `nome` a cui possono essere passate `*argc` opzioni specificate dalle stringhe di caratteri (`argv[1],argv[2],argv[n-1]`).
Per le applicazioni di Laboratorio di Calcolo non è, in genere, necessario passare alcun parametro e si consiglia di usare quindi sempre `argc=NULL` e `argv=NULL` (per maggiori dettagli sulla gestioni delle opzioni consultare la definizione di TApplication sul manuale completo di ROOT).

`void Run(bool retrn)`

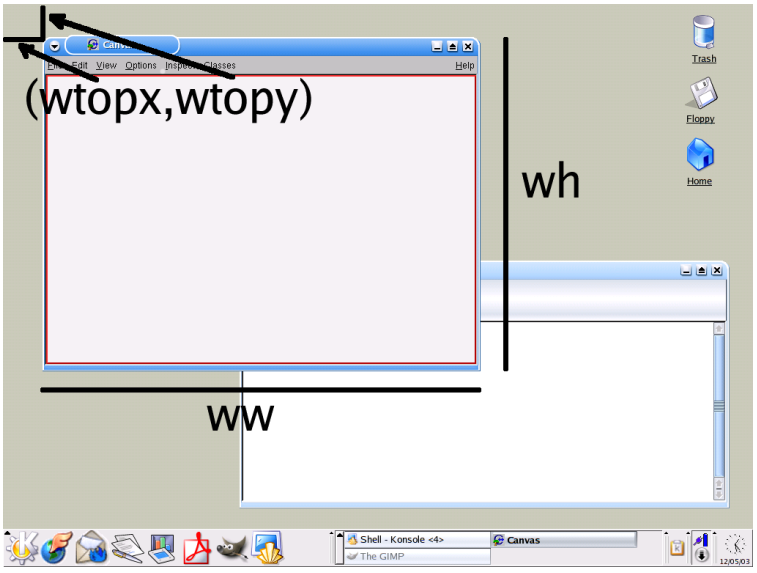
Attiva l'interfaccia grafica e consente all'utente di modificare il grafico creato (o semplicemente di bloccare l'esecuzione del programma per vederne il risultato). Dopo avere apportato le modifiche desiderate l'utente può, selezionando la voce "Quit ROOT" dal menu "File", tornare al programma principale (se `retrn=true`) o terminare l'esecuzione del programma (se `retrn=false`). Se il metodo Run() viene chiamato senza argomento la libreria assume `retrn=false`.

`TCanvas`

È la classe che gestisce le finestra grafica (canvas significa tela)

`TCanvas(const char *nome, const char *titolo, int wtopx, int wtopy, int ww, int wh)`

Crea una finestra grafica con identificatore `nome` e titolo `titolo` di dimensioni `ww`×`wh` (in pixel) avente bordo superiore destro in posizione (`wtopx`,`wtopy`) rispetto al bordo dello schermo (come schematizzato in figura);



`TCanvas(const char *nome, const char *titolo, int form)`

Crea una finestra grafica con identificatore `nome` e titolo `titolo` di dimensioni predefinite (se `form < 0` la barra del menu interattivo non è mostrata).

`form=1` 700x500 a (10,10)

`form=2` 500x500 a (20,20)

`form=3` 500x500 a (30,30)

`form=4` 500x500 a (40,40)

`form=5` 500x500 a (50,50)

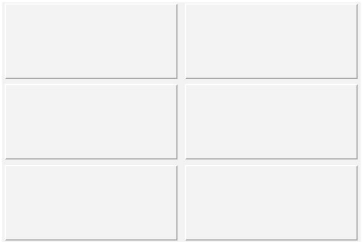
`TCanvas()`

Crea una finestra grafica 700x500 a (10,10);

`void Divide(int nx, int ny)`

Divide la finestra in `nx` zone lungo x e `ny` zone lungo y (le zone o sottodivisioni sono dette **pad**).

Esempio: le istruzioni
```
TCanvas c;
c.Divide(2,3);
```
creano una finestra e la dividono in 6 pad uguali.



Le pad sono identificate da un numero intero che va da 1 a `nx·ny`

`void cd(int npad)`

Permette di muoversi nelle pad create con `Divide()`. Dopo l'esecuzione di questo metodo la pad `npad` è la pad corrente (quella, cioè, in cui apparirà il prossimo grafico).

`void Close()`

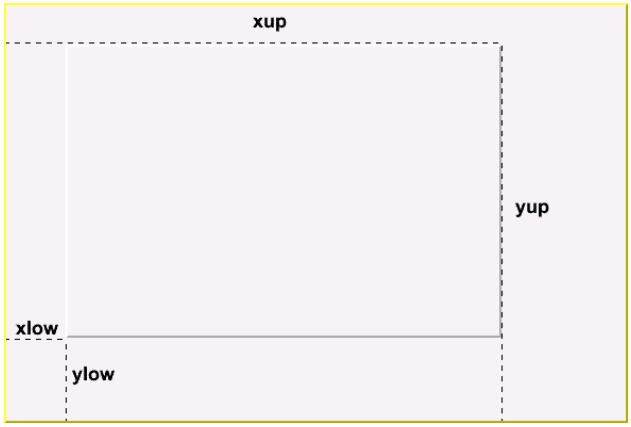
Chiude la finestra

TPad

La pad è in generale un contenitore di oggetti grafici (la canvas è un particolare tipo di pad: per questo motivo tutti i metodi di TPad sono anche metodi per TCanvas).
Ad ogni pad è associata la lista degli oggetti grafici in essa contenuti. Il "disegno" di un oggetto (associato al metodo `Draw()` comune a molte classi) non è altro che l'aggiunta dell'oggetto a tale lista (non implica, da solo, un disegno "reale" sulla finestra grafica).

```
TPad(const char *nome,const char *titolo,double xlow,double ylow,
     double xup,double yup)
```

Crea una pad di nome `nome` con titolo `titolo` posizionata, rispetto alla canvas, come mostrato in figura



`void Draw()`

Disegna la pad.

`void cd()`

Trasforma la pad a cui è applicato nella pad corrente

`void Update()`

Agisce sulla pad aggiornando il disegno di tutti i suoi oggetti se la pad è etichettata come modificata (ogni volta che un oggetto viene disegnato in una pad automaticamente questa viene etichettata come modificata).

`void Modified()`

Forza a considerare modificata la pad su cui agisce.

`void Print(const char *nomefile)`

Salva il contenuto della pad in un file in diversi formati

se nomefile è " ", il file prodotto è nomepad.ps

se nomefile comincia con un punto, nomepad è aggiunto prima del punto

se nomefile contiene .ps, produce un file Postscript

se nomefile contiene .eps, produce un file Encapsulated Postscript

se nomefile contiene .gif, produce un file GIF

`void SetFillColor(int icol)`

Definisce il colore di sfondo `icol` secondo la convenzione riportata in tavola 1.2.

`void SetGrid()`

Disegna una griglia di linee ortogonali nella pad. È utile per visualizzare facilmente il valore delle coordinate di un punto.

La pad corrente (quella su cui stiamo disegnando) è identificata dal puntatore globale `gPad`. Questo identificherà di volta in volta la canvas, una sua sottivisione creata con `Divide` o una pad esplicitamente creata dall'utente.

Aggiornamento delle pad

Per ottimizzare la velocità di esecuzione una pad non è aggiornata ogni volta che un oggetto in esso contenuta viene modificato. L'aggiornamento è automatico solo dopo la chiamata al metodo `Draw` di un oggetto.
Per forzare l'aggiormento di una pad (e poter quindi visualizzare le modifiche apportate) occorre usare le istruzioni

```
mypad.Modified()  // la pad mypad viene identificata come modificata
mycanvas.Update() // la canvas mycanvas aggiorna tutte le pad modificate
```

1.3 Grafici 2D

TGraph

Classe per grafici bidimensionali

`TGraph()`

Crea un grafico bidimensionale (senza riempirlo).

`TGraph(int n, double* x, double* y)`

Crea il grafico di `n` punti con coordinate date dai vettori `x` e `y`.

`void Draw(const char *opt)`

Disegna il grafico nella pad corrente. La stringa `opt` può selezionare le seguenti opzioni:

- `P` ogni punto è disegnato con il marker corrente;
- `L` i punti sono uniti con una spezzata;
- `C` i punti sono uniti con una curva continua;
- `A` gli assi sono disegnati (ed i loro estremi automaticamente determinati); se l'opzione `A` non è specificata il grafico viene disegnato sul grafico preesistente.

Le opzioni possono essere combinate in una unica stringa (ad es. "AP").

`void SetPoint(int n, double x, double y)`

Assegna al punto `n` (l'indice parte zero) le coordinate `(x,y)`; se la dimensione del `TGraph` è inferiore o uguale a `n`, quest'ultimo viene automaticamente ridimensionato in modo contenere `n+1` elementi.

`TAxis* GetXaxis()`

Ritorna il puntatore all'asse X del grafico.

`TAxis* GetYaxis()`

Ritorna il puntatore all'asse Y del grafico.

TGraphErrors

Classe per grafici bidimensionali con errori

`TGraphErrors()`

Crea un grafico bidimensionale con errori (senza riempirlo).

`TGraphErrors(int n, double* x, double* y,double* ex, double* ey)`

Crea il grafico di `n` punti con coordinate date dai vettori `x` e `y` con errori `ex` e `ey`.

`void Draw(const char *opt)`

(vedi `TGraph`)

`void SetPoint(int n, double x, double y)`

Assegna al punto `n` (l'indice parte zero) le coordinate `(x,y)`; se la dimensione del `TGraphErrors` è inferiore o uguale a `n`, quest'ultimo viene automaticamente ridimensionato in modo contenere `n+1` elementi.

`void SetPointError(int n, double ex, double ey)`

Attribuisce al punto `n` gli errori `ex` ed `ey`.

`TAxis* GetXaxis()`

Ritorna il puntatore all'asse X del grafico.

`TAxis* GetYaxis()`

Ritorna il puntatore all'asse Y del grafico.

TLine

Classe per le linee

`TLine()`

Crea una linea

`void DrawLine(double x1, double y1, double x2, double y2)`

Disegna, su un grafico esistente, una linea congiungente i punti di coordinate `(x1,y1)` e `(x2,y2)`

TAxis

Classe per assi dei grafici. Poiché gli assi vengono automaticamente creato dai grafici ne omettiamo il costruttore.

`void SetLimits(double min, double max)`

Modifica l'asse in modo che gli estremi siano `min` e `max`

`void SetTitle(const char *nome)`

Assegna all'asse il nome `nome`. Per l'utilizzo di caratteri speciali, lettere greche, etc... valgono le stesse opzioni di `TLatex`

TLatex

Classe per la scrittura di stringhe di testo sui grafici.

TLatex(double x, double y, const char *nome)

Crea la stringa di testo `nome` alle coordinate `x` e `y` (valori normalizzati a 1).
La stringa `opt` può contenere chiavi:

- `^` : `x^{2}` da x^2;
- `_` : `x_{2}` da x_2;
- `#` : permette di scrivere lettere greche

```
#alpha #beta #gamma #delta #epsilon #zeta #eta #theta #iota
#kappa #lambda #mu #nu #xi #omicron #pi #varpi #rho #sigma
#tau #upsilon #phi #varphi #chi #psi #omega #Gamma #Delta
#Theta #Lambda #Xi #Pi #Sigma #Upsilon #Phi #Psi #Omega
```

o simboli matematici: tra cui `#sqrt{m}` (che da $\sqrt{m}$) e gli altri simboli in tabella

≤ #leq	/ #/	∞ #infty	) #GT
♣ #club	♦ #diamond	♥ #heart	♠ #spade
↔ #leftrightarrow	← #leftarrow	↑ #uparrow	→ #rightarrow
↓ #downarrow	° #circ	± #pm	" #doublequote
≥ #geq	× #times	∝ #propto	∂ #partial
• #bullet	÷ #divide	≠ #neq	≡ #equiv
≈ #approx	… #3dots	| #cbar	¯ #topbar
↵ #downleftarrow	ℵ #aleph	ℑ #Jgothic	ℜ #Rgothic
⊙ #odot	⊗ #otimes	⊕ #oplus	∅ #oslash
∩ #cap	∪ #cup	⊃ #supset	⊇ #supseteq
⊄ #notsubset	⊂ #subset	⊆ #subseteq	∈ #in
∉ #notin	∠ #angle	∇ #nabla	® #oright
© #ocopyright	™ #trademark	∏ #prod	√ #surd
· #upoint	¬ #corner	∧ #wedge	∨ #vee
⇔ #Leftrightarrow	⇐ #Leftarrow	⇑ #Uparrow	⇒ #Rightarrow
⇓ #Downarrow	♦ #diamond	〈 #LT	□ #Box
© #copyright	™ #void3	∑ #sum	℘ #voidn
| #lbar	⌊ #arcbottom	¯ #topbar	⌈ #arctop
⌊ #bottombar	⌈ #arcbar	⎨ #ltbar	∫ #int
∥ #parallel	⊥ #perp	〉 #GT	ƒ #voidb

void Draw()

Disegna la stringa

1.4 Funzioni

TF1

Classe per le funzioni unidimensionali.

```
TF1(const char* nome, double fun(double* ,double* ),double xmin,
     double xmax, int npar)
```

Crea il grafico di una funzione parametrica (data la funzione esterna `double fun(double *, double *)`) con `npar` parametri la cui variabile indipendente è compresa nell'intervallo `[xmin,xmax]`.
Ad esempio:

```
double f(double* x, double *par){
  return par[0]+x[0]*par[1];
}
```

rappresenta la funzione $f(x) = p_0 + xp_1$ (`x[0]` è la variabile indipendente e par è il vettore di parametri).
Si noti che per scrupolo di generalità anche il prototipo della funzione unidimensionale richiede un puntatore per la variabile indipendente, rendendo uniforme il trattamento di funzioni uni e bi-dimensionali (si veda `TF2`).

```
TF1(const char* nome, const char* formula,double xmin,double xmax)
```

Crea il grafico di una funzione (parametrica o meno), con variabile indipendente nell'intervallo `[xmin,xmax]`, specificandone la formula matematica (`formula`).
Esempi:

$f(x) = x^2 + 2x$	→	`formula = "x^2+2*x"`
$f(x) = e^{3x}$	→	`formula = "exp(-3*x)"`
$f(x,p) = p_2x^2 + p_1x + p_0$	→	`formula = "[2]*x^2+[1]*x+[0]"`

```
double Eval(double x)
```

Ritorna il valore della funzione in `x`

```
void SetParameter(int i, double val)
```

Assegna al parametro `i` (l'indice parte da zero) il valore `val`.

```
void SetParameters(double* val)
```

Assegna ai parametri il vettore di valori `val`.

```
void FixParameter(int i, double val)
```

Fissa il parametro `i` (l'indice parte da zero) al valore `val` (l'azione di questo metodo ha un effetto diverso da quello di `SetParameter` solo se la funzione è utilizzata dal metodo `Fit` (vedi MINUIT)).

```
void Draw(const char *opt)
```

Disegna il grafico della funzione nella pad corrente.
La stringa `opt` può selezionare le seguenti opzioni:

`SAME` : sovrappone la funzione al grafico esistente;

`L` : i punti sono uniti con una spezzata;

`C` : i punti sono uniti con una curva continua;

La funzione, all'atto del disegno, viene campionata in un numero finito di punti (l'opzione default è `L`). Le opzioni possono essere combinate in una unica stringa (ad es. "LSAME").

```
TAxis* GetXaxis()
```

Ritorna il puntatore all'asse X del grafico.

```
TAxis* GetYaxis()
```

Ritorna il puntatore all'asse Y del grafico.

```
void SetMaximum(double max)
```

Regola a `max` il massimo dell'asse Y.

```
void SetMinimum(double min)
```

Regola a `min` il minimo dell'asse Y.

```
void SetRange(double xmin, double xmax)
```

Regola l'intervallo della variabile indipendente a `[xmin,xmax]`.

TF2

Classe per le funzioni bidimensionali.

```
TF2(const char* nome, double fun(double* ,double* ),double xmin,
      double xmax, double ymin, double ymax)
```

Crea il grafico di una funzione parametrica (data la funzione esterna `double fun(double *, double *)`) con `npar` parametri le cui variabili indipendenti sono comprese nell'intervallo `[xmin,xmax]`$\times$`[ymin,ymax]`
Ad esempio:

```
double f(double* x, double *par){
  return par[0]*x[1]*x[1]+x[0]x[0]**par[1];
}
```

rappresenta $f(x,y) = p_0x^2 + p_1y^2$ (`x[0]=x`, `x[1]=y` sono le variabili indipendenti)

```
TF2(const char* nome, const char* formula,double xmin,double xmax,
      double ymin, double ymax)
```

Crea il grafico di una funzione (parametrica o meno), con variabili indipendenti nell'intervallo `[xmin,xmax]`$\times$`[ymin,ymax]`, specificandone la formula matematica (`formula`).
Esempi:
$f(x,y) = x^2 + y^2$ $\rightarrow$ `formula = "x^2+y^2"`
$f(x,p) = p_1x^2 + p_0y^2$ $\rightarrow$ `formula = "[1]*x^2+[0]*y^2"`

`double Eval(double x, double y)`

Ritorna il valore della funzione in `(x,y)`

`void SetParameter(int i, double val)`

Assegna al parametro `i` (l'indice parte da zero) il valore `val`.

`void SetParameters(double* val)`

Assegna ai parametri il vettore di valori `val`.

`void Draw(const char *opt)`

Disegna il grafico della funzione nella pad corrente.
Le opzioni che la stringa può selezionare sono quelle di `TH2D`.

`TAxis* GetXaxis()`

Ritorna il puntatore all'asse X del grafico.

`TAxis* GetYaxis()`

Ritorna il puntatore all'asse Y del grafico.

`TAxis* GetZaxis()`

Ritorna il puntatore all'asse Z del grafico.

`void SetMaximum(double max)`

Regola a `max` il massimo dell'asse Z.

`void SetMinimum(double min)`

Regola a `min` il minimo dell'asse Z.

`void SetRange(double xmin, double ymin, double xmax, double ymax)`

Regola l'intervallo delle variabili indipendenti a `[xmin,xmax]`×`[ymin,ymax]`.

1.5 Grafici 3D

TView

Classe per la creazione di sistemi di riferimento tridimensionali

`TView(int sistema)`

Crea un sistema sistema di riferimento tridimensionale nella pad corrente (questa operazione è sempre necessaria prima del disegno di qualsiasi oggetto 3D).

```
void SetRange(double xmn, double ymn, double zmn, double xmx, double ymx,
      double zmx)
```

Dimensiona lo spazio 3D del grafico a `[xmn,xmx]`×`[ymn,ymx]`×`[zmn,zmx]`.

`void ShowAxis()`

Specifica che gli assi del sistema devono essere disegnati (tale disegno avviene comunque soltanto in seguito al disegno di un oggetto 3D).

TPolyLine3D

Classe per grafici 3D rappresentanti curve definite per punti.

`TPolyLine3D()`

Crea il grafico di un curva (senza riempirlo).

`TPolyLine3D(int n, double *x, double *y, double *z)`

Crea il grafico di una curva definita da `n` punti con coordinate date dai vettori `x, y e z`.

`void SetPoint(int n, double x, double y, double z)`

Assegna al punto `n` (l'indice parte da zero) le coordinate `(x,y,z)` se la dimensione della `TPolyLine3D` è inferiore o uguale a `n` quest'ultima viene automaticamente ridimensionata in modo da contenere `n+1` elementi.

`void Draw()`

Disegna il grafico nella pad corrente.

TPolyMarker3D

Classe per grafici 3D rappresentanti un insieme di punti.

`TPolyMarker3D()`

Crea il grafico di un insieme di punti (senza riempirlo).

`TPolyMarker3D(int n, double *x, double *y, double *z)`

Crea il grafico di un insieme di `n` punti con coordinate date dai vettori `x, y e z`.

`void SetPoint(int n, double x, double y, double z)`

Assegna al punto `n` (l'indice parte da zero) le coordinate `(x,y,z)`; se la dimensione del `TPolyMarker3D` è inferiore o uguale a `n`, quest'ultimo viene automaticamente ridimensionato in modo contenere `n+1` elementi.

`void Draw()`

Disegna il grafico nella pad corrente.

1.6 Istogrammi

Un istrogramma è un diagramma che rappresenta la distribuzione di frequenza di un certo tipo di evento in funzione di una o più variabili caratteristiche. Lo spazio delle variabili è suddiviso in intervalli regolari (canali o celle) compresi tra un valore minimo ed un valore massimo; l'istogramma è costruito disegnando, per ogni canale (o cella), un rettangolo (o un parallelepipedo in 2D) di base pari alla larghezza del canale e di altezza pari al numero di eventi che hanno variabile caratteristica contenuta in quella cella.
Diverse implementazioni di classi di istogrammi sono disponibili a seconda della precisione desiderata (`TH1C`, `TH2C` precisione char; `TH1S`, `TH2S` precisione short; `TH1F`, `TH2F` precisione float; `TH1D`, `TH2D` precisione double). Nel seguito utilizzeremo la notazione double pur essendo la descrizione comunque a tutte le classi.

TH1D

Classi per gli istogrammi unidimensionali.

`TH1D(const char* nome, const char *titolo, int ncx, float lowx, float upx)`

Crea un'istrogramma unidimensionale identificato dalla stringa `hist` con titolo `titolo` di `ncx` canali con estremo inferiore `lowx` ed estremo superiore `upx`.

`Fill(double x, double w)`

Riempe l'istogramma con il valore `x` con peso `w`.

`Draw(const char* opt)`

Disegna l'istogramma con l'opzione `opt`.
Opzioni generali:

`AXIS` : Disegna solo gli assi

`HIST` : Disegna solo il contorno dell'istrogramma (e non gli eventuali errori)

`SAME` : Sovrappone al precedente disegno sulla stessa pad

`CYL` : Usa coordinate Cilindriche

`POL` : Usa coordinate Polari

`SPH` : Usa coordinate Sferiche

`LEGO` : Disegna parallelepipedi per rappresentare il contenuto dei canali

`SURF` : Rappresenta il contenuto dell'istogramma con una superficie

Opzioni per istogrammi 1D:

`C` : Disegna una curva continua che unisce le altezze dei canali dell'istogramma

`E` : Disegna le barre di errore

`E1` : Aggiunge alle barre di errore linee perpendicolari pari alla larghezza del canale

`L` : Disegna una linea che unisce le altezze dei canali dell'istogramma

`P` : Rappresenta ogni canale con il marker corrente

`*H` : Rappresenta ogni canale con un asterisco (*)

`Reset()`

Azzera il contenuto dell'istogramma.

`double GetEntries()`

Ritorna il numero di valori (entrate) immessi nell'istogramma.

`double GetBinCenter(int ibin)`

Ritorna la coordinate del centro del canale `ibin`.

`double GetBinContent(int binx)`

Ritorna il contenuto del canale identificato dall'indice `binx`.

`int GetNbinsX()`

Ritorna il numero di canali dell'istogramma lungo l'asse x.

`double GetMean()`

Ritorna la media dell'istogramma.

`double GetRMS(int axis)`

Ritorna l'RMS (Root Mean Square = deviazione standard) dell'istogramma lungo l'asse `axis`(=1,2,3).

`TAxis* GetXaxis()`

Ritorna il puntatore all'asse X dell'istogramma.

`TAxis* GetYaxis()`

Ritorna il puntatore all'asse Y dell'istogramma.

TH2D

Classi per gli istogrammi bidimensionali.

```
TH2D(const char *hist, const char *titolo, int ncx, float lowx,
      float upx, int ncy, float lowy, float upy)
```

Crea un'istrogramma bidimensionale identificato con la stringa `hist` con titolo `test` di `ncx(ncy)` canali lungo x(y) con estremi inferiore `lowx (lowy)` ed estremi superiori `upx (upy)`.

`Fill(double x,double y, double w)`

Riempe l'istogramma con la coppia di valori `x`e `y` con peso `w`.

`Draw(const char* opt)`

Disegna l'istogramma con l'opzione `opt`.
Opzioni per istogrammi 2D (per le opzioni generali vedi `TH1D`):

`ARR` : Disegna frecce che indicano il gradiente tra celle adiacenti

`BOX` : Disegna un quadrato per ogni cella con superficie proporzionale al suo contenuto

`COL` : Disegna un quadrato per ogni cella con una gradazione di colore variabile con il suo contenuto

`COLZ` : Come `COL`. In aggiunta stampa la tabella di colori

`CONT` : Disegna curve di livello

`CONT1` : Disegna curve di livello con tipi diversi di linea per ogni livello

`CONT2` : Disegna curve di livello con lo stesso tipo di linea per tutti i livelli

`CONT4` : Disegna curve di livello riempiendo l'area tra un livello e l'altro con colori diversi

`SCAT` : Disegna uno "scatter-plot" (disegno con puntini a densità variabile)

`TEXT` : Disegna, per ogni cella, il suo contenuto in forma testuale

`Reset()`

Azzera il contenuto dell'istogramma.

`double GetEntries()`

Ritorna il numero di valori (entrate) immessi nell'istogramma.

`double GetBinContent(int binx, int biny)`

Ritorna il contenuto del canale (o cella) identificato dagli indici `binx` e `biny`.

`int GetNbinsX()`

Ritorna il numero di canali dell'istogramma lungo l'asse x.

`int GetNbinsY()`

Ritorna il numero di canali dell'istogramma lungo l'asse y.

`TAxis* GetXaxis()`

Ritorna il puntatore all'asse X dell'istogramma.

`TAxis* GetYaxis()`

Ritorna il puntatore all'asse Y dell'istogramma.

`TAxis* GetZaxis()`

Ritorna il puntatore all'asse Z dell'istogramma.

1.7 Configurazioni generali ed opzioni

Molte delle opzioni qui descritte possono essere impostate utilizzando l'interfaccia grafica attivata con il metodo `Run` di `TApplication`. È tuttavia utile, nel caso di operazioni ripetute, poter definire

le opzioni desiderate nel programma, una volta per tutte.
I colori saranno nel seguito identificati con la variabile intera `icol` secondo lo schema in Tavola 1.2, il tipo di linea con `ilin` (1 continua, 2 tratteggiata, 3 punteggiata, 4 tratteggiata-punteggiata), il tipo simbolo associato ad un punto con `imark` secondo lo schema in Tavola 1.1

TStyle

Classe che definisce lo stile dei grafici. Non viene descritto il costruttore poiché, per le esigenze di Laboratorio di Calcolo, è sufficiente eseguire metodi del puntatore `gStyle`, variabile globale di ROOT, che identifica lo stile corrente.

`void SetOptLogx(int iopt)`

Imposta la scala logaritmica (lineare) per l'asse x se `iopt`=1(0 default).

`void SetOptLogy(int iopt)`

Imposta la scala logaritmica (lineare) per l'asse y se `iopt`=1(0 default).

`void SetOptLogz(int iopt)`

Imposta la scala logaritmica (lineare) per l'asse z se `iopt`=1(0 default).

`void SetOptLogz(int iopt)`

Imposta la scala logaritmica (lineare) per l'asse z se `iopt`=1(0 default).

`void SetTitleOffset(float offset,const char *axis)`

Definisce un parametro (offset) che controlla la distanza tra gli assi ed il loro nome;

`offset` = 1 indica distanza default

`offset` = 1.2 indica che la distanza sarà 1.2×(distanza default)

`axis` specifica su quale asse operare ("x","y","x"); se `axis`="xyz" esegue l'operazione su tutti e tre gli assi.

`void SetTitleOffset(float size, const char *axis)`

Specifica la dimensione (`size`) del nome degli assi. `axis` specifica su quale asse operare ("x","y","x").

Inizia qui la lista dei metodi comuni a `TStyle` e ad altre classi. L'opzione fissata da un metodo di `gStyle` è valida per tutti gli oggetti che vengono utilizzati nel seguito; l'opzione fissata dal metodo di un oggetto è invece valida solo per quell'oggetto.

Metodi comuni a TStyle, TGraph, TGraphErrors, TPolyMarker3D

`voit SetMarkerStyle(short int imark)`

Definisce il simbolo (marker) che rappresenta i punti nei grafici secondo la convenzione riportata in tavola 1.1 (default `imark=1`).

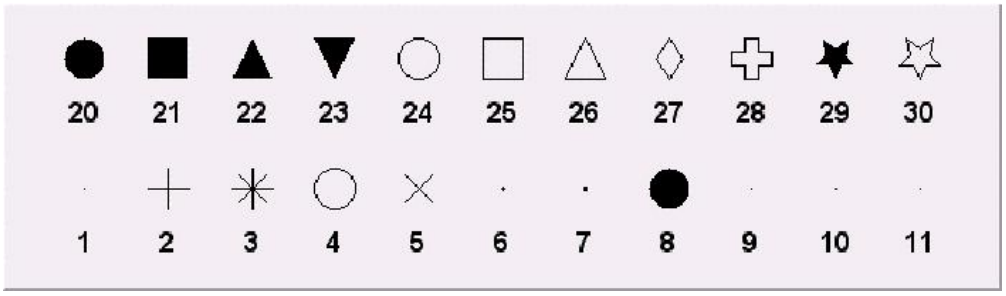


Tabella 1.1: Tavola dei simboli

Tabella 1.2: Tavola degli indici di colore

`void SetMarkerColor(short int icol)`

Definisce il colore del marker secondo la convenzione riportata in tavola 1.2 (default `icol=1`).

`void SetMarkerSize(float size)`

Definisce la dimensione del marker (default `size=1`).

Metodi comuni a TStyle, TLine, TGraph, TPolyLine3D

`void SetLineStyle(short int ilin)`

Definisce il tipo di linea; `ilin`: 1=continua, 2=tratteggiata, 3=tratto-punteggiata, 4=punteggiata.

`void SetLineColor(short int icol)`

Definisce il colore della linea secondo la convenzione riportata in tavola 1.2 (default `icol=1`).

`void SetLineWidth(float size)`

Definisce lo spessore della linea (default `size=1`).

Metodi comuni a TStyle, TF1 e TH1D

`void SetLineStyle(short int ilin)`

Definisce il tipo di linea da utilizzare negli istrogrammi; `ilin`: 1=continua, 2=tratteggiata, 3=tratto-punteggiata, 4=punteggiata.

`void SetLineColor(short int icol)`

Definisce il colore per il bordo degli istrogrammi secondo la convenzione riportata in tavola 1.2 (default `icol=1`).

`void SetLineWidth(float size)`

Definisce lo spessore del bordo degli istogrammi (default `size=1`).

Metodi comuni a TStyle e TH1D

`void SetFillStyle(short int ifill)`

Definisce lo "stile" da utilizzarsi per il disegno degli istogrammi secondo la convenzione che segue:

`ifill = 0` vuoto (default);

`ifill = 1001` pieno;

`ifill = 2001` tratteggiato;

`ifill = 300x` secondo la convenzione riportata in tavola 1.3;

`ifill = 4000` finestra trasparente;

da `4001` a 4100 si passa da una finestra 100% trasparente a 100% opaca.

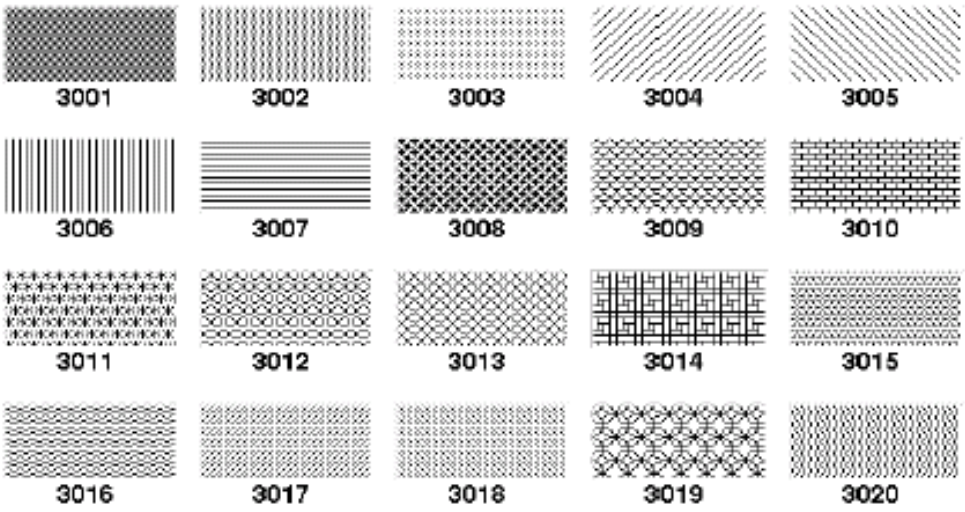


Tabella 1.3: Tavola degli stili per istogrammi

`void SetFillColor(short int icol)`

Definisce il colore da utilizzarsi per riempire l'interno degli istrogrammi secondo la convenzione riportata in tavola 1.2.

`SetOptStat(int mode)`

Definisce il tipo di informazioni mostrate nel pannello statistico dell'istogramma.
Il parametro `mode` può essere `iourmen` (default = 0):

- `n` = 1 : stampa il nome dell'istogramma
- `e` = 1 : stampa il numero di entries
- `m` = 1 : stampa il valore medio
- `r` = 1 : stampa l'RMS
- `u` = 1 : stampa il numero delle entries al di sotto dell'estremo inferiore
- `o` = 1 : stampa il numero delle entries al di sopra dell'estremo inferiore
- `i` = 1 : stampa il valore della somma dei canali

Capitolo 2

MINUIT

MINUIT è un package per la ricerca di minimi di funzioni multiparametriche.
La principale applicazione riguarda la stima, tramite il metodo del best fit (di cui i minimi quadrati sono la particolare realizzazione quando gli errori sono gaussiani), del miglior valore di uno o più parametri e dei loro errori.
L'utente fornisce a MINUIT la funzione di χ^2 che calcola la somma degli scarti quadratici tra i dati ed i valori aspettati (calcolati sulla base di una funzione teorica multiparametrica) pesati con gli errori sui dati. Gli unici parametri liberi della funzione χ^2 sono i parametri della funzione teorica. MINUIT minimizza il χ^2 rispetto a tali parametri o, in altre parole, trova i valori dei parametri che danno il minor valore di χ^2.
La classe `TMinuit` descritta nel seguito permette all'utente di avere il massimo controllo possibile sulla minimizzazione fornendo la possibilità di scrivere la funzione di χ^2 da minimizzare (questa strategia sarà quella seguita, anche motivi scopi didattici, nel corso). Può essera tuttavia utile, per semplici applicazione, specificare la funzione parametrica lasciando a ROOT il compito di calcolare e minimizzare la funzione di χ^2; questo compito è affidato al metodo `void Fit(const char* name)` comune alle classi `TGraphErrors` e `THxx`.

2.1 Breve descrizione dell'uso del metodo `Fit`

Nella descrizione che segue si utilizzerà, negli esempi, la classe `TGraphErrors` e, come funzione di fit, la retta (resta inteso che quanto detto si applica parimenti alla classe `THxx` e a tutte le altre funzioni).

2.1.1 Uso di funzioni predefinite

Le funzioni seguenti sono automaticamente create da ROOT quando è chiamato il metodo `Fit`

- **gaus** gaussiana con 3 parametri: f(x)=**p0***exp(-0.5*((x-**p1**)/**p2**)^2)
- **expo** esponenziale con 2 parametri: f(x)=exp(**p0**+**p1***x)
- **polN** polinomio di grado N: f(x)=**p0**+**p1***x+**p2***x^2+... +**pN***x^N

Esempio:

```
TGraphErrors gr(....);
...
gr.Fit("pol1");
```

Per default la funzione di fit viene aggiunta alla struttura dell'istogramma (o del grafico) e disegnata nella pad corrente. Per le funzioni predefinite non c'è bisogno di inizializzare i parametri (viene fatto automaticamente da ROOT).

2.1.2 Uso di funzioni definite dall'utente

La funzione di fit può essere un oggetto della classe `TF1` definito tramite formula o funzione esterna: il nome identificativo della funzione sarà allora utilizzato come argomento per il metodo `Fit`; l'inizializzazione dei parametri è, in questo caso, obbligatoria.
Esempio (funzione di fit TF1 definita tramire formula):

```
TGraph gr(.....);
..
TF1 myfit("myfit","[0]+[1]*x", 0, 2);
myfit.SetParameter(0, 1);
myfit.SetParameter(1, 0.05);
gr.Fit("myfit");
```

Esempio (funzione di fit TF1 definita con funzione esterna):

```
double func(double* x, double* par){
  return x[0]*par[1]+par[0];
}
int main(){
  TGraph gr(.....);
  ...
  TF1 myfit("myfit",func,0,2,2);
  myfit.SetParameter(0, 1);
  myfit.SetParameter(1, 0.05);
  gr.Fit("myfit");
  ...
  return 0;
}
```

2.2 MINUIT

TMinuit

Classe per la minimizzazione di funzioni parametriche.

`TMinuit(int npar)`

Costruttore della classe; `npar` specifica il numero di parametri.

`int SetErrorDef(double up)`

Definisce il valore di $\Delta\chi^2$ utilizzato per calcolare l'errore. Per definizione `up=1` per funzioni di χ^2 (nel corso di Laboratorio di Calcolo ci limiteremo a queste) e `up=0.5` per funzioni di Likelihood.

`void SetFCN(void fcn(int &, double *, double&, double*, int))`

Definisce la funzione di χ^2 da minimizzare. Ad esempio dati `n` punti (con errore trascurabile sulle x) ed una funzione parametrica lineare:

```
void fcn(int &npar, double *deriv, double &f, double *par, int iflag){
  f = 0;    .
  for (int i=0;i<n;i++){
    f += pow((y[i]-(par[1]*x[i]+par[0]))/ey[i],2);
  }
}
```

dove i parametri `npar` e `par` rappresentano rispettivamente il numero di parametri ed il vettore di parametri forniti a `fcn` da `TMinuit` durante le varie fasi della minimizzazione, `f` è il valore della funzione χ^2 calcolato da `fcn` che viene ritornato a `TMinuit`, `deriv` il vettore delle derivate della funzione χ^2 rispetto ai parametri calcolate dall'utente (utilizzo opzionale) ed infine `iflag` un intero che identifica le varie fasi della minimizzazione (utilizzo opzionale).

```
int DefineParameter(int parno, const char *nome,double initval,
      double initerr, double lowerlimit, double upperlimit)
```

Definisce un parametro fissandone il numero `parno`, il nome `nome`, il valore iniziale `initval`, il passo iniziale di variazione `initerr`, il limite inferiore `lowerlimit` e il limite superiori `upperlimit`.
Se `lowerlimit=upperlimit=0.0` il parametro sarà lasciato libero di variare tra $-\infty$ e $+\infty$.

```
int GetParameter(int parno, double &currentvalue, double &currenterror)
```

Ritorna il valore (`currentvalue`) del parametro `parno` ed il suo errore (`currenterror`).

```
void Migrad()
```

Esegue la minimizzazione utilizzando l'algoritmo `Migrad`.

```
void mnexcm(const char* command, double *plist, int llist, int &ierflg)
```

Esegue il comando identificato dalla stringa `command` (`mnexcm` sta per **min**uit **ex**ecute co**m**mand) utilizzando il vettore di parametri `plist` di dimensione `llist` e ritorna come codice di errore l'intero `ierflg` con la seguente codifica:

0 : command executed normally
1 : command is blank, ignored
2 : command line unreadable, ignored
3 : unknown command, ignored
4 : abnormal termination (e.g., MIGRAD not converged)
9 : reserved
10 : END command
11 : EXIT or STOP command
12 : RETURN command

I nomi dei comandi possono essere espressi in lettere maiuscole o minuscole, la più corta abbreviazione di un comando è segnalata da lettere maiuscole.
I comandi più utili sono elencati di seguito nel formato
`nome-del-comando [componenti-del-vettore-plist]`

- `MIGrad [maxcalls] [tolleranza]`
  Procede alla minimizzazione della funzione con il metodo `Migrad`, il più efficiente e completo, raccomandato per funzioni generiche. La minimizzazione produce come sotto prodotto anche la matrice degli errori dei parametri che è usualmente affidabile a meno di espliciti messaggi di errore. L'argomento opzionale `[maxcalls]` specifica approssimativamente il numero di chiamate alla funzione di χ^2 dopo cui la minimizzazione viene interrotta anche se la convergenza non è ancora stata raggiunta. Il secondo argomento opzionale `[tolleranza]` specifica la precisione richiesta per il calcolo del minimo della funzione. Il valore di default di questo parametro è 0.1, la minimizzazione si ferma quando la distanza stimata (in verticale) dal minimo è minore di `0.001*[tolleranza]*up` (per il valore di `up` vedere `SetErrorDef`). In altre parole nel caso di minimizzazione di χ^2 la precisione default con cui viene stimata il valore della funzione nel minimo è 1e-4. Una chiamata al metodo `Migrad()` precedentemente illustrato corrisponde alla seguente chiamata a `mnexcm`

  ```
  TMinuit minuit;
  double plist[1];
  int    ierflg;
  ...
  minuit.mnexcm("MIGrad",plist,0,ierflg);
  ```

  I fit eseguiti automaticamente con il metodo `Fit` utilizzano il metodo di minimizzazione `Migrad` con valore di tolleranza eguale a $10^{-6} \cdot \sum_{i=1}^{N} \sigma(y_i)^2$.
- `MINImize [maxcalls] [tolleranza]`
  Procede alla minimizzazione della funzione utilizzando il metodo `Migrad` ma converte a metodi più semplici `SIMplex` se `Migrad` non converge. Gli argomenti sono gli stessi di `Migrad`.
- `HESse [maxcalls]`
  Instruisce `Minuit` a calcolare per differenze finite la matrice degli errori. Cioè calcola la matrice completa delle derivate seconde della funzione rispetto ai parametri. L'argomento opzionale `[maxcalls]` specifica approssimativamente il massimo numero di chiamate alla funzione.
- `MINOs [maxcalls] [parnum] [parnum]` ...
  Calcola con precisione l'errore sui parametri tenendo conto di effetti di non linearità e della correlazione tra i parametri. Gli errori sono in generale asimmetrici. Il primo argomento opzionale `[maxcalls]` specifica approssimativamente il massimo numero di chiamate alla funzione, gli argomenti seguenti specificano il numero dei parametri per i quali si vuole eseguire il calcolo degli errori (per default viene eseguito per tutti).

Una tipica seguenza di minimizzazione, utile per stimare precisamente gli errori, è la sequente

```
TMinuit minuit;
double plist[1];
int    ierflg;
...
minuit.mnexcm("MIGrad",plist,0,ierflg);
minuit.mnexcm("HESse",plist,0,ierflg);
minuit.mnexcm("MINOs",plist,0,ierflg);
```

`void mnhelp(const char* command)`

Fornisce informazioni sul comando di `mnexcm` identificato dalla stringa `command`; se si omette l'argomento stampa la lista completa dei comandi.

Un esempio dell'utilizzo di MINUIT è fornito nel paragrafo 3.3.

Nel seguito presentiamo un'estratto del manuale di MINUIT che ne illustra i concetti principali.

2.3 Basic concepts of MINUIT

The MINUIT package acts on a multiparameter Fortran function to which one must give the generic name FCN. In the ROOT implementation, the function FCN is defined via the MINUIT SetFCN member function when an Histogram.Fit command is invoked. The value of FCN will in general depend on one or more variable parameters.

2.3.1 Basic concepts - The transformation for parameters with limits.

For variable parameters with limits, MINUIT uses the following transformation:

$$P_{\text{int}} = \arcsin\left(2\frac{P_{\text{ext}} - a}{b-a} - 1\right) \qquad P_{\text{ext}} = a + \frac{b-a}{2}\left(\sin P_{\text{int}} + 1\right)$$

so that the internal value P_{int} can take on any value, while the external value P_{ext} can take on values only between the lower limit a and the upper limit b. Since the transformation is necessarily non-linear, it would transform a nice linear problem into a nasty non-linear one, which is the reason why limits should be avoided if not necessary. In addition, the transformation does require some computer time, so it slows down the computation a little bit, and more importantly, it introduces additional numerical inaccuracy into the problem in addition to what is introduced in the numerical calculation of the `FCN` value. The effects of non-linearity and numerical roundoff both become more important as the external value gets closer to one of the limits (expressed as the distance to nearest limit divided by distance between limits). The user must therefore be aware of the fact that, for example, if he puts limits of $(0, 10^{10})$ on a parameter, then the values 0.0 and 1.0 will be indistinguishable to the accuracy of most machines.
The transformation also affects the parameter error matrix, of course, so MINUIT does a transformation of the error matrix (and the "parabolic" parameter errors) when there are parameter limits. Users should however realize that the transformation is only a linear approximation, and that it cannot give a meaningful result if one or more parameters is very close to a limit, where $\partial P_{\text{ext}}/\partial P_{\text{int}} \approx 0$. Therefore, it is recommended that:

1. Limits on variable parameters should be used only when needed in order to prevent the parameter from taking on unphysical values.

2. When a satisfactory minimum has been found using limits, the limits should then be removed if possible, in order to perform or re-perform the error analysis without limits.

How to get the right answer from MINUIT.

MINUIT offers the user a choice of several minimization algorithms. The `MIGRAD` algorithm is in general the best minimizer for nearly all functions. It is a variable-metric method with inexact line search, a stable metric updating scheme, and checks for positive-definiteness. Its main weakness is

that it depends heavily on knowledge of the first derivatives, and fails miserably if they are very inaccurate.
If parameter limits are needed, in spite of the side effects, then the user should be aware of the following techniques to alleviate problems caused by limits:

Getting the right minimum with limits.

If MIGRAD converges normally to a point where no parameter is near one of its limits, then the existence of limits has probably not prevented MINUIT from finding the right minimum. On the other hand, if one or more parameters is near its limit at the minimum, this may be because the true minimum is indeed at a limit, or it may be because the minimizer has become "blocked" at a limit. This may normally happen only if the parameter is so close to a limit (internal value at an odd multiple of $\pm\frac{\pi}{2}$ that MINUIT prints a warning to this effect when it prints the parameter values.
The minimizer can become blocked at a limit, because at a limit the derivative seen by the minimizer $\partial F/\partial P_{\rm int}$ is zero no matter what the real derivative $\partial F/\partial P_{\rm ext}$ is.

$$\frac{\partial F}{\partial P_{\rm int}} = \frac{\partial F}{\partial P_{\rm ext}}\frac{\partial P_{\rm ext}}{\partial P_{\rm int}} = \frac{\partial F}{\partial P_{\rm ext}} = 0$$

Getting the right parameter errors with limits.

In the best case, where the minimum is far from any limits, MINUIT will correctly transform the error matrix, and the parameter errors it reports should be accurate and very close to those you would have got without limits. In other cases (which should be more common, since otherwise you wouldn't need limits), the very meaning of parameter errors becomes problematic. Mathematically, since the limit is an absolute constraint on the parameter, a parameter at its limit has no error, at least in one direction. The error matrix, which can assign only symmetric errors, then becomes essentially meaningless.

Interpretation of Parameter Errors:

There are two kinds of problems that can arise: the **reliability** of MINUIT's error estimates, and their **statistical interpretation**, assuming they are accurate.

Statistical interpretation:

For discussuion of basic concepts, such as the meaning of the elements of the error matrix, or setting of exact confidence levels see:

1. F.James. Determining the statistical Significance of experimental Results. Technical Report DD/81/02 and CERN Report 81-03, CERN, 1981.

2. W.T.Eadie, D.Drijard, F.James, M.Roos, and B.Sadoulet. Statistical Methods in Experimental Physics. North-Holland, 1971.

Reliability of MINUIT error estimates.

MINUIT always carries around its own current estimates of the parameter errors, which it will print out on request, no matter how accurate they are at any given point in the execution. For example, at initialization, these estimates are just the starting step sizes as specified by the user. After a `MIGRAD` or `HESSE` step, the errors are usually quite accurate, unless there has been a problem.

MINUIT, when it prints out error values, also gives some indication of how reliable it thinks they are. For example, those marked `CURRENT GUESS ERROR` are only working values not to be believed, and `APPROXIMATE ERROR` means that they have been calculated but there is reason to believe that they may not be accurate.
If no mitigating adjective is given, then at least MINUIT believes the errors are accurate, although there is always a small chance that MINUIT has been fooled. Some visible signs that MINUIT may have been fooled are:

1. Warning messages produced during the minimization or error analysis.
2. Failure to find new minimum.
3. Value of `EDM` too big (estimated Distance to Minimum).
4. Correlation coefficients exactly equal to zero, unless some parameters are known to be uncorrelated with the others.
5. Correlation coefficients very close to one (greater than 0.99). This indicates both an exceptionally difficult problem, and one which has been badly parameterized so that individual errors are not very meaningful because they are so highly correlated.
6. Parameter at limit. This condition, signalled by a MINUIT warning message, may make both the function minimum and parameter errors unreliable. See the discussion above "*Getting the right parameter errors with limits*".

The best way to be absolutely sure of the errors, is to use "independent" calculations and compare them, or compare the calculated errors with a picture of the function. Theoretically, the covariance matrix for a "physical" function must be positive-definite at the minimum, although it may not be so for all points far away from the minimum, even for a well-determined physical problem. Therefore, if MIGRAD reports that it has found a non-positive-definite covariance matrix, this may be a sign of one or more of the following:

A non-physical region: On its way to the minimum, MIGRAD may have traversed a region which has unphysical behaviour, which is of course not a serious problem as long as it recovers and leaves such a region.

An underdetermined problem: If the matrix is not positive-definite even at the minimum, this may mean that the solution is not well-defined, for example that there are more unknowns than there are data points, or that the parameterization of the fit contains a linear dependence. If this is the case, then MINUIT (or any other program) cannot solve your problem uniquely, and the error matrix will necessarily be largely meaningless, so the user must remove the underdeterminedness by reformulating the parameterization. MINUIT cannot do this itself.

Numerical inaccuracies: It is possible that the apparent lack of positive-definiteness is in fact only due to excessive roundoff errors in numerical calculations in the user function or not enough precision. This is unlikely in general, but becomes more likely if the number of free parameters is very large, or if the parameters are badly scaled (not all of the same order of magnitude), and correlations are also large. In any case, whether the non-positive-definiteness is real or only numerical is largely irrelevant, since in both cases the error matrix will be unreliable and the minimum suspicious.

An ill-posed problem: For questions of parameter dependence, see the discussion above on positive-definiteness.
Possible other mathematical problems are the following:

Excessive numerical roundoff: Be especially careful of exponential and factorial functions which get big very quickly and lose accuracy.

Starting too far from the solution: The function may have unphysical local minima, especially at infinity in some variables.

Capitolo 3

Esempi di utilizzo delle librerie grafiche

Questo paragrafo raccoglie alcuni esempi, non esaustivi, dell'utilizzo delle librerie ROOT.

3.1 Grafico di un insieme di punti bidimensionali

```
#include <TCanvas.h>
#include <TGraphErrors.h>
#include <TApplication.h>

using namespace std;
TApplication  myapp("app",NULL,NULL); // crea l'interfaccia grafica

int main(){

  double  x[5]={1.,2.,3.,4.,5.};
  double  y[5]={1.,2.,3.,4.,5.};
  double ex[5]={0.01,0.01,0.01,0.01,0.01};
  double ey[5]={0.01,0.01,0.01,0.01,0.01};

  TCanvas tela;     // crea (e apre) la finestra grafica
  TGraphErrors gr(5,x,y,ex,ey); // crea il grafico
  gr.Draw("AP");    // disegna il grafico ("A"-> disegna gli assi
                    //                     "P"-> con il marker corrente)
  gr.GetXaxis()->SetTitle("X");  // Titolo sull'asse x
  gr.GetYaxis()->SetTitle("Y");  // Titolo sull'asse y
  myapp.Run(true);  // runna l'interfaccia grafica dando all'utente la possibilita'
                    // di intervenire
  tela.Close();     // chiude la finestra

  return 0;
}
```

3.2 Grafico di un insieme di punti e sovrapposizione di una funzione

```
#include <TCanvas.h>
#include <TGraphErrors.h>
#include <TApplication.h>
#include <TF1.h>

using namespace std;
TApplication  myapp("app",NULL,NULL); // crea l'interfaccia grafica

double func(double* x, double* par){
  return par[0]+par[1]*x[0];
}
int main(){

  double  x[5]={1.,2.,3.,4.,5.};
  double  y[5]={1.,2.,3.,4.,5.};
  double ex[5]={0.01,0.01,0.01,0.01,0.01};
  double ey[5]={0.01,0.01,0.01,0.01,0.01};

  TCanvas tela;     // crea (e apre) la finestra grafica
  TGraphErrors gr(5,x,y,ex,ey); // crea il grafico
  gr.Draw("AP");    // disegna il grafico ("A"-> disegna gli assi
                    //                     "P"-> con il marker corrente)
  gr.GetXaxis()->SetTitle("X");  // Titolo sull'asse x
  gr.GetYaxis()->SetTitle("Y");  // Titolo sull'asse y

  TF1 f("myfunc",func,0.0,6.0,2);// Creo una funzione (retta) a due param.
  f.SetParameter(0,0.0);         // fisso il valore del param 0
  f.SetParameter(1,1.0);         // fisso il valore del param 1
  f.Draw("SAME");                // sovrappongo la funzione al grafico
  myapp.Run(true);  // runna l'interfaccia grafica dando all'utente la possibilita'
                    // di intervenire
  tela.Close();     // chiude la finestra

  return 0;
}
```

3.3 Best fit a un insieme di punti

```
#include <TCanvas.h>
#include <TApplication.h>
#include <TGraphErrors.h>
#include <TF1.h>
#include <cmath>
#include <TMinuit.h>
#include <string>

using namespace std;
TApplication  myapp("app",NULL,NULL); // crea l'interfaccia grafica

// Per utilizzare Minuit e' necessario che le coordinate dei punti
// siano globali
#define ndat 5
double  x[ndat]={1.,2.,3.,4.,5.};
double  y[ndat]={1.,2.,3.,4.,5.};
double ex[ndat]={0.01,0.01,0.01,0.01,0.01};
double ey[ndat]={0.01,0.01,0.01,0.01,0.01};

double function(double *xcoord, double *par){
  return xcoord[0]*par[1]+par[0];
}

void chi2(int &npar, double *deriv, double &f, double *par, int flag){

  f = 0;
  for (int i=0;i<ndat;i++){
    f += pow((y[i]-function(&x[i],par)),2)/(pow(ey[i],2)+pow(ex[i]*par[1],2));
  }

}



#define npar 2

int main(){

  TCanvas tela;     // crea (e apre) la finestra grafica

  TGraphErrors gr(ndat,x,y,ex,ey); // crea il grafico

  // Minuit

  TMinuit minuit(npar);      // inizializzo con il numero di parametri
  minuit.SetFCN(chi2);       // dice qual'e la funzione che calcola il chi2
  minuit.SetErrorDef(1.);    // definisce Delta(chi2)=1 per calcolare l'errore
```

```
  double par[npar] ={0.00,0.00};
  double step[npar]={0.01,0.01};
  double min[npar] ={0.00,0.00};
  double max[npar] ={0.00,0.00};
  string cpar[npar]={"term","coeff"}; // nomi delle variabili;
  for (int i=0;i<npar;i++){
    minuit.DefineParameter(i,cpar[i].c_str(),par[i],step[i],min[i],max[i]);
    // assegna nome, val in, step, min, max per il parametro 1
  }

  minuit.Migrad();      // invoca il metodo di minimizzazione Migrad

  double outpar[npar],err[npar];
  for (int i=0;i<npar;i++){
    minuit.GetParameter(i,outpar[i],err[i]); // restituisce il valore dei parametri
  }

  TF1 func("funcplot",function,0,10,npar);   // crea il grafico di funzione a npar parametri
  func.SetParameters(outpar); // assegna i valori fittati ai due parametri della funzione

  gr.Draw("AP");      // disegna il grafico
  func.Draw("SAME");  // disegna la funzione sovrapposta

  myapp.Run(true);    // runna l'interfaccia grafica dando all'utente la possibilita'
                      // di intervenire
  tela.Close();       // chiude la finestra

  return 0;

}
```

3.4 Creazione, riempimento e grafico di un istogramma

```
#include <TCanvas.h>
#include <TApplication.h>
#include <TH1D.h>
#include <TGraphErrors.h>
#include <cstdlib>

using namespace std;
TApplication  myapp("app",NULL,NULL); // crea l'interfaccia grafica

int main(){

  TCanvas tela;     // crea (e apre) la finestra grafica

  TH1D histo("nome","titolo",100,0.0,1.0); // crea l'istogramma
  for (int i=0;i<1000;i++){
    float x = random()/(RAND_MAX+1.);  // numeri casuali distribuiti unif.
                                       // tra 0 e 1
    histo.Fill(x);  // riempe l'istogramma
  }
  histo.GetXaxis()->SetTitle("X"); // da il nome all'asse X
  histo.Draw();     // disegna l'istogramma

  myapp.Run(true);  // runna l'interfaccia grafica dando all'utente la possibilita'
                    // di intervenire
  tela.Close();     // chiude la finestra

  return 0;

}
```

Capitolo 4

Uso interattivo di ROOT

ROOT possiede anche un'interfaccia interattiva a cui si accede inviando, da terminale, il comando

```
> root
```

sul vostro schermo comparirè un finestra-logo e, quindi, un cursore con il numero incrementale `n` delle operazioni da voi eseguite

```
root [n]
```

I comandi dell'interfaccia interattiva sono istruzioni C++ (che possono contenere tutte le classi di ROOT): la differenza, rispetto alla creazione di un programma esterno che usa le librerie di ROOT, è che non occorre compilare: ogni istruzione viene "interpretata" dopo che è stato mandato il comando invio.
Per eseguire le stesse operazioni presenti in un programma C++ si possono inserire le istruzioni una ad una; è tuttavia più pratico creare un file di comandi, detto "macro", (usualmente con estensione .C) ed eseguirlo con il comando

```
root [n] .x nomefile.C
```

Si possono distinguere due tipi di macro che esamineremo nel seguito.

4.1 Macro non strutturata

Dato il file `macro1.C`

```
{
  ...
  ... istruzioni ROOT/C++
  ...
}
```

le istruzioni in esso contenute si possono eseguire semplicemente dando

```
root [n] .x macro1.C
```

4.2 Macro strutturata

Supponiamo di avere una macro contenente a sua volta varie funzioni `macro2.C`

```
void fun1(){
  ...
  ... istruzioni ROOT/C++
  ...
}
void fun2(){
  ...
  ... istruzioni ROOT/C++
  ...
}
...
```

tutte le funzioni possono "caricate" con il comando

```
root [n] .L macro2.C
```

e poi eseguite una alla volta dando (as esempio):

```
root [n] fun1()
```

In generale dato un programma costruito con le librerie ROOT questo può diventare una macro ROOT semplicemente eliminando gli header file e tutte le chiamate alla classe `TApplication` e ai suoi metodi.

Resta il fatto che l'interfaccia interattiva di ROOT è vantaggiosa soprattutto quando si devono eseguire un piccolo (o medio) numero di operazioni (l'operazione di "interpretazione" può, infatti, rallentare l'esecuzione in modo sensibile).

Infine per chiudere una sessione interattiva di ROOT si invia il comando:

```
root [n] .q
```