ROOT

Which Even Monkeys Can Use

Y.Shirasaki, O.Tajima

平成 11 年 11 月 4 日

目 次

第1章　Welcome to ROOT

ROOT の起動に必要な各設定と、ROOT の起動、終了の仕方について説明します。

1.1　前準備

まず ROOT を起動する前にいくつか準備をしなければなりません。

```
yasuhiro@s3(100)> setenv ROOTSYS /home/yasuhiro/Root
yasuhiro@s3(101)> set path=( $path ${ROOTSYS}/bin )
yasuhiro@s3(102)> setenv LD_LIBRARY_PATH ${ROOTSYS}/lib
```

1.2　起動と終了

それでは ROOT に入ってみましょう。

```
yasuhiro@s3(103)> root
```

と打つと、かっこいい ROOT のタイトル表示の後、次のように出てきて、

```
  ******************************************
  *                                        *
  *        W E L C O M E  to  R O O T      *
  *                                        *
  *   Version   2.22/09      19 July 1999   *
  *                                        *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch           *
  *                                        *
  ******************************************

FreeType Engine v1.1 used to render TrueType fonts.

CINT/ROOT C/C++ Interpreter version 5.14.9, Jul 17 1999
Type ? for help. Commands must be C++ statements.
```

```
Enclose multiple statements between { }.
```

その後に CINT の prompt が

```
root [0]
```

と出てきて ROOT に入れました。ROOT に入れたら、今度は ROOT から出てみましょう。

```
root [?]
```

が出ている時、

```
root [?] .q
```

と打つと、ROOT から出ることができます。もう一度 ROOT に入りたい時は、

```
yasuhiro@s3(104)> root
```

とすると、入ることが出来ます。

第2章　Histogram

この章では主に 1 次元のヒストグラムを書く方法について説明します。

2.1　ヒストグラムの書き方

2.1.1　1 次元のヒストグラム

まず TH1 の説明をします。これはヒストグラムを作りたい時に使います。とりあえず

```
1.6
2.0
0.23
...
```

という様なデータを用意して下さい。まずヒストグラムをつくります。

```
root [0] TH1S h1("name", "title", 150, -10, 20);
```

new TH1S()	1 次元 short int のヒストグラムを作ります。
"name"	ヒストグラムの名前
"title"	ヒストグラムの題
150, -10, 20	150 はヒストグラムの階級の数 -10, 20 はヒストグラムを描きたい範囲

さらに、作ったヒストグラム (名前は h1) にデータ (filename.dat) を読み込ませます。

```
root [1] #include <fstream.h>
root [2] ifstream data("filename.dat");
root [3] double x;
root [4] while (data >> x) h1.Fill(x);
root [5] data.close();
```

そこで、

```
root [6] h1.Draw();
```

とすると、ヒストグラムが描けます。(図 2.1)

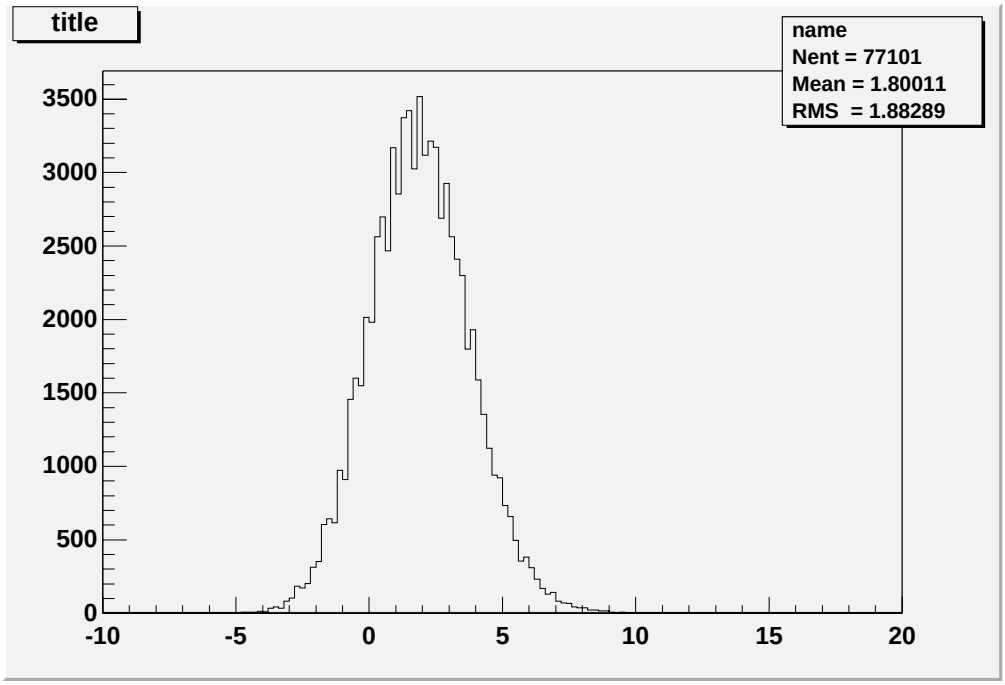


図 2.1: TH1 で描いたヒストグラム

また、いろいろなヒストグラムが描けて

```
root [7] h1.Draw(”E”);
```

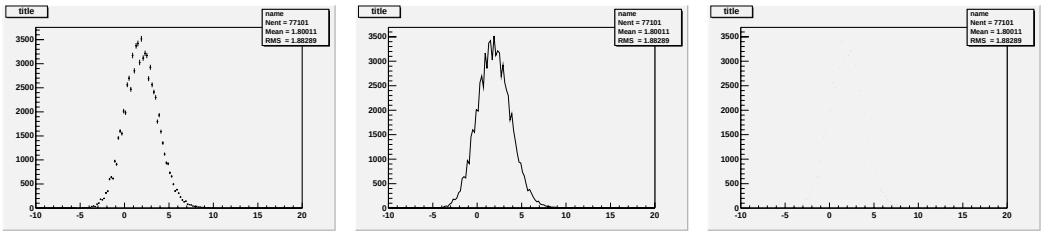
"E"を指定するとエラーバーが描けます。(図 2.2(a))

```
root [8] h1.Draw(”L”);
```

"L"を指定すると折れ線グラフになります。(図 2.2(b))

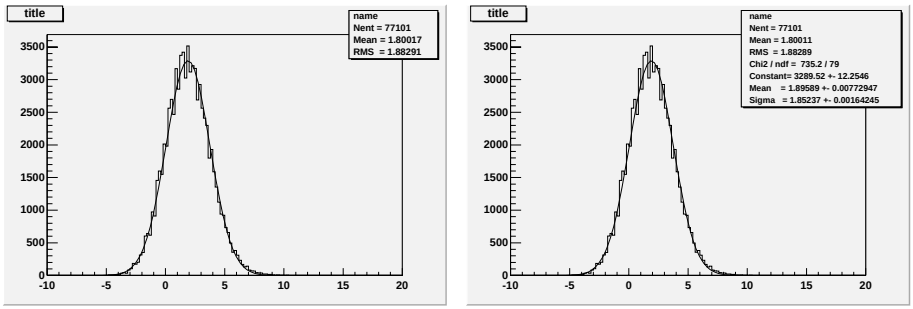
```
root [9] h1.Draw(”P”);
```

"P"を指定すると点グラフになります。(図 2.2(c))



(a) エラーバー　(b) 折れ線　(c) 点

図 2.2: TH1 で描いた様々なヒストグラム



(a)　　(b)

図 2.3: ヒストグラムでの fitting

2.1.2　ヒストグラムの fitting

h1 というヒストグラムがあるとして、

```
root [10] h1.Fit("gaus");
```

とすると、Gaussian で fitting します。引数の "gaus" というのが fitting する関数名を表しています。(図 2.3(a))

他にもいくつか標準で用意されている関数があって、

```
root [11] h1.Fit("expo");
root [12] h1.Fit("pol1");
root [13] h1.Fit("pol2");
```

"expo" は exponential fitting で、"pol1" や "pol2" というのはそれぞれ 1 次、2 次のべき関数を表し、poln(n は任意) とすると n 次の関数を用いて fitting を行うことができます。

fitting したときに、パラメータを図の上に出力したい時には

```
root [14] gStyle->SetOptFit();
```

と打っておきます。(図 2.3(b))

任意の関数での fitting

次に任意の関数に対する fitting をやってみます。例として、

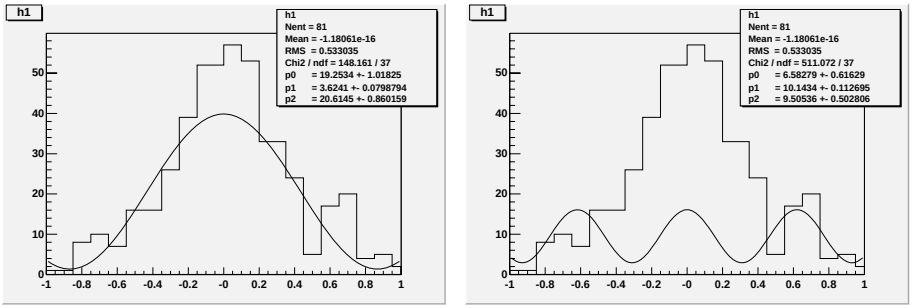
$$p_1 \cos(p_2 x) + p_3 \tag{2.1}$$

という関数でやってみます。まず 1 次元関数 TF1 を作ります。

```
root [15] TF1 f1("f1", "[0]*cos([1]*x)+[2]");
```

さらに h1 という TH1 を作っておいたとします。まず、fitting の前にそれぞれのパラメータの初期値を入れておきます。

```
root [16] f1.SetParameters(10, 1, 10);
```



(a)　　(b)

図 2.4: cos の fitting

TF1 の詳細については Function の章を見て下さい。この関数のパラメータに fitting の結果が入ります。また、10, 1, 10 が今回設定する初期値です。それで、

```
root [17] h1.Fit("f1");
```

で、fitting します。結果が図 2.4(a) です。

また、図 2.4(b) は初期値を

```
root [18] f1.SetParameters(10, 10, 10);
```

としたもので、変な初期値をほり込むとむちゃくちゃになるという例になっています。

2.1.3　一部分を取り出す

ヒストグラムの一部分だけをプロットするには TAxis の SetRange を使います。この際、SetRange の引数は bin 番号なので X の最大値、最小値を FindBin で対応する bin 番号に変換して入れてやる必要があります。例えば [-10, 20] の範囲で作られた TH1S h1 の部分 [-5, 5] を描くには

```
root [0] h1 = new TH1S("name", "title", 150, -10, 20);
:
:
root [5] h1->GetXaxis()->SetRange(h1->GetXaxis()->FindBin(-5),
                                  h1->GetXaxis()->FindBin(5));
root [6] h1->Draw();
```

とします。

第3章　Graph

この章では2次元データのプロットについて説明します。スキャッタープロットなども含まれます。

3.1　データのプロット

3.1.1　２次元データプロット

まず、はじめに

386.0	7.5384
406.0	10.7109
421.0	14.525
⋮	⋮

という様なデータ (filename.dat) を用意して下さい。

次に、これらのデータ配列とそのイベント数を用意します。例えばデータを float 配列 x[10], y[10] に読み込ませてみます。この 10 というのは配列の大きさです。配列はデータが溢れないように大きめにとりましょう。

```
root [0] #include <fstream.h>
root [1] float x[10], y[10];
root [2] ifstream data("filename.dat");
root [3] int index=0;
root [4] while(!data.eof()) {data >> x[index] >> y[index]; index++;}
root [5] data.close();
```

さて、データ配列とイベント数が用意できたらそれをプロットしてみましょう。

```
root [6] graph = new TGraph(index, x, y);
root [7] graph->Draw("AP");
```

”AP”というのはプロットする時のオプションで

A:　軸をグラフの周りに書く。
P:　データを指定したマーカー (デフォルトでは・) でプロットする。

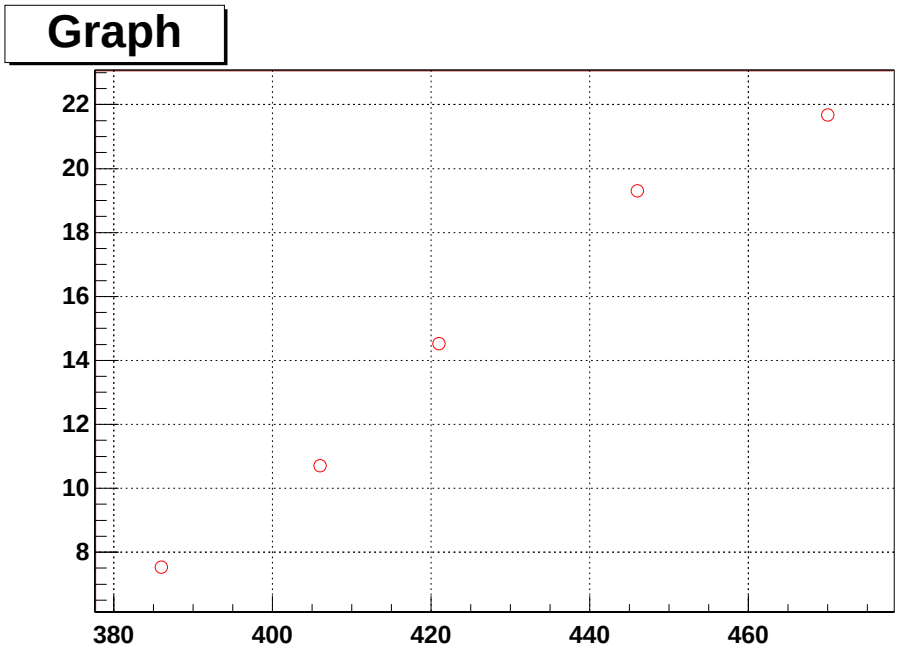


図 3.1: TGraph によるプロット

という意味です。(図 3.1)

また、いろいろなグラフが描けて

```
root [6] graph->Draw("APL");
```

"L"を指定すると折れ線グラフになります。(図 3.2(a))

```
root [8] graph->Draw("APC");
```

"C"を指定すると ∘ と ∘ の間を滑らかな曲線で結んだグラフになります。(図 3.2(b))

3.1.2　エラーバー付きのプロット

次にエラーバー付きのプロットについて説明します。先ほどの x, y のデータ配列に加えて x, y 座標のエラーの配列 x_error, y_error を用意します。ここでは既に配列にデータが読み込んであるものとします。

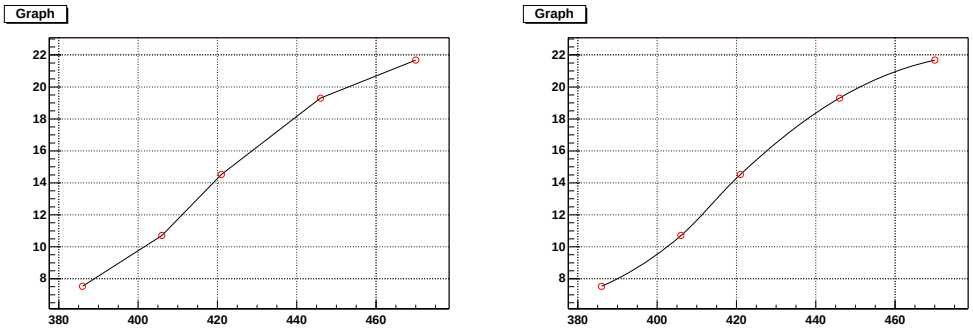
```
root [9] graph = new TGraphErrors(index, x, y, x_error, y_error);
root [10] graph->Draw("AP");
```

とすると、エラーバー付きのプロットのグラフが描けます。(図 3.3(a))

また、fitting はヒストグラムの時と全く同様に出来ます。

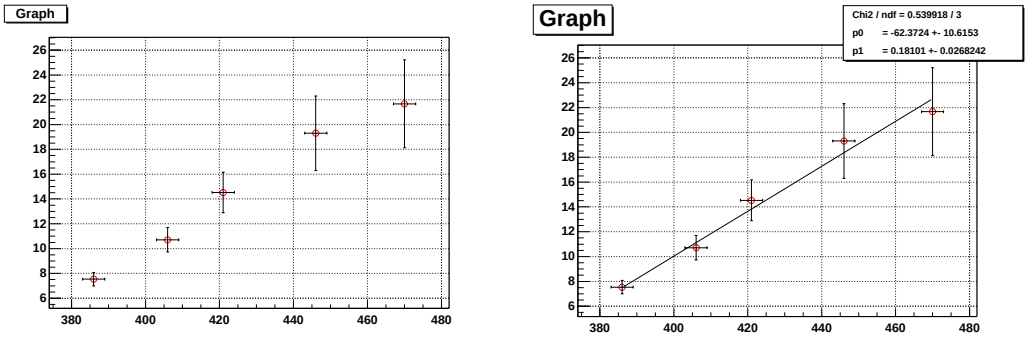
```
root [11] graph->Fit("pol1");
```

のようにすれば 1 次関数で fitting 出来ます。(図 3.3(b))



(a) 折れ線　　(b) 曲線

図 3.2: TGraph で描いた様々なグラフ



(a) エラーバー付きのプロット　　(b) 1 次関数での fitting

図 3.3: エラーバーと fitting

第4章　Function

ROOT では、色々な関数 (1 次元・2 次元) を使用、plot することができます。

4.1　TF1, TF2

関数は C の文法で入力します。ただし Fortran の如く $x**n$ で x^n を表現できるという点で拡張されています。まずは 1 変数の関数から。

```
root [0] TF1 f1("f1", "x*sin(x)*exp(-0.1*x)", -10, 10); f1.Draw();
```

とすると、関数

$$f(x) = x \sin x e^{-0.1x} \tag{4.1}$$

の plot を -10 から 10 まで行います。(図 4.1)

次に、2 変数関数を描いてみます。

```
root [1] TF2 f2("f2", "abs(sin(x)/x)*(cos(y)*y)", -6, 6, -6, 6);
```

とすると、関数

$$f(x,y) = \left|\frac{\sin x}{x}\right| y \cos y \tag{4.2}$$

の plot を x 軸方向に -6 から 6 まで、y 方向に -6 から 6 までずつ描きます (図 4.2)。(abs() というのは C の関数で絶対値をとる関数です。)

```
root [2]  f2.Draw("surf");
root [3]  f2.Draw("cont1");
root [4]  f2.Draw("lego");
```

などを用いて TH2 の時と同様に様々に描けます。

4.2　パラメータ付きの関数

関数にはパラメータを含める事ができます。fitting に使用する関数に使用する場合が多いです。パラメータ付きの関数は、普通の TF1 や TF2 と同様に作れて、ただパラメータ部分を [0]、[1]、[2] と置いていくだけです。

```
root [5] TF1 f3("f3", "[0]+[1]*exp(x)", -5, 5);
```

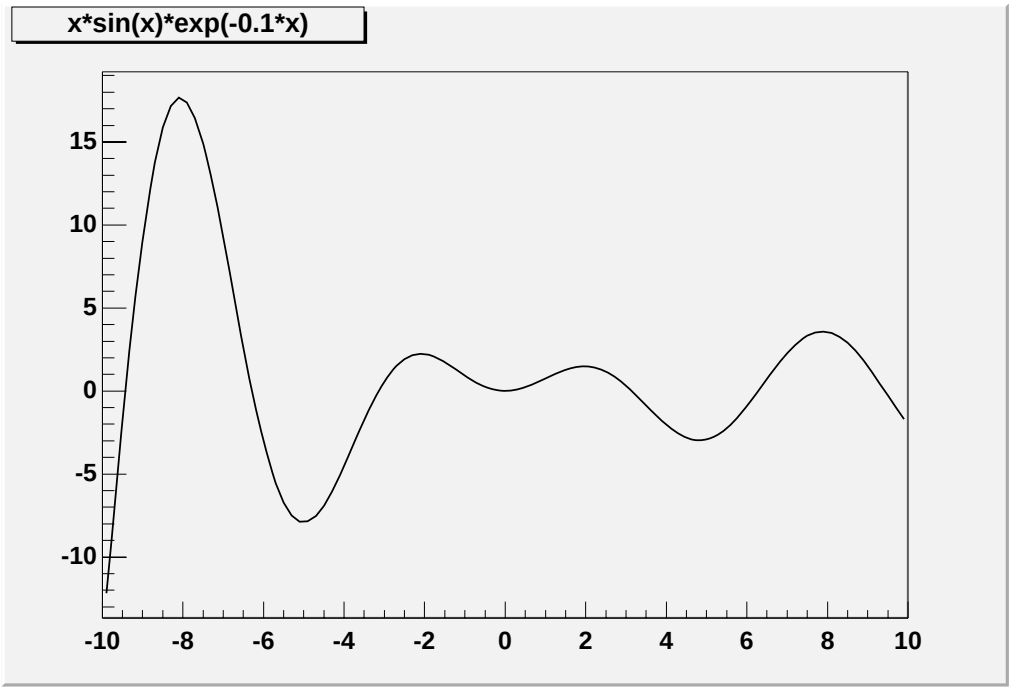


図 4.1: 1 変数関数のプロット。$f(x) = x \sin x e^{-0.1x}$

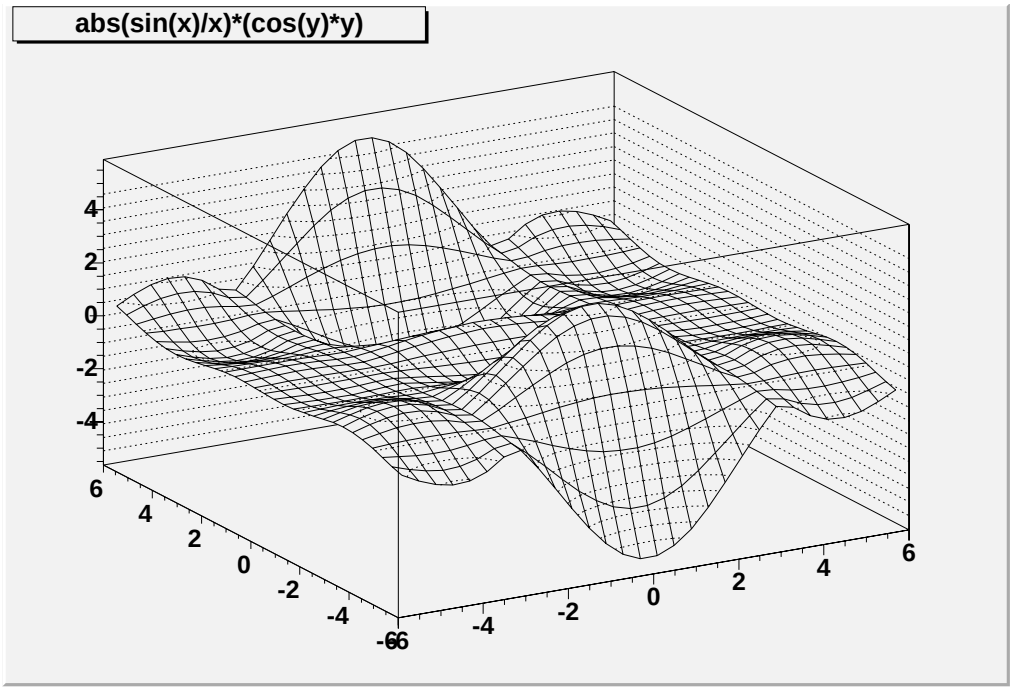


図 4.2: 2 変数関数のプロット。$f(x, y) = \left|\dfrac{\sin x}{x}\right| y \cos y$

ただし全てのパラメータの初期値が 0 になっているので、このままではただの平らな直線になってしまいます。そこで、パラメータを変更してみます。

```
root [6] f3.SetParamters(1, 2);
```

ここではパラメータ 0 に 1 を、パラメータ 1 に 2 を設定しています。

このパラメータは関数を fitting に使用する際に初期値として使用されます。

4.3　よく使う関数

比較的使用頻度の高い gaussian や landau 関数等はあらかじめ gaus や landau という名前で定義されており、これらを使用する事で、簡単に目的の関数を書く事ができます。

例えば、double gaussian

$$f(x) = 100e^{-\frac{x^2}{2}} + 20e^{-\frac{x^2}{2\times 3^2}}$$

は

```
root [7] TF1 f4("f4", "gaus(0)+gaus(3)", -10, 10);
root [8] f4.SetParameters(100, 0, 1, 20, 0, 3);
```

のように書く事ができます。

第5章　Ntuple

実際の物理解析でヒストグラムを作成するというのは、沢山のデータをいろいろな角度から見ることにより、その中に含まれる情報を見つけ出すということです。

その作業をサポートする仕組みが、Ntuple です。ntuple とは、double とか tirple、quadruple、··· の n 番目を意味し、エヌテューブルと読みます。Ntuple を利用することによって、あるデータの集合から、いろいろなヒストグラムを作成することが簡単にできます。

5.1　Ntuple の使い方

Ntuple を利用するために、まず、空の Ntuple を作成します。

```
root [0] ntuple
         = new TNtuple("name", "title", "変数リスト" [,バッファサイズ]);
```

ここで変数リストとは Ntuple で作成する変数リストで、次のように":" で区切った文字列で指定します。

"x:y:z:energy"

"px:py"

Ntuple を作成したら、ファイルからデータを読み込みます。

```
root [1] ifstream data("データファイル名");
root [2] float x, y, z, energy;
root [3] while( data >> x >> y >> z >> energy)
            ntuple->Fill(x, y, z, energy);
root [4] data.close();
```

データが読み込めたら、Ntuple の中身を確認しましょう。

```
ntuple->Print();
```

で Ntuple の内容を表示させることができます。

あとはこのデータを使って自由にヒストグラムを作成することができます。ヒストグラムを作成するには、Ntuple の Draw メソッドを使用します。

```
root [5] ntuple->Draw("変数リスト" ["条件"]" ["表示オプション"]);
```

変数リストは、ヒストグラムを作成するときに使用する変数のリストです。

”x”	x の 1D ヒストグラムを作成します。
”sqrt(x)“	演算した結果のヒストグラムを作成することもできます。 また、“:” で区
“x+y/z”	複数の変数で演算を行うこともできます。

切ることによって、2D、3D のヒストグラムを作成することもできます。

“x:y”	x と y の 2D ヒストグラムを作成します。
“x:y:z”	x、y、z の 3D ヒストグラムを作成します。 条件を与えることによっ
“x*2:sqrt(y)”	演算した結果も利用できます。

て、特定のデータのみからヒストグラムを作成することができます。

“x<0”	x が 0 未満のデータのみからヒストグラムを作成します。
“sqrt(x + y)>4”	演算した結果も利用できます。 表示の
“x>0 && z<0”	複数の条件を組み合わせることができます。

オプションは、ヒストグラムに与えるものと同じです。

例として、

```
root [5] ntuple->Draw(‘‘sqrt(x):y*2’’, ‘‘z<0’’);
```

とすると、Ntuple に読み込んだデータの中から、z が 0 未満のデータについて、sqrt(x) と y*2 の 2D ヒストグラムを作成します。

5.2　作成したヒストグラムの取扱い

作成したヒストグラムは、デフォルトでは”htemp” というテンポラリの変数に作成されますが、これを変更して独自の変数に作成することもできます。

作成される変数を変更するには、変数リストの部分に”¿¿” を追加して、

```
root [6] ntuple->Draw(‘‘sqrt(x):y*2 >> hist’’, ‘‘z<0’’);
```

の様にします。このようにすることで”hist” という変数にヒストグラムが作成されます。

また、通常は作成するたびにヒストグラムがリセットされますが、リセットせずに、既存のヒストグラムに追加したいときは、“+” を用いて、

```
root [7] ntuple->Draw(‘‘sqrt(x):y*4 >>+ hist’’, ‘‘z<0’’);
```

とします。

作成されたヒストグラムの変数名を決めたら、後はヒストグラムと全く同様に操作できます。

例えば Gaussian で fitting する時は、

```
root [8] hist->Fit(‘‘gaus’’);
```

とすればよいわけです。

5.3　条件の定義 TCut

同様な条件を与えて複数のヒストグラムを作成する際に、いちいち同じ条件を打ち込むのはあまり頭の良い方法とは言えません。そんな時は TCut を用いると便利です。使い方は、

```
root [9] TCut positiveCut = "x>0 && y>0 && z>0";
root [10] ntuple->Draw(‘‘sqrt(x):y*2’’, positiveCut);
root [11] ntuple->Draw(‘‘sqrt(x):z’’, positiveCut);
:
:
```

とするだけです。

また、複数の条件、たとえば”positiveCut” と”energyCut” を組み合わせる場合は、

```
root [12] ntuple->Draw(‘‘sqrt(x):y*2’’, positiveCut && energyCut);
```

とします。

第6章　Graphics

ここではグラフなどの上に色々な図形や文字を描く方法を紹介します。

6.1　Terminal 上や Scirpt からの描画

例えば座標上の 2 点 (1,2) と (3,4) を直線で結ぶには

```
root [1] TLine l1(1, 2, 3, 4); l1.Draw();
```

とします。これと同様にして

```
root [2] TBox b1(1, 2, 3, 4); b1.Draw();
```

とすると、左下が (1,2) で右上が (3,4) の長方形を描きます。直線を矢印にするには

```
root [3] TArrow a1(1, 2, 3, 4); a1.Draw();
```

とすると (1,2) から (3,4) への矢印 → を描きます。↔ というふうな両方向矢印が描きたい時は、

```
root [4] a1.SetOption("<>"); a1.Draw();
```

とします。また TArrow を作る時の引数を変えて

```
root [5] TArrow a1(1, 2, 3, 4, 0.05, "<>"); a1.Draw();
```

としても同じ事ができます。このときの 0.05 は矢印の大きさを表しています。(省略値が 0.05 になっています)

座標上に円を描きたい時には (ここでは中心が (4,5) で、半径が 2)

```
root [6] TArc ar1(4, 5, 2); ar1.Draw();
```

とします。ちなみに楕円は TEllipse を用います。逆に、x 軸と y 軸のスケールが違う場合には TEllipse を用いてみかけ上の正円を描く事ができます。

座標上に (例えば (2,3) に) マーカーを打つ場合は

```
root [7] TMarker m1(2, 3, 20); m1.Draw();
```

とします。最後の 20 というのはマーカーの種類です。

また、文字をいれる場合には

```
root [8] TText t1(2, 3, "Text foo bar baz"); t1.Draw();
```

等とします。

6.2　Graphics Editor

しかし、ROOT には左図のような便利なグラフィックエディタが付いているので、一度グラフを描いた後に文字や線、矢印、題字などを入れる際にはそれを用いると良いでしょう。グラフィックエディタはキャンバスの上のメニューバー中から Edit -> Editor と選ぶと起動できます。

現われたメニューの中から描きたいものを選んでクリックし、キャンバス上にマウスで始点 (と終点) を指定するだけで描く事ができます。

6.3　属性の変更

キャンバス上に描かれた文字や矢印、線、円等の太さや文字種、色などを変更したい場合にはその図形や文字の上で、マウスの右ボタンを押した時に出てくるメニュー中の下の方にある『Set ... Attributes』(... は図形によって違う) を選ぶと良いでしょう。属性変更窓が現われるので、変えたい属性をクリックして、Apply すると直ちに反映されます。

第7章　せこいわざ

基本的な図の書き方ができたので、次はその図に手を加えてみる事にします。

7.1　画面分割

一画面上に複数個の図を書きたい時には、Canvas を Pad に分割する必要があります。まず Canvas を作成してそれからそれを分割してみます。

```
root [0] TCanvas c1("Name", "Title");
root [1] c1.Divide(2, 2);
```

画面上に x 軸方向に 2 個 (最初の 2)、y 方向に 2 個 (2 番目の 2) のグラフを書きます。(全部で 4 個) 実際の図を書く前にそれぞれの Pad に移動してから図を書きます。

```
root [2] c1.cd(1);
root [3] TF1 f1("f1", "sin(x)", -2, 2); f1.Draw();
root [4] c1.cd(4);
root [5] TF1 f2("f2", "cos(x)", -2, 2); f2.Draw();
```

それぞれの Pad を更に分割する事もできます。(図 7.1)

```
root [6] c1.cd(3);
root [7] gPad->Divide(3, 2);
```

7.2　座標軸

ROOT では、勝手に座標軸を書いてくれますが、(例えばグラフの部分だけを書くためなどに) 思った通りの座標軸を書くには仮の TH1F を使ってやると便利です。

```
root [8] TH1F waku("waku", "Title", 2, 350, 550);
root [9] waku.SetMinimum(20.0);
root [10] waku.SetMaximum(110.0);
root [11] waku.Draw();
```

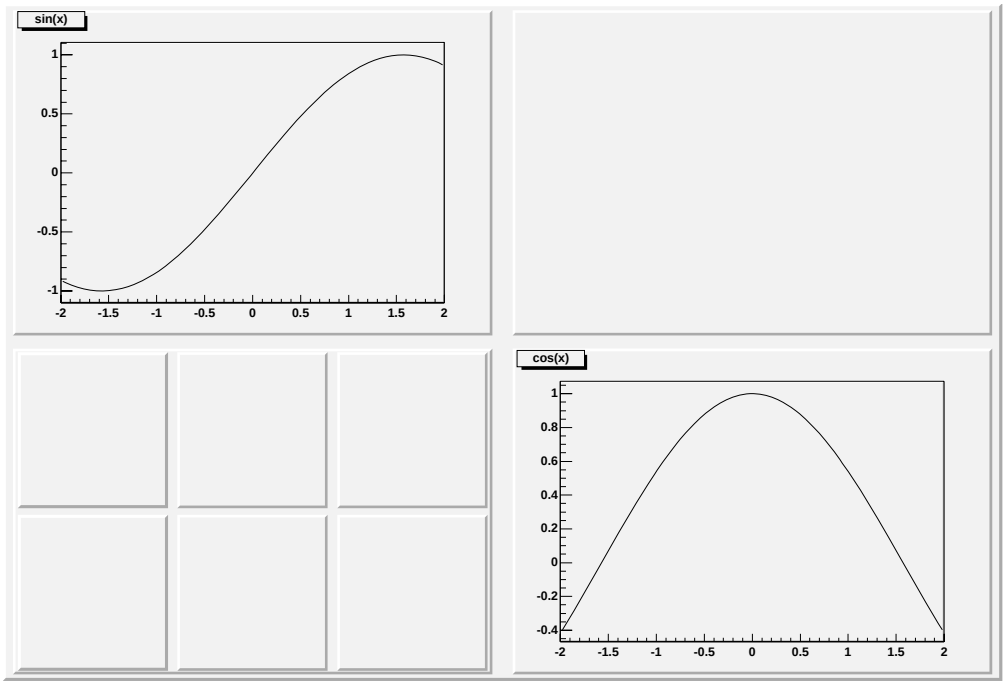


図 7.1: 画面分割の例

こうすると座標軸だけを書いて、TGraph graph を使って

```
root [12] graph.Draw("LP");
root [13] c2.Update();
```

等として重ね書きをします。TPad c2 に対して

```
root [14] c2.SetLogx();
root [15] c2.SetLogy();
```

とやると、次にこの Pad に書く図から x 軸、y 軸がそれぞれ log 目盛になります。元に戻すには

```
root [16] c2.SetLogx(0);
root [17] c2.SetLogy(0);
```

とすれば良いです。

また、

```
root [18] c2.SetGridx();
root [19] c2.SetGridy();
```

とすると座標軸上に grid を引きます。これも次にこの Pad に図を書く時から有効になります。

7.3　題

ROOT で書いた図の上にはそれぞれの図を作る際に与えた題が自動的に付きます。また、大題を付ける時には

```
root [0] TCanvas c1("c1", "Title", 50, 50, 700, 500);
root [1] TPaveLabel l1(0.3, 0.91, 0.7, 0.99, "Global Title");
root [2] l1.Draw();
root [3] TPad p1("p1", "title", 0, 0, 1, 0.9); p1.Draw(); p1.cd();
```

とやるとよいでしょう。

グラフなどを書いた時に勝手に付いてくるタイトルも

```
root [4] TF1 f1("f1", "sin(x)", -3, 3); f1.Draw();
root [5] TPaveText* t1 =  gROOT.FindObject("title");
root [6] t1.SetTextFont(131);
root [7] t1.SetX1NDC(0.4); t1.SetX2NDC(0.6);
root [8] t1.Draw();
```

とすると、書体、枠位置などを指定できます。

x 軸、y 軸のタイトル (axis title) をつけるには、TH1F h1 に対して

```
root [9] h1.SetXTitle("Wavelength");
root [10] h1.SetYTitle("Transparency");
```

とします。大題と同様に

```
root [11] h1.GetXaxis()->SetLabelFont(112);
root [12] h1.GetXaxis()->SetTitleOffset(.5);
root [13] h1.GetXaxis()->SetTitleSize(.07);
```

等とすると文字の書体や大きさが指定できます。TitleOffset は各座標軸からのタイトルの距離を指定しています。

グラフ中に文字を書きたいときには

```
root [14] TText t1(1.8, 0.44, "text"); t1.Draw();
```

とします。この例ではグラフ上の点 (1.8, 0.44) を先頭として text と表示します。書体や大きさを変えるには t1 に対して SetTextFont() や SetTextSize() 等のメンバを用います。

7.4　ヒストグラムの修飾

ヒストグラムを描く時に、色々とできて、

```
root [15] h1.SetFillColor(2);
root [16] h1.SetFillStyle(3002); h1.Draw();
```

とすると、ヒストグラムに模様が付きます。これを用いると

```
root [17] h1.SetFillColor(2);
root [18] h1.SetFillStyle(3002); h1.Draw();
root [19] h2.SetFillColor(4);
root [20] h2.SetFillStyle(3002); h2.Draw();
```

とやれば、見やすくヒストグラムを重ねられます。

ヒストグラムの太さを変えたい時は TH1 は TAttLine を継承しているので

```
root [21] h1.SetLineWidth(5); h1.Draw();
```

とします。

7.5 Graph の修飾

TGraph も TAttLine、TAttFill、TAttMaker を継承しているのでそれぞれのメンバを用いてグラフの色や太さ、線種等を自由に変更できます。

```
root [22] graph.SetFillColor(5);
root [23] graph.SetLineColor(3);
root [24] graph.SetLineWidth(4);
root [25] graph.SetMarkerColor(6); graph.Draw("ALP");
```

7.6 Global Style

ここまでに TH1 や TGraph の色々な attribute を変更してみましたが他にも色々なものがあります。これら全ては TStyle* gStyle を元に初期化されるので

```
root [26] gROOT->SetStyle("Plain");
```

として全体の見栄えを変更する事ができます。

7.7 補完

ROOT では tcsh 等のように TAB キーを使用して補完を行うことができます。例えば ROOT のチュートリアルディレクトリ (`${ROOTSYS}/tutorials`) において

```
root [27] .x gr
```

までタイプした後に TAB キーを押してみましょう。graph.C と補完されたはずです。このようにファイル名を補完する他にも、クラス名やメソッド名なども補完することができます。試しに色々な場合に TAB キーを押してみるとよいでしょう。

第8章　印刷と Sourcefile

ここでは描いた図の PostScript や EPS, GIF 形式ファイルへの出力方法と、いわゆる『マクロ』の使い方について説明します。

8.1　印刷の仕方

ROOT で描いた絵を印刷するには TCanvas c1 に対して

```
root [10] c1.Print("filename.ps");
```

とすると filename.ps というファイルができるので、これを印刷します。

EPS 、GIF 形式ファイルの作り方

Print メンバの呼び出しの際に引数を filename.eps とすると EPS 形式で、また filename.gif とすると GIF 形式で出力する事ができます。

8.2　sourcefile の作成

ROOT で絵を描く時にいちいち打つのが面倒臭い時は、source file (いわゆるマクロ) を作ると便利です。例えば graph.cc というファイルを作ります。

```
#include <fstream.h>

void graph()
{
  c1 = new TCanvas("c1", "Title", 0, 0, 700, 500);
  c1->SetGridx();
  c1->SetGridy();
  c1->SetFillColor(0);
  c1->Draw();

  waku = new TH1F("waku", "Title", 2, 350, 550);
  waku->SetMinimum(20.0);
```

```
  waku->SetMaximum(110.0);
  waku->SetDirectory(0);
  waku->SetTitleOffset(1.2);
  waku->SetXTitle("Wavelength");
  waku->SetYTitle("Transparency");
  waku->Draw("A");

  ifstream data("2dhist.dat");
  float x[500], y[500];
  int event = 0;
  while (data >> x[event] >> y[event])
    event++;

  g1 = new TGraph(event, x, y);
  g1->Draw("LP");
  c1->Update();

  c1->Print();
}
```

というようにします。この source file を用いて ROOT で絵を描くには

```
root [0] .x graph.cc
```

とします。なお、この source file では `2dhist.dat` というファイルにデータが入っていることを仮定しています。

8.3　~/.root_hist

ROOT を終了すると、.root_hist というファイルがホームディレクトリに作成されます。これは終了時までに ROOT で実行したコマンドの履歴ファイルで、これを使って source file を描くと結構便利かと思います。

第9章　ライブラリとしての ROOT

cxx や g++ 等から ROOT を class library として利用する方法を説明します。

9.1 Makefile

${ROOTSYS}/test/Makefile が良い例になっています。その環境で compile する際に必要な compiler option が CXXFLAGS として、link の際に必要な library が ROOTLIBS と ROOTGLIBS として定義されています。ですから単純な Makefile は以下のようになるはずです:

```
ROOTLIBS      = -L$(ROOTSYS)/lib -lNew -lBase -lCint -lClib -lCont \
-lFunc -lGraf -lGraf3d -lHist -lHtml -lMatrix -lMeta \
-lMinuit -lNet -lPostscript -lProof -lTree -lUnix -lZip
ROOTGLIBS     = -lGpad -lGui -lGX11 -lX3d
GLIBS       = $(ROOTLIBS) $(ROOTGLIBS) -lXpm -lX11 -lm -lPW
CXX           = cxx
CXXFLAGS      = -O -nostdnew -D__osf__ -D__alpha -I$(ROOTSYS)/include
LD            = cxx
LDFLAGS       = -g
TARGET       = test
OBJS          = main.o

all: ${TARGET}

${TARGET}: ${OBJS}
        ${CXX} ${CXXFLAGS} -o $@ ${OBJS} ${GLIBS}

.cc.o:
        ${CXX} ${CXXFLAGS} -c $<

clean:
        rm -f *.o core ${TARGET}
```

9.2　プログラム本体

最低限 TROOT object が必要です。ですから、形式的には次のようなものになるでしょう。

```
#include "TROOT.h"

class TTestBench {
(略)
};

int main(int argc, char** argv)
{
    TROOT Root("Root", "TEST Program");
    return TTestBench(argc, argv).Run();
}
```

${ROOTSYS}/test 以下に多くの例が含まれているので、これを参照しても良いでしょう。