Dis45 ユーザーのための ROOT 入門
猿にも使える ROOT：番外編

藤井恵介
高エネルギー加速器研究機構

平成 12 年 12 月 22 日

概 要

これは、白崎、田島両氏の有名な入門書「猿にも使える ROOT」のおかけでどうにかこうにか簡単なプロットができるようになった ROOT 初心者である著者が、いざ射影や、ビン幅の調節等、その一歩先へ進まんとするも思うにまかせず、試行錯誤を繰り返した結果でき上がった備忘録である。多くの場合、その解決策は自分でも目を覆いたくなるほどダサダサであるが、できないよりはましであるので恥を忍んで公開する。もっと賢いやり方を御存じの方、是非、御教示願いたい。

ROOT に関する予備知識は前提としていないが、C++ の基本的な文法についての知識は仮定している。C++ に関するちゃんとした本はどれも分厚くってめげるという人には、P.F.Kunz が高エネルギー屋のために行なった C++ 講習のノートをすすめる。また、Dis45 ユーザーのためと銘うってみたものの、実際にはそれほど Dis45 ユーザーに特化したものにはならなかった。そのおかげでと言えば言い訳になるが、そうでない人 (Paw 派の人 ?) にも少しは役に立つかも知れない。

目 次

第 1 章　なぜ ROOT を使うのか？ **1**

第 2 章　ROOT 事始 **2**

第 3 章　対話モードで ROOT を使う **5**

3.1 まずは窓を開く . . . 5
3.1.1 窓は好きなだけ開ける . . . 5
3.1.2 画面分割 (W n m) . . . 6
3.2 ヒストグラムの作成 . . . 7
3.2.1 １次元ヒストグラムを例にして . . . 7
3.2.2 Draw のオプション . . . 8
3.2.3 GUI を使ったお化粧 . . . 9
3.2.4 一様でないビン幅のヒストグラム . . . 10
3.3 プロットをいじくる . . . 10
3.3.1 プロットのリスト . . . 10
3.3.2 射影 (PROX または PROY) . . . 11
3.3.3 PROFILE . . . 12
3.3.4 上限下限 (MAMI) . . . 12
3.3.5 スライス (BANX または BANY) . . . 13
3.3.6 AVX または AVY . . . 13
3.3.7 ビン幅の変更 (CHBN) . . . 14
3.3.8 部分の拡大 (BLOW) . . . 15
3.3.9 軸を log にしたい . . . 15
3.3.10 プロット間の演算 (OPER) . . . 15
3.3.11 重ね書き (TC) . . . 16
3.3.12 統計情報の表示 (SET STAT ...) . . . 16
3.4 プロットの情報を得る . . . 16
3.5 グラフの作成：重ね書き . . . 18
3.6 GUI を使ったプロットの扱い . . . 20
3.7 フィッティング . . . 22
3.7.1 標準関数によるフィッティング . . . 22
3.7.2 任意関数によるフィッティング . . . 22
3.7.3 フィット情報の表示 (SET FIT ...) . . . 22
3.8 Ntuple の扱い . . . 22
3.9 グローバルのリセット . . . 24
3.10 ヒストリーファイル . . . 24
3.11 環境設定ファイル . . . 24
3.12 ROOT 特有のコマンド . . . 25

第 4 章 ROOT を使ったアプリケーションの開発 **26**
4.1 作法について . . . 26
4.1.1 アプリケーション . . . 26
4.1.2 コンパイルとリンク . . . 26
4.1.3 クラスの拡張と Dictionary . . . 26
4.2 使ってみよう、便利なクラス . . . 29
4.2.1 TString と TObjString . . . 29
4.2.2 コンテナクラス . . . 30
4.3 プログラム中でインタープリターを使う . . . 31
4.4 システムコール . . . 31
4.5 ROOT の拡張の実際 . . . 31
4.5.1 例題：TH1E クラスの実装 . . . 31

第 5 章 付録 **32**
5.1 ダサダサのマクロ . . . 32
5.1.1 rebin.C . . . 32
5.1.2 blow.C . . . 33
5.2 ソースからの ROOT のコンパイル . . . 33
5.2.1 コンパイルの手順 . . . 33
5.2.2 rmkdepend のパスの問題 . . . 35

第1章　なぜ ROOT を使うのか？

高エネルギーの業界には、Paw とか Dis45 とか、解析のための便利な道具がすでにある。これらは十分使い込まれており、機能も豊富である。ROOT も、それを対話的に解析を進めるためのワークベンチとして使うことを前提としており、また、後から出てきたものであるからして、機能はさらに豊富 (なはず) である (残念ながら著者はその豊富な機能を未だ使いこなせてはいないのであるが)。しかしながら、現時点における機能の豊富さよりもさらに重要な点は、その機能の拡張性である。これはひとえに ROOT のオブジェクト指向の設計による。今後、実験がより大規模になり、データー構造もより複雑化し、データ解析もさらに大規模になると予想されることを考えると、この拡張性が決定的に重要になる。Paw とか Dis45 にはじきに限界がおとずれ、そしてその限界は容易には克服できないということだ。

オブジェクト指向技術に基づくソフトウエアー開発の可能性は、もちろん ROOT だけではない。その点で ROOT の特徴はどこにあるのであろうか？ROOT は単に対話的データ処理だけでなく、オンラインのデータ収集から、オフラインでの大規模なバッチデータ処理を含むすべての実験の局面においてその「枠組み」を提供することを目指している。全ての基礎とういのが ROOT の語源のようだ。ROOT チームがしばしば強調することは、「部品」でなく「枠組み」を提供するという点である。これはプログラムに規範を与え、統一性を維持する。多人数による大規模なプログラムの開発にとっては重要な点である。

しかし、実際問題として最も重要な点は、それが完全にフリーであることであろう。ROOT は CERN の ROOT チームによって開発が進められており、非商用目的での使用は無料である。また、ソースも完全に公開されており変更も自由である。これだけのものがただで使えるというのは、驚きである。

ROOT のホームページ

```
http://root.cern.ch/
```

をご覧になることを勧める。ROOT に関する様々な情報や、最新版の入手が可能だ。

これまでライセンスが必要であった CERNLIB も最近 GPL 化され、無償で手にはいるようになったが、これはむしろ CERNLIB の役割もほぼ終りに近づいたためと見るべきであろう。

能書きはこれぐらいにして、実際の ROOT の使用感について少しコメントしておく。対話的な使い方でまず気付く著しい特徴は、コマンドライン言語が C++ であることだ。日本ヒューレットパッカードの後藤さんという方が開発された内蔵の C++ インタープリターによって、標準入力を 1 行 1 行 C++ プログラムとして実行する。つまり、対話的に C++ の解析プログラムが開発でき、必要とあればそれを後でバッチプログラムとして実行できるわけだ。Paw や Dis45 と比べると少しタイプの量が増えるという欠点もあるが、それとても C++ を学ぶものにとっては格好の道場となりうる。それに、C++ プログラムをマクロとしてロードして実行できるので、よく使う手続きはマクロ化しておけばよいわけである。コマンドライン言語が C++ という完全なプログラミング言語であるということは、対話セッションでありながらすさまじい自由度を持っているということだ。やろうと思えば、対話セッションの中で、ほとんど何でもできてしまうのである。

一方、ROOT のセッションで作成されたプロットはマウスで いろいろ編集できる。線やフィル、文字の属性を GUI で簡単に変えられ、カラーのきれいな図を容易に EPS フォーマットで用意できる点は、Paw や Dis45 と比べて大分使い勝手が向上している。

それでは早速使ってみよう。

第2章　ROOT 事始

まず、ROOT をインストールしなくてはならないが、tar の展開だけなので何のことはない。著者の場合、/opt 以下に展開している。

```
# tar -zxvf root_v2.25.02_..._tar.gz -C /opt
```

ソースファイルからコンパイルしたい場合 (ソースをいじりたい場合) については付録に簡単に説明した。動作環境については、ROOT のホームページに説明があるが、以下の説明では基本的に LINUX で bash 環境を使っていると仮定している (著者の環境は linuxppc 2k)。そうでない環境の人は適当に読み変えること。

次に若干の環境変数の設定。

```
$ export ROOTSYS=/opt/root
$ export PATH=$PATH:$ROOTSYS/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib
```

いちいち手で打つのが面倒なら、これらは、.bash_profile か .bashrc に書いておいても良い。これで、取り敢えず準備完了だ。

まずは /opt/root/tutorials にあるデモを走らせてみよう。適当なディレクトリーに例題をコピーする。

```
$ cd <somewhere>/
$ cp -R $ROOTSYS/tutorials .
$ cd tutorials
```

ROOT の 対話セッションを開始するには、

```
$ root
```

と打てば良い。ロゴが表示されセッションが始まる。

図 2.1: ROOT のロゴ

このロゴがうっとうしければ、−l をつける。

```
$ root -l
```

root コマンド自体のコマンドラインオプションは −? で見られる。こんな感じだ。

```
$ root -?
Usage: root [-l] [-b] [-n] [-q] [dir] [file1.C ... fileN.C]
Options:
  -b : run in batch mode without graphics
  -n : do not execute logon and logoff macros as specified in .rootrc
  -q : exit after processing command line macro files
  -l : do not show splash screen
 dir : if dir is a valid directory cd to it before executing
```

−b や −q は、バッチモードで解析をする際にいずれお世話になる便利なオプションである。特にイベント毎にプロットの更新をすると大変時間がかかるので、−b は助かるオプションである。

さて、ROOT の対話セッションが始まると、

```
  ****************************************
  *                                      *
  *        W E L C O M E  to  R O O T    *
  *                                      *
  *   Version   2.25/02    23 August 2000   *
  *                                      *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch         *
  *                                      *
  ****************************************
CINT/ROOT C/C++ Interpreter version 5.14.47, August 12 2000
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
```

と出てくる。root[0] というのが、root のインタープリター (RCINT) のプロンプトである。[0] は打ち込んだコマンドラインの番号で、以前のコマンドのヒストリーは、普通のシェル環境のように、矢印の上下ボタンで行ったり来たりできる。CNTRL + P または CNTRL + N でも良い。実際、ROOT のコマンドラインは emacs のような編集ができ、また TAB をたたけば、コマンドライン補間もしてくれる。クラスとか関数、変数の名前なども補間してくれるのでとても便利である。

さて、早速デモを走らせてみよう。

```
root [0] .x benchmarks.C
root [1] .q
```

.x はプログラム benchmarks.C のロードと実行、.q が ROOT の対話セッションの終了である。既に述べたように、コマンドライン補間が効くので、.x be あたりまで打って TAB をたたけば .x benchmarks.C までででてくるはずである。そこでリターンキーを押せば、ROOT のベンチマークが実行される。これは ROOT でできることの例題の集合であり、見ていて結構楽しいし、対話セッションでできることのおおよその感じがつかめる。.x demos.C とやれば、ほぼ同じ例題のセットが対話的に実行できる。ちなみに、図 2.2 は、hsum というボタンを押したところ。

基本的に、"." で始まるのが対話セッション独自のコマンドで、それ以外は RCINT という内蔵の C++ インタープリターに C++ プログラムとして渡される。

次章より、仕事に使おうとしたときやりたくなることについてどうしたらできるか考えてみよう。

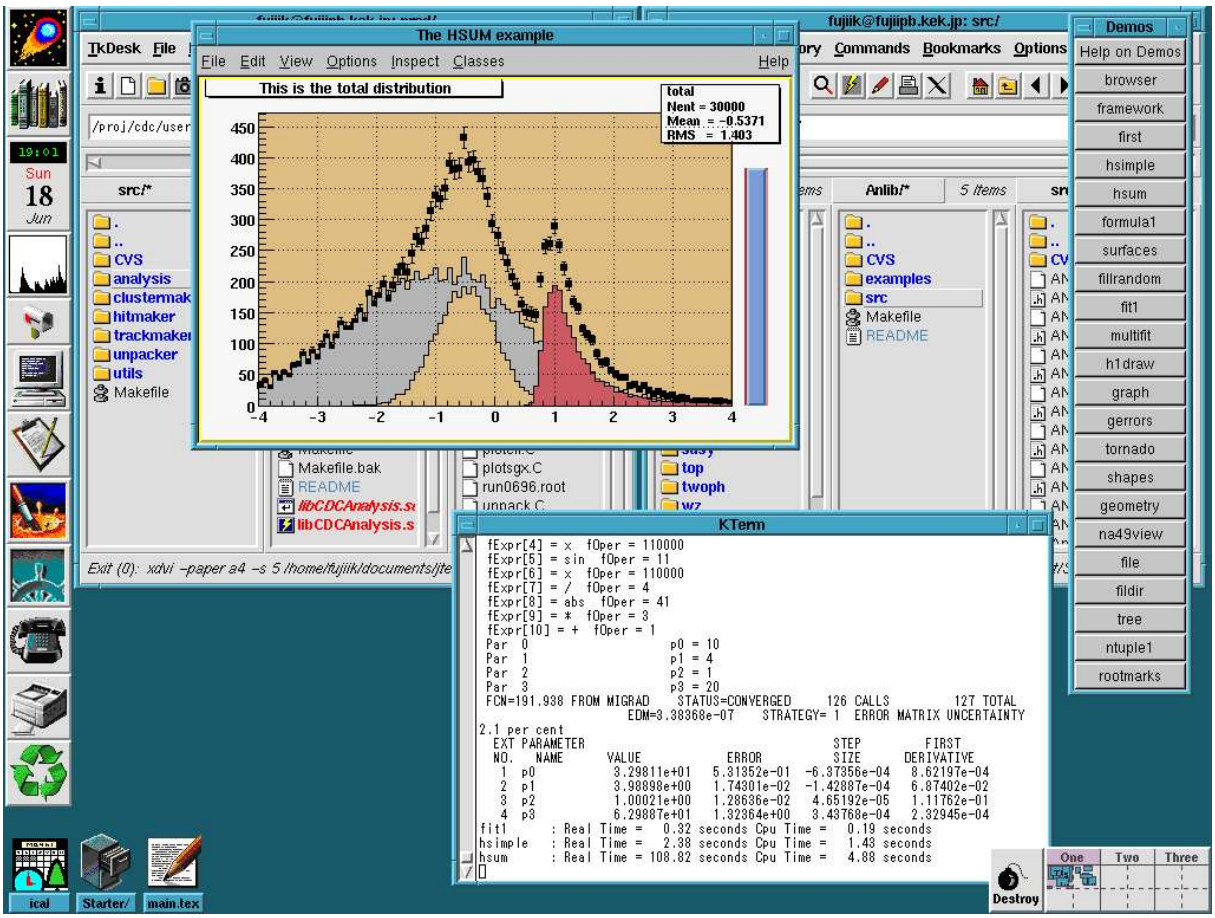

図 2.2: demo.C の実行画面。hsum のボタンを押してみたところ。実際にはヒストグラムがフィルされるようすが動的に見られる。動くところが見せられないのが残念である。

第3章　対話モードで ROOT を使う

3.1　まずは窓を開く

3.1.1　窓は好きなだけ開ける

dis45 と違って、ROOT のセッションでは、グラフなどを表示する窓 (TCanvas オブジェクト) はいくらでも作れる。

```
TCanvas *c1 = new TCanvas("c1","My Canvas",10,10,400,400);
```

などとすると、”c1” という名前の、”My Canvas” というタイトルのキャンバスが X の座標 (左上が原点) で (10,10) を起点にして 400 ドット× 400 ドット の窓が開き、フォーカスがこの窓に移る[1]。

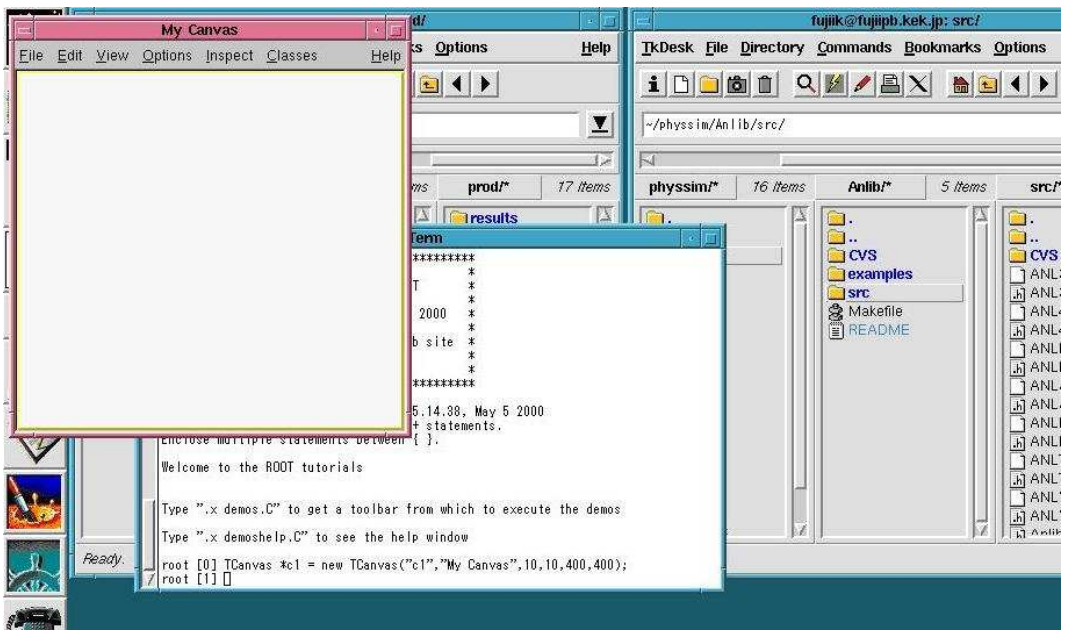

図 3.1: TCanvas の作成。タイトル、X の画面上での配置に注意。

もちろん

```
TCanvas c1("c1","My Canvas",10,10,400,400);
```

でも良い。この場合は、c1 は実体である。

実は、ROOT の対話セッションの中では、ポインターと実体との区別がなかったり、宣言なしの変数に代入できたり、行末の ”;” を省略できたりいろいろショートカットがあるが、著者の場合、悪い習慣を覚えるといけないので C++ の勉強のつもりでちゃんと打つことにしている。

[1]今後、ROOT のコマンドラインのプロンプトは省略する。自分で打ち込む部分だけ書くのでそのつもりで。

3.1.2 画面分割 (W n m)

通常、dis45 を使っている場合には、ウインドウが開くや否や画面分割にかかる。自然に手が "w 2 2" とか動いてしまうのである。Paw でいうところの "zone 2 2" である。

これを ROOT でやる場合は

```
c1->Divide(n,m);
```

で、このキャンバスを n 列、m 行の区画に分ける。この時点では、４つの区画全体のまわりに黄色 (とは限らな

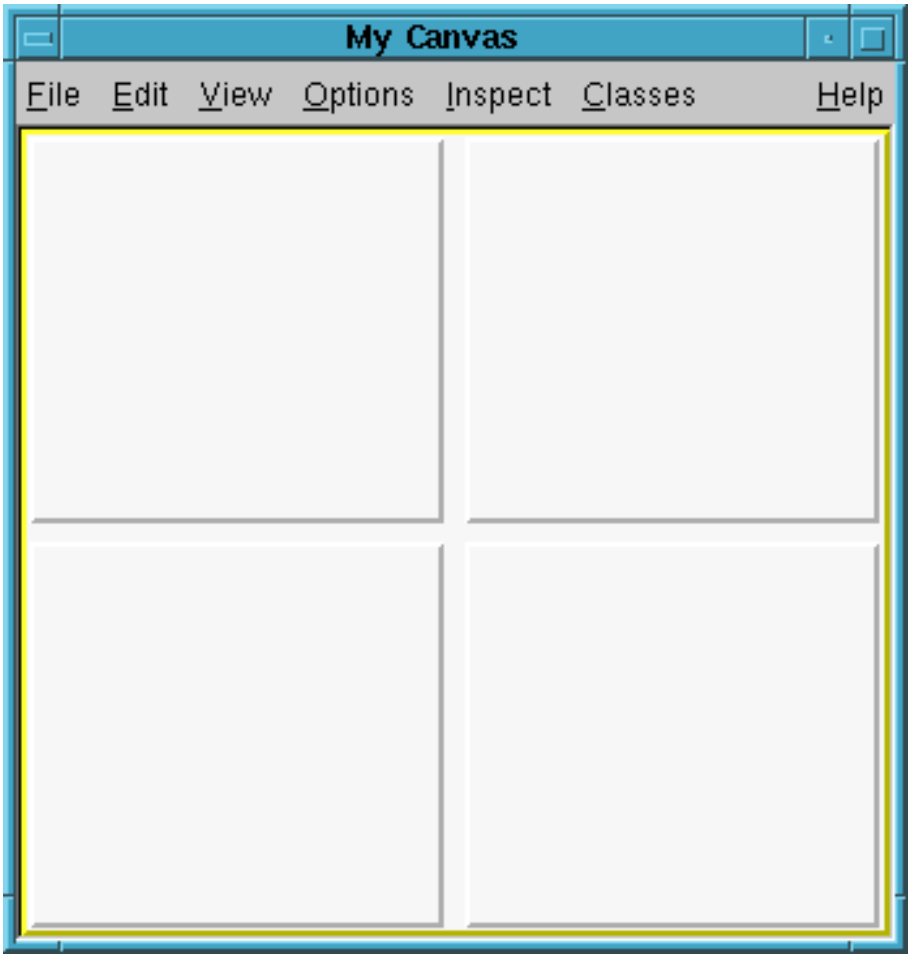

図 3.2: TCanvas の分割。c1-> Divide(2,2) の結果。

いが) の枠がついている。これは、c1 に対応するトップディレクトリーにいることを示している。

各々の区画はサブディレクトリーの扱いである。i 番目の区画で絵を書こうとする場合には、

```
c1->cd(i);
```

などとして、その区画に移ってから作業をしなくてはいけない。その区画の枠の色が変わって、ディレクトリーの移動が起きたことが分かる。プロットの描画は、現在いるディレクトリーに対して起きるので、移動せずに描画すると、上書きしてしまい悲しい思いをする。

3.2　ヒストグラムの作成

このあたりは「猿にも」に詳しく書かれているので繰り返しになるが、各自の解析プログラムで繰り返し書くことなので、基本的な手続きをおさらいしておく。

ROOT のヒストグラムは、全て TH1 という基底クラスから派生してできている。1 次元ヒストグラムの場合、ストレージの型により Char_t 型の TH1C、Short_t 型の TH1S、Float_t 型の TH1F、Double_t 型の TH1D がある[2]。2 次元 (TH2)、3 次元 (TH3) の場合も同様である。次元によらず、ユーザーインターフェースの仕様は、基底クラスである TH1 で規定されている。

3.2.1　1 次元ヒストグラムを例にして

まずは、ヒストグラムの箱を作成する。どの型でも扱いは同じなので、Float_t 型の 1 次元ヒストグラムを例にとれば、

```
TH1F *h1 = new TH1F("name","title",60,-10.,20.);
```

のような感じである。ここで、"name" は、TNamed クラスの派生クラスとしての名前であり、あとで見るように、ポインターを知らなくてもこの名前をたよりにオブジェクトを探すことができる。"title" はヒストグラムの題であり、60 はビンの数、$-10.$ と $20.$ は横軸の下限と上限である。

TCanvas の場合と同様、

```
TH1F h1("name","title",60,-10.,20.);
```

のように作っても良い。ただし、プロット関連の ROOT のクラスのメンバー関数はその戻り値として新たにできたプロットのポインターを返すことが多いので、混乱を避けるため、著者の場合、自分で作る際もポインターを使うことにしている。

さて、箱ができたのでこのヒストグラムにデータをフィルしてみよう。$x = 5.$ のところにデータをフィルするには

```
Double_t x = 5.;
```

として、

```
h1->Fill(x);
```

で良いわけだが、たくさんフィルするのに手ではやってられない。たくさんデータをフィルしたければ、「猿にも」にあるようにファイルから読み込んだりすれば良いわけだが、ここでは、練習なので、正規分布乱数を使ってフィルしてみる。

```
gRandom->SetSeed();
Int_t i;
for (i=0; i<10000; i++) h1->Fill(gRandom->Gaus(5.,3.));
```

この例では、乱数のシードをセットした後、中心 5. で、幅 3. の正規分布乱数 10000 点をフィルしている。$gRandom->SetSeed()$ は省略できるようである。乱数には他にも、Exp とか Landau とか Poisson とか Binomial とかいろいろあるので試してみるのも一興である。今の例では、for ループを 1 行で書くことができたが、複数行にわたるコマンドを書きたい場合も生ずる。そんな場合は "{" をうち複数行モードにはいってから入力する。複数行モードの終は "}" である。

ヒストグラムは重みつきでフィルすることもできる。

[2] ROOT では、マシン依存性を除くため、生の型、char、short、float、double などを使わないという習慣になっている。

```
Double_t w = 0.5;
h1->Fill(x,w);
```

とかすると、ビンは 1 ではなく w だけ増えることになる。これらのフィルの操作は、後で見るように 2 次元ヒストグラムに関しても同様である。

さて、フィルの結果を描画するには

```
h1->Draw();
```

と打つ。

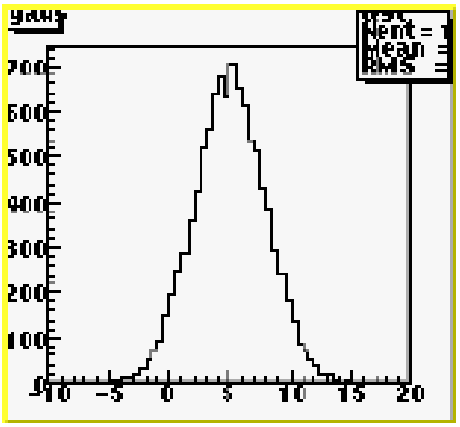

図 3.3: ヒストグラムの描画。

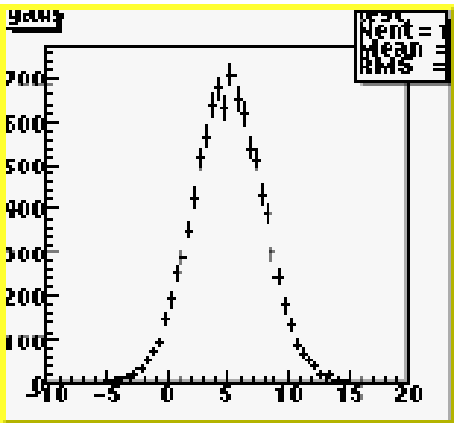

図 3.4: 誤差棒をつける。

味もそっけもないヒストグラムである。

3.2.2 Draw のオプション

実は、ヒストグラムクラスの Draw メンバー関数にはいろいろオプションを与えることができる。例えば

```
h1->Draw("e");
```

とすると誤差棒をつけることができる (図 3.4)。

オプションにはいろいろあるが、著者がよく使うのは

e	誤差棒
p	プロット
c	滑らかな曲線で結ぶ
l	折れ線で結ぶ

などである。

それ以外によく使うオプションとしては ”same” がある。これは複数のプロットの重ね書きに使う。

```
h1->Draw("same");
```

フロットマーカーの種類、サイズ、色の変更は

```
h1->SetMarkerStyle(21);
h1->SetMarkerSize(1);
h1->SetMarkerColor(2);
```

などと行なえるが、次に述べる GUI を使った方が簡単であろう。

3.2.3 GUI を使ったお化粧

プロットの属性 (見栄え) に関する変更は、そのほとんど全てについて GUI でできる。これは、およそ全ての操作がコマンドラインベースの Dis45 と比べると非常に助かる (Paw++ では GUI は結構使えるが)。

ヒストグラムの境界線またはプロットのデーター点をマウスで触ると、カーソルが十字から矢印に変わる。そこで右ボタンを押せばコンテクストメニューが出る。

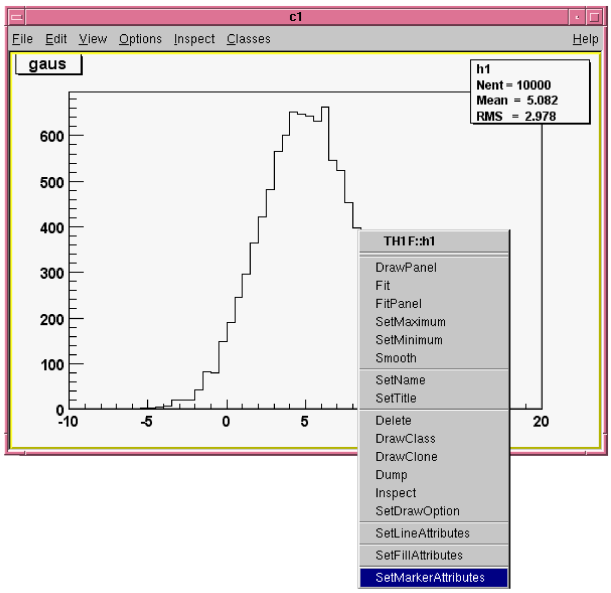

図 3.5: GUI によるヒストグラムの属性の変更。SetMarkerAttributes を選んだところ。

コンテクストメニューの内容については、いろいろ試してみるのが手っ取り早い。もっぱら見栄えに凝るなら、SetLineAttributes、SetFillAttributes、SetMarkerAttributes が役に立つ。SetLineAttributes は、ヒストグラムや誤差棒の線の太さや色など線の属性、SetFillAttributes は、塗りつぶしのオプション、SetMarkerAttributes は、プロットのマーカーの変更のためのダイアログである。これらの GUI の使い方に説明は不要であろう。

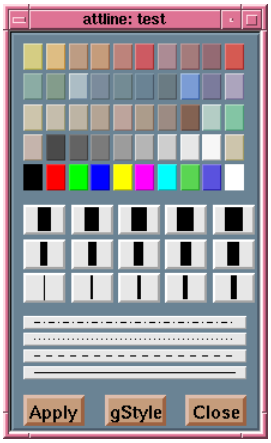

図 3.6: SetLineAttributes

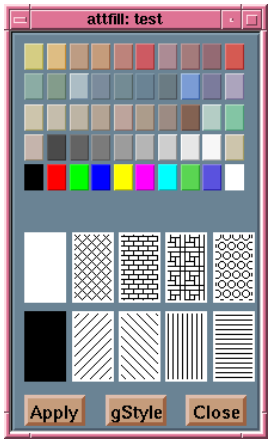

図 3.7: SetFillAttributes

図 3.8: SetMarkerAttributes

さらに、ラベルやタイトル、背景や軸についても右ボタンでクリックしてみると幸せになれるかも知れない。いろいろ試してみられたい。

3.2.4 一様でないビン幅のヒストグラム

いままでは、ビン幅が一定の場合を扱ってきた。時には、統計の少ないビンをまとめたりして、ビン幅が一様でないようなプロットを作りたい場合があろう。そんな場合は別のコンストラクターを使うことになる。

```
Int_t    nbins = 5;
Double_t xbins[] = {0.,1.,2.,3.,5.,10.};
TH1F *h1 = new TH1F("name","title",nbins,xbins);
```

などとすると、各ビンの下限が、0.、1.、2.、3.、5. で最後のビンの上限が 10. のヒストグラムの箱ができる。xbins の大きさは nbins + 1 であることに注意。

3.3 プロットをいじくる

データー解析でやっていることと言えば、基本的には、いろいろなプロットを作ってそれを眺め、射影したり、スライスしてみたりしていじくり回し、何とか意味のある情報を引っ張り出すことである。ここでは、ROOT でそれをどうやるか考えてみる。

3.3.1 プロットのリスト

プロットをいじる前にどのプロットをいじるか知る必要がある。

```
gDirectory->ls();
```

とすれば、現在ディレクトリーにある名前つきオブジェクト (TNamed の子孫) のリストが見られる。

ROOT のプロットは全て TNamed クラスから派生している。つまりおのおの名前がついており、ポインターを失っても探すことが可能だ。例えば、"MyPlot" などという名前のついた倍精度型の 1 次元ヒストグラムの場合は、

```
TH1D *h1 = (TH1D *)gROOT->FindObject("MyPlot");
```

などとすれば、ポインターを知ることができる。これは

```
((TH1D *)gROOT->FindObject("MyPlot"))->Draw();
```

などとできることを意味する。

これに対する ROOT 独特のショートカットとして

```
MyPlot->Draw();
```

も許される。が、悪い習慣が身についてしまうといけないので、初めのうちはちゃんとした C++ を使うようにした方が良いと思う。この便利さの誘惑は抗し難いが。

逆にポインターを知っていれば

```
cout << h2->GetName() << endl;
```

とすれば名前が分かる。

ヒストグラムを含む ROOT のファイルを

```
TFile *file = new TFile("hogehoge.root");
```

のようにロードした場合は、

```
file->ls();
```

とすることでもリストが得られる。

3.3.2 射影 (PROX または PROY)

2 次元プロットの各軸への射影はよくやる操作だが、ROOT では X 軸への射影は

```
h2->ProjectionX()->Draw();
```

などとする。ここで、h2 は問題の 2 次元プロットへのポインターである。

Y 軸への射影は

```
h2->ProjectionY()->Draw();
```

である。

後で使い回すことを考えると、

```
TH1D *h1 = h2->ProjectionX();
```

などとした後で

```
h1->Draw();
```

の方が良いかも知れない。

名前を知っていれば、射影の名前は、もとの 2 次元プロットの名前に "_px" または "_py" を付け加えたものになる。

具体的な例題をやってみよう。まず、例によって、プロットを描画する窓を開いておこう。

```
TCanvas *c1 = new TCanvas("c1","My Canvas",10,10,400,400);
c1->Divide(2,2);
```

開いた窓は、2 × 2 に分割した。

次に、最初の区画に行ってから、練習に使う 2 次元プロットを作ろう。

```
c1->cd(1);
TH2F *h2 = new TH2F("htest","Test",50,0.,10.,60,-10.,20.);
```

練習だから、また適当な乱数を使うことにする。

```
{
  gRandom->SetSeed();
  Int_t i;
  for (i=0; i<10000; i++) {
    Double_t x = 10. * gRandom->Rndm();
    Double_t y = 0.05*x*x + gRandom->Gaus(0.,0.2*x + 1.);
    h2->Fill(x,y);
  }
}
h2->Draw();
```

for 文が複数行にまたがるので、"{" と "}" で囲んである。

これを Y 軸と X 軸に射影してみよう。

```
c1->cd(2);
h2->ProjectionY()->Draw();
c1->cd(3);
h2->ProjectionX()->Draw();
```

すると、既に述べたようにそれぞれの射影のヒストグラムの名前は、もとの名前に "_px" あるいは "_py" をつけたものになる。今の例では、"htest_px" と "htest_py" である。

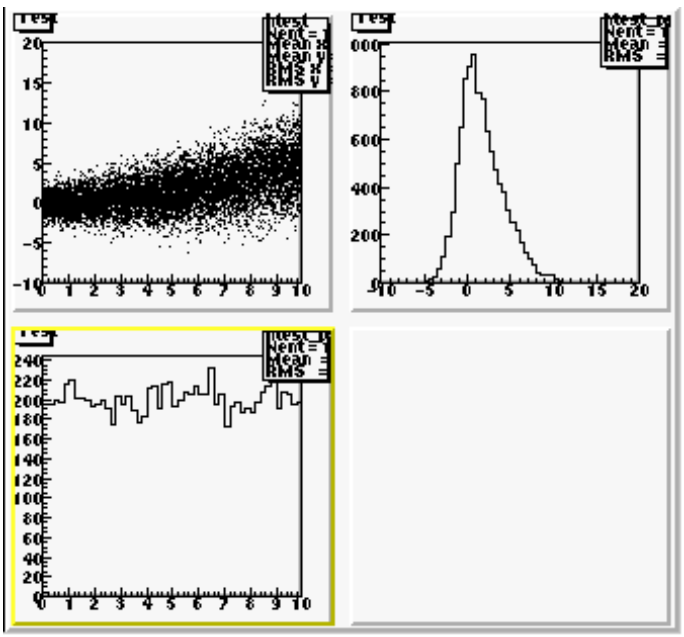

図 3.9: 2 次元プロットの両軸への射影。

3.3.3 PROFILE

上で作った 2 次元プロットの例題にプロファイリングを施してみよう。

```
TProfile *h2_pfx    = h2->ProfileX();          // プロファイルの作成
TH1D     *h2_pfx_px = h2_pfx->ProjectionX();   // TH1D へコンバート
h2_pfx_px->Draw();                             // 描画
```

結果は図 3.10 のような感じで、もとの 2 次元プロットの芯の部分の形、$y = 0.05 * x * x$ が見える。ここでは、真

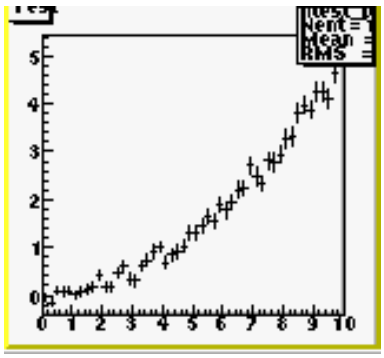

図 3.10: 図 3.9 の 2 次元プロットのプロファイリング。X-Y の相関の関数形が見える。

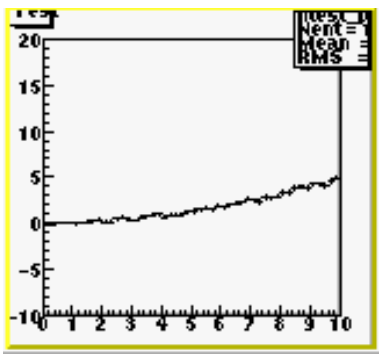

図 3.11: 図 3.10 の Y 軸の上限と下限を図 3.9 の 2 次元プロットに合わせる。

面目にポインターを使ったが、プロファイルのヒストグラムの名前はもとの名前に "_pfx" (ProfileY() をした場合には "_pfy") をつけたものになるので、今の場合、"htest_pfx" でもアクセスできる。

3.3.4 上限下限 (MAMI)

ちょっと、見にくいので、もとの 2 次元プロットの縦軸のスケールに合わせてみよう。

```
h2_pfx_px->SetMinimum(-10.);          // 下限を -10. に
h2_pfx_px->SetMaximum(20.);           // 上限を 20. に
```

これで、より相関関係が見やすくなった。

3.3.5 スライス (BANX または BANY)

X 軸のビン first から、ビン last までをスライスして Y 軸に射影 (BANX) するには

```
TH1D *hbx = h2->ProjectionY("htest_banx",first,last);
```

とする。Y 軸に沿ってスライスし、X 軸方向へ射影 (BANY) するのであれば

```
TH1D *hby = h2->ProjectionX("htest_bany",first,last);
```

などとすれば良い。

first、last はビンの番号なので、今の例で X 軸の $(1.,2.)$ をスライスし、それを Y 軸に射影しガウスフィットするには

```
Int_t first = h2->GetXaxis()->FindBin(1.);            // 座標値 1. をビン番号へ
Int_t last  = h2->GetXaxis()->FindBin(2.);            // 座標値 2. をビン番号へ
TH1D *hbx = h2->ProjectionY("htest_banx",first,last); // スライス
hbx->Draw();                                          // 描画
hbx->Fit("gaus");                                     // スライスをガウスフィット
```

などとする。

3.3.6 AVX または AVY

著者が dis45 で一番お世話になったのがこれである。AVX と同じことを ROOT でするには

```
h2->FitSlicesY();    // avx に対応
htest_0->Draw();     // peak 値
htest_1->Draw();     // <y>(x)
htest_2->Draw();     // sigma_y(x)
htest_chi2->Draw();  // chi2(x)
```

などとする。ここで、h2 の名前が "htest" であったことを思い出すこと。正しくは、

```
((TH1D *)gROOT->FindObject("htest_0"))->Draw();     // peak 値
((TH1D *)gROOT->FindObject("htest_1"))->Draw();     // <y>(x)
((TH1D *)gROOT->FindObject("htest_2"))->Draw();     // sigma_y(x)
((TH1D *)gROOT->FindObject("htest_chi2"))->Draw();  // chi2(x)
```

とすべきところであるが、ついに悪い習慣に染まってきた。

AVY なら FitSlicesX() を使う。実は FitSlicesX() または FitSlicesY() は gauss だけでなく、任意関数を使うことができるが、それに関しては各自研究されたい。

図 3.12 は

```
h2->FitSlicesY();    // avx に対応
c1->cd(1);
```

```
h2->Draw();
c1->cd(2);
htest_0->Draw();     // peak 値
c1->cd(3);
htest_1->Draw();     // <y>(x)
c1->cd(4);
htest_2->Draw();     // sigma_y(x)
```

とした結果である。

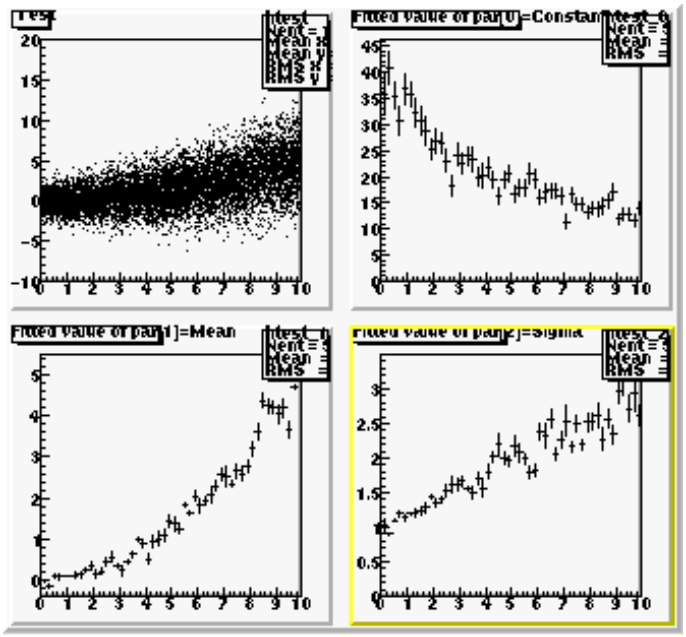

図 3.12: 2 次元プロットの Y スライスのガウスフィット、つまり、dis45 で言うところの AVX。

3.3.7 ビン幅の変更 (CHBN)

1 次元ヒストグラムの場合、

```
h1->Rebin(5);
```

で、5 ビンずつまとめる。これは、非可逆的な変更なので、変更前のヒストグラムをとっておきたければ、

```
TH1D *h1_chbn = h1->Rebin(5,"hnew");
```

などと、新しいヒストグラムの名前 ("hnew") を与えてやれば良い。

2 次元ヒストグラムの場合はメンバー関数がないのでマクロで誤魔化す。

```
.L rebin.C;
TH2D *h2_chbn = Rebin(h2,2,3);
h2_chbn->Draw();
```

ダサダサだが仕方ない。rebin.C の中身は付録に載せておく。

注意事項として、Rebin(...) は、ProfileX() とか、FitSlicesY(...) の結果に対しては使えないことに注意する。ビンがマージされた場合、ビンの中身の平均でなく和が、新しい中身になるからである。あえて使うなら、あとでビン幅のスケール因子で割っておかねばならない。間違いのもとなので、Rebin(...) してから ProfileX() とか FitSlicesY() とかすること。

3.3.8 部分の拡大 (BLOW)

$(-5., 5.)$ の範囲を拡大するには

```
h1->GetXaxis()->SetRange(h1->GetXaxis()->FindBin(-5.),
                         h1->GetXaxis()->FindBin(5.));
```

とする。これはまた、軸の値から対応するビン番号をうる方法を示している。

直接、値を入れられないのがうっとおしい、これをいちいち打つのが面倒ちいという人は、またマクロで誤魔化せば良い。

```
.L blow.C;
Blow(h1,-2.,5.5);
```

これは、可逆的な変更である。blow.C の中身は付録に載せておく。

3.3.9 軸を log にしたい

軸をリニアースケールで表示するか、log スケールにするかは、表示だけの問題であるので、プロットの表示されるキャンバスの区画 (TPad オブジェクト) の属性である。そこで、スケールの変更は TPad のメンバー関数を使って行なう。i 番目の区画にあるプロットの Y 軸を log スケールに変更するには

```
c1->cd(i);                              // i 番目の区画に移動
gROOT->GetSelectedPad()->SetLogy();     // Y 軸を log に
```

とすれば良い。もとに戻すには

```
gROOT->GetSelectedPad()->SetLogy(0);    // Y 軸をリニアーに
```

とする。しかし、こんな面倒なことを手で打つ必要はない。問題の TPad のすみのあたり (プロットの中ではダメ) を、マウスで右クリックすればコンテクストメニューが現れ、グリッドのあるなし、log なのかリニアーなのか、ティックマークの調節などなど、そこで見栄えに関してやりたいことのほとんどができるのだ。

3.3.10 プロット間の演算 (OPER)

対話的にプロットをいじっていて一番オブジェクト指向ぽいのは多分これであろう。

プロットとプロットの四則演算は (もちろん、ビン幅が共通していて初めて許されるわけだが) 、単純に +、=、*、/ でよい。

```
TH1D *haplusb  = new TH1D(h1a + h1b);
haplusb->Draw();
TH1D *haminusb = new TH1D(h1a - h1b);
haminusb->Draw();
TH1D *hatimesb = new TH1D(h1a * h1b);
hatimesb->Draw();
TH1D *hadivbyb = new TH1D(h1a / h1b);
hadivbyb->Draw();
```

と言った具合だ。

また、1 次元ヒストグラム h1 のビンの中身を scale 倍するには、

```
TH1D *hscaled = new TH1D(scale*h1);
hscaled->Draw();
```

でよい。

3.3.11 重ね書き (TC)

既に述べたとおり、

```
h->Draw("same");
```

異なったヒストグラムを重ねて比較するだけでなく、ヒストグラムに、誤差棒をつけるなどしたい時もこれを使う。

3.3.12 統計情報の表示 (SET STAT ...)

フレーム内に表示されるプロットの統計情報の制御は

```
gStyle->SetOptStat(mode)
```

ここで mode = iourmen は 2 進数で、

```
n = 1;  ヒストグラムの名前の表示
e = 1;  エントリー数の表示 ( 重みゼロも含む )
m = 1;  平均値の表示
r = 1;  rms の表示
u = 1;  アンダーフロー数の表示
o = 1;  オーバーフロー数の表示
i = 1;  ビンの積分値の表示
```

である。デフォールトは mode = 0001111 であるが、しばしば見えている部分の積分が知りたくなるので、著者の場合、

```
gStyle->SetOptStat(1001111);
```

を愛用している。

3.4 プロットの情報を得る

ヒストグラムに蓄えられた情報をもとに何か計算したいというのはしばしば起こる欲求である。そんな時のための技を紹介する。その際、まず覚えておかなくてはならないことはビン番号の数え方の習慣である。

```
TH1D *h1 = new TH1D("name","title",nbins,xmin,xmax);
```

として作ったヒストグラムの場合、

```
0             : underflow bin
1 - nbins     : ordinary bins
nbins+1       : overflow bin
```

となる。

既に存在しているヒストグラムのビンの数や、上下限を知るには、

```
nxbins = h1->GetXaxis()->GetNbins();
xmin   = h1->GetXaxis()->GetXmin();
xmax   = h1->GetXaxis()->GetXmax();
```

また、ビンの中心値や、ビンのエントリーの数を知るには

```
xc     = h1->GetXaxis()->GetBinCenter(i); // center of i-th bin
z      = h1->GetBinContent(i)             // content of i-th bin
```

が役に立つ。

2 次元プロットの場合、Y 軸のビンの数や、上下限を知るには、容易に想像できるように、

```
nybins = h2->GetYaxis()->GetNbins();
ymin   = h2->GetYaxis()->GetXmin();
ymax   = h2->GetYaxis()->GetXmax();
```

とすれば良い。Y 軸であっても GetXmin() や GetXmax() である点に注意。X 軸については 1 次元の場合と同じ。

また、ビンの中心値や、ビンのエントリーの数を知るには

```
xc     = h2->GetXaxis()->GetBinCenter(i);     // x center of bin (i,j)
yc     = h2->GetYaxis()->GetBinCenter(j);     // y center of bin (i,j)
z      = h2->GetBinContent(h2->GetBin(i,j));  // content  of bin (i,j)
```

などとする。GetBin(i, j) で 2 次元のビンを 1 次元に翻訳してくれる。逆に (x, y) から (i, j) を知るには、Blow でやったように、

```
Int_t i = h2->GetXaxis()->FindBin(x);
Int_t j = h2->GetYaxis()->FindBin(y);
```

を使う。

ビンの積分は

```
sum = h1->Integral(i1,i2);
sum = h2->Integral(i1,i2,j1,j2);
```

てな感じ。

さて、練習として、あるヒストグラムの平方根をとることを考えてみよう。

```
void Sqrt(TH1 *h)
{
   Double_t x, y, dy;

   for (int i=0; i<h->GetNbinsX()+1; i++)
   {
      y  = h->GetBinContent(i);
      dy = h->GetBinError(i);
      if (y > 0.) {
          y  = TMath::Sqrt(y);
          dy = 0.5*dy/y;
      } else {
          y = dy = 0.;
      }
```

```
      h->SetBinContent(i,y);
      h->SetBinError(i,dy);
   }
}
```

のようなマクロを sqrt.C とかいう名前で用意しておけば

```
.L sqrt.C
Sqrt(h1);
```

とすることにより、h1 の中身の平方根がとられる。このマクロではもとのヒストグラムを破壊してしまう。それがいやなら、もとのヒストグラムをコピーしてからするか、新しいヒストグラムのポインターを返すようにマクロを改良すれば良い。

3.5 グラフの作成：重ね書き

ここまでは、ヒストグラムの使い方をみてきたが、データ点を直接キャンバスにプロットしたいという要求もしばしば起きる。このような場合、TGraph あるいは誤差棒つきのプロットの場合は TGraphErrors を使うことになる。基本的な使い方は「猿にも」にあるので重複するが、複数のグラフの重ね書きなど書いてないこともあるので補足しておく。

グラフを作ることの練習なのでデータ点については適当に関数と乱数を組み合わせて作ろう。

```
const Int_t np = 10;   // # data points
Int_t i;
Double_t x[np], dx[np], y1[np], y2[np], dy[np];
{
   for (i=0; i<np; i++) {
      x [i] = 0.5 + i;
      dx[i] = 0.3;
      y1[i] = 4. + 0.1*x[i]*x[i] + gRandom->Gaus(0.,1.);
      y2[i] = 8. + 0.1*x[i]*x[i] + gRandom->Gaus(0.,1.);
      dy[i] = 1.;
   }
}
```

後で 2 つのグラフの重ね書きをしたいので、y は 2 種類作ってある。

これを、ただ単にプロットするだけなら

```
TGraph *gr1 = new TGraph(np,x,y1);
gr1->Draw("ap");
```

あるいは誤差棒つきなら

```
TGraphErrors *gre1 = new TGraphErrors(np,x,y1,dx,dy);
gre1->Draw("ap");
```

で良いわけで、これは「猿にも」にも書いてある。ここで、Draw(option) メソッドの option として、"a"（軸の描画）と "p"（プロット）を 2 つ指定したが、重ね書きがしたければ 2 つめをプロットする時に "p" のみとすればよい。この他よく使う option として、点を直線でつなぐ "l" や、滑らかな曲線でつなぐ "c" があるのはヒストグラムの場合と同じである。

上の例ではデフォールトの軸を使ったが、プロットの範囲を指定したり重ね書きをしようと思うと先にグラフの枠を書いておくのが望ましい。

```
Double_t xlo = 0.;    // x の下限
Double_t xhi = 10.;   // x の上限
Double_t ylo = 0.;    // y の下限
Double_t yhi = 20.;   // y の上限
TCanvas *c1 = new TCanvas("c1","My Canvas",10,10,400,400);
TH1F *frame = gPad->DrawFrame(xlo,ylo,xhi,yhi);
```

こうしておけば

```
TGraphErrors *gre1 = new TGraphErrors(np,x,y1,dx,dy);
gre1->Draw("p");
TGraphErrors *gre2 = new TGraphErrors(np,x,y2,dx,dy);
gre2->Draw("p");
```

とした時に指定通りの軸の範囲で重ね書きできる。

既にヒストグラムの説明の際に述べたようにプロットマーカーや曲線の種類や色、大きさなど、お化粧は GUI でできるが、もちろん

```
gre1->SetLineColor(2);     // 線に赤を指定
gre1->SetMarkerColor(2);   // マーカーに赤を指定
gre1->SetMarkerStyle(8);   // マーカーに○ を指定
gre1->SetMarkerSize(1.3); // マーカーのサイズ指定
gre1->Draw("p");           // 描画
gre2->SetLineColor(4);     // 線に青を指定
gre2->SetMarkerColor(4);   // マーカーに青を指定
gre2->SetMarkerStyle(22); // マーカーに△ を指定
gre2->SetMarkerSize(1.3); // マーカーのサイズ指定
gre2->Draw("p");           // 描画
```

のように直接コマンドラインで指定することもできる。できたプロットは図 3.13。

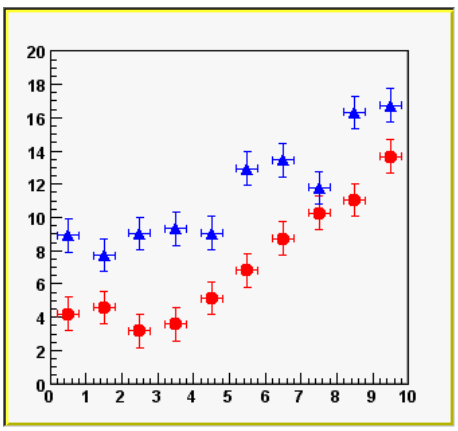

図 3.13: グラフの重ね書き。

3.6 GUI を使ったプロットの扱い

これまで説明したプロットの扱いは、プロットのお化粧に限らず、かなりの部分を GUI で行うことができる。
ROOT のプロンプトで

```
TFile *file = new TFile("file.root","recreate");
```

とうつと、”file.root” と言う名前の出力用 ROOT ファイルができる。これ以降、作成したプロットは特に指定しなければこのファイルに保存される。すでに、プロットを含んだ ROOT ファイルがある場合には

```
TFile *file = new TFile("file.root");
```

とするか、ROOT 起動時に

```
root -l file.root
```

とすれば良い。ここで、

```
TH2F *h2 = new TH2F("htest","Test",50,0.,10.,60,-10.,20.);
{
  gRandom->SetSeed();
  Int_t i;
  for (i=0; i<10000; i++) {
    Double_t x = 10. * gRandom->Rndm();
    Double_t y = 0.05*x*x + gRandom->Gaus(0.,0.2*x + 1.);
    h2->Fill(x,y);
  }
}
```

として、テスト用の 2 次元プロットを作っておく。
準備ができたら

```
TBrowser b;
```

として、ROOT ファイルブラウザーを開く。”ROOT Files” と示されたディレクトリーアイコンをダブルクリックすると file.root と書かれたアイコンがでてくる (図 3.14)。これをダブルクリックすると、上で作った ”htest” というプロットのアイコンが出てくる (図 3.15)。ここで、このアイコンをダブルクリックすれば、自動的にキャ

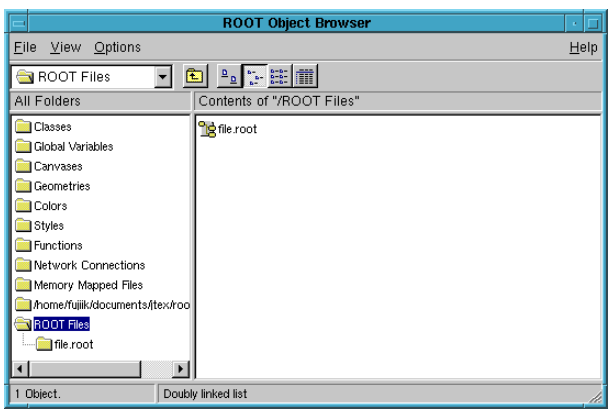

図 3.14: TBrowser を開き ”ROOT Files” をダブルクリックしたところ。

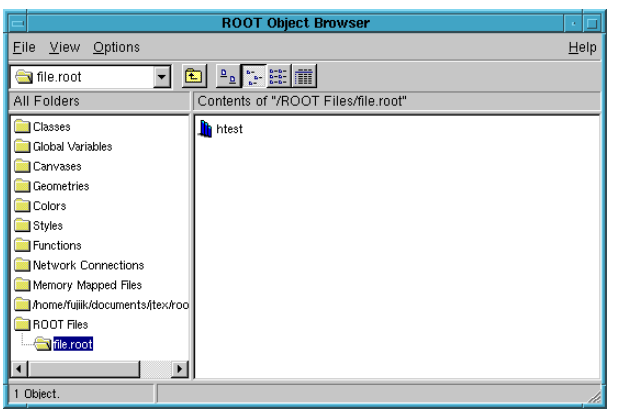

図 3.15: 更に ”file.root” をダブルクリックしたところ。

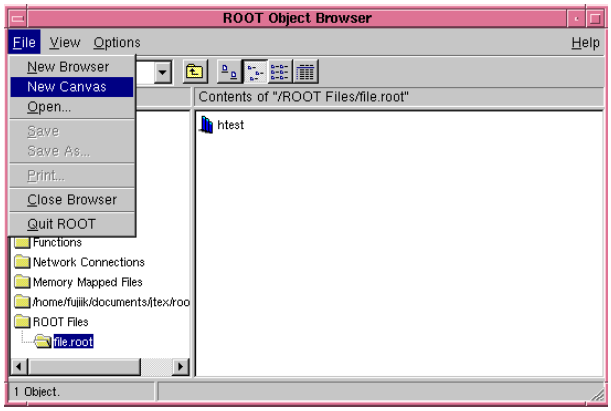

図 3.16: ”File” メニューから ”New Canvas” を選び TCanvas を開く。

ンバスウインドウが開いてプロットが表示される。

そうではなくてあらかじめ分割されたキャンバスの 1 区画に表示するには、TBrowser の ”File” メニューから”New Canvas” という項目を選ぶ (図 3.16)。出てきたキャンバスを右クリックし、出てきたメニューから”Divide” を選ぶ (図 3.17)。すると、ダイアログボックスが現れるので、”nx” と ”ny” に適当な数を入れる。ここでは $(2, 2)$ に分割したいので、”nx”、”ny” ともに 2 を入れて ”OK” を押す (図 3.18)。分割されたキャンバスの第 1 区画に

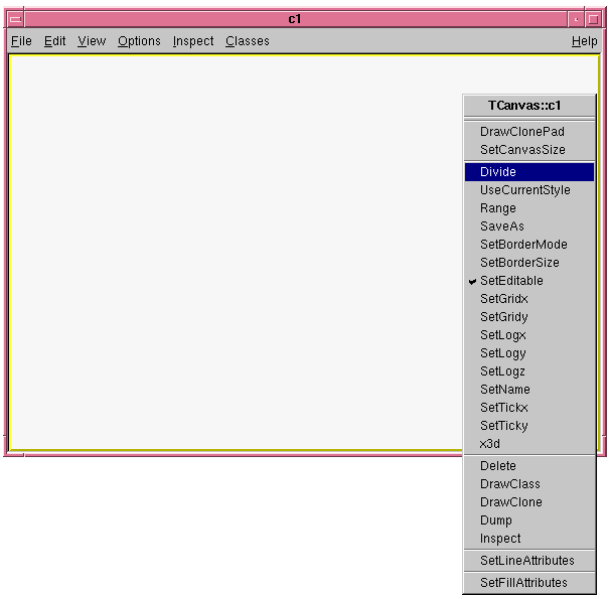

図 3.17: TCanvas を右クリックしてメニューから ”Divide” を選ぶ。

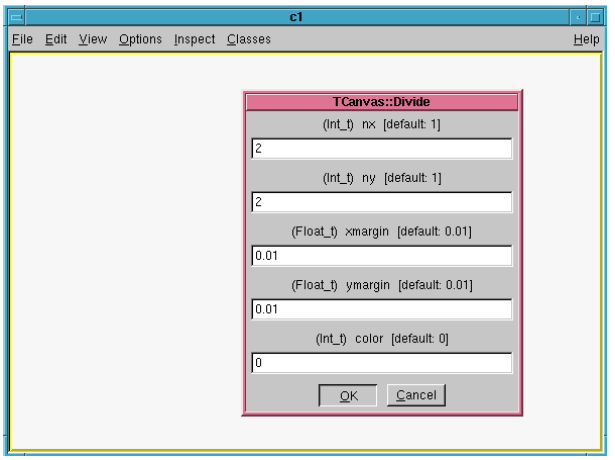

図 3.18: すると、TCanvas :: Divide ダイアログが現れる。

移動するため、マウスの真ん中ボタンで第 1 区画をクリックする。これで第 1 区画の回りの枠の色が変わりフォーカスが第 1 区画に移動したことが分かる。ここでおもむろに先ほどのブラウザーの ”htest” アイコンをダブルクリックすれば、めでたくこの区画にプロットを表示できる。

今度はこれの Y 軸への射影をしてみよう。プロットを右クリックして現れたメニューから”ProjectionY” を選択し、必要に応じて (BANX をしたい場合) ”firstxbin” と ”lastxbin” にビンナンバーを指定する。ここで、”OK” を押せば、”name” を特に指定しなければ”htest_py” というプロットができたはずである。ここで、TBrowser の ”View” メニューから”Refresh” を選択すれば、”htest_py” のアイコンが現れるはずである。これを第 2 区画にプロットしたいので、真ん中ボタンで第 2 区画をクリックして第 2 区画に移動してから、”htest_py” アイコンをダブルクリックして Y 射影を表示する。

3.7 フィッティング

フィットについては、「猿にも」に詳しいので、ほんとに簡単にさらりと。

3.7.1 標準関数によるフィッティング

極めてしばしばやるフィットは

```
h1->Fit("gaus");
h1->Fit("expo");
h1->Fit("pol1");
h1->fit("pol2");
....
```

などである。これらについてはこれ以上の説明を要しないであろう。

これらについても GUI が用意されている。ヒストグラムを触って、ポインターが矢印になったら、右クリックして、コンテクストメニューを出し、FitPanel を選んで研究されたい。

3.7.2 任意関数によるフィッティング

上記のコンテクストメニューで Fit を選べば、任意関数が与えられる。実際に起こることは、コマンドラインで

```
TF1 *fn = new TF1("fn","[0]*exp(-[1]*x)+[2]*x+[3]",0.,5.);
fn->SetParameters(1.,5.,2.,0.);
fn->SetParNames("a","b","c","d");
h1->Fit("fn","r");
```

などとするのと同じである。

3.7.3 フィット情報の表示 (SET FIT ...)

フレーム内に表示されるプロットのフィット情報の制御は

```
gStyle->SetOptFit(mode)
```

ここで mode = pcev は 2 進数で、

- `v = 1;` パラメータの名前と値の表示
- `e = 1;` 誤差の表示 (`e = 1` は `v = 1` を要求)
- `c = 1;` Chi**2/NDF の表示
- `p = 1;` 確率の表示

である。デフォールトは mode = 0111 である。

3.8 Ntuple の扱い

Ntuple の使い方についても、「猿にも」に書いてあるのでさらりと。まず箱を用意する。つまり

```
TNtuple *tup = new TNtuple("tupname", "tuptitle", "adc1:adc2:adc3");
```

などとして、例えば 3 つの変数 "adc1"、"adc2"、 "adc3"、からなる Ntuple を作る。またもや練習なのでダミーデータを詰め込もう。

```
{
  Int_t i;
  for (i=0; i<10000; i++) {
    Float_t mu   = 100. + gRandom->Gaus(0.,30.);
    Float_t adc1 = gRandom->Gaus(mu,5.);
    Float_t adc2 = gRandom->Gaus(mu,5.);
    Float_t adc3 = gRandom->Gaus(mu,5.);
    tup->Fill(adc1,adc2,adc3);
  }
}
```

この Ntuple の内容をヒストグラムしたければ

```
tup->Draw("adc1");
```

あるいは、カットつきなら例えば

```
tup->Draw("adc1","adc2>80. && adc3<150.");
```

等とすればよい。
2 次元プロットなら

```
tup->Draw("adc1:adc2","adc3>80.");
```

これを "hadc12" などという名前の 2 次元プロットにしたかったら

```
tup->Draw("adc1:adc2 >> hadc12","adc3>80.");
```

hadc12 は普通の 2 次元プロットなので自由にいじれる。
また、

```
tup->Draw("adc1-adc2","adc3<150.");
tup->Draw("adc1-adc2:adc1","adc3<150.");
```

等 Ntuple の内容に演算を加えてからヒストグラムすることも可能である。作った Ntuple をファイルに書き出すには

```
TFile file("testtup.root","RECREATE");
tup->Write();
```

でよい。
まあ、ここまでは「猿にも」の受け売りであるが、時には、イベント毎に Ntuple の内容に直接アクセスしたい場合もでてくる。今保存した Ntuple の入った ROOT ファイルを読んでイベント毎の情報を引き出すことを考えよう。ROOT を再起動して、TNtuple を "tupname" という名前で作ったことを思いだし、

```
TFile *file = TFile::Open("testtup.root");
TNtuple *tup = (TNtuple *)gROOT->FindObject("tupname");
```

として保存した Ntuple へのポインターを得る。ここで、例えば、イベント 5 の変数 "adc1" の値が取ってきたかったら、

```
Float_t a1;
tup->SetBranchAddress("adc1",&a1);       // link "adc1" to a1
tup->GetEntry(5);                        // get event 5
cerr << a1 << endl;
```

さて、カットされたサブサンプルに対して情報を得たい場合は TEventList を使う。例えば、カット後の最初のイベントならば

```
tup->Draw(">>elist","adc3 > 80","goff"); // "goff" = graphics OFF
TEventList *elist = (TEventList *)gDirectory->Get("elist");
Int_t nlist = elist->GetN();             // get No. selected events
Int_t event = elist->GetEntry(1);        // event No. for 1st selected
tup->GetEntry(event);                    // get event No.event
cerr << a1 << endl;
```

等とする。

3.9 グローバルのリセット

いい忘れたが、ROOT の対話セッションで打ち込んだシンボルは基本的にグローバルスコープである。それらをリセットするには

```
gROOT->Reset();
```

とする。これで、ヒープに作ったものは別だが、きれいな気持ちで再出発できる。

3.10 ヒストリーファイル

手で長いコマンドを打ち込んでも、RCINT にはそのバックアップの機能はない。そこで、

```
$HOME/.root_hist
```

をもとに、よく使うコマンドシーケンスはマクロにしておくとよい。

3.11 環境設定ファイル

マクロ、および共有ライブラリーの検索パスなどは

```
$HOME/.rootrc
```

で設定する。著者の場合、

```
Unix.*.Root.DynamicPath: .:$(ROOTSYS)/lib:$(JSFROOT)/lib:$(KFLIBROOT)/lib
Unix.*.Root.MacroPath:   .:$(HOME)/macros:$(ROOTSYS)/macros:$(JSFROOT)/macro
```

などとなっている。JSFROOT は JSF の ルートディレクトリー、KFLIBROOT は Physsim のルートディレクトリーだ。

Rebin とか Blow とか、自分のマクロを自動的にロードさせる。毎回、.L で自分のマクロをロードするのも面倒だからだ。上で述べた .rootrc の中で、MacroPath に入っているところに rootalias.C というファイルを置く。

```
$HOME/macros/rootalias.C
```

その中身は

```
#include "blow.C"
#include "rebin.C"
```

のように、自動ロードしたい自家製マクロをインクルードしておく。もちろん、blow.C、rebin.C も MacroPath に入っているディレクトリーのどこかに置いておくこと。

3.12 ROOT 特有のコマンド

セッション中で

```
.?
```

とすると ROOT の対話セッション固有のコマンドについてのヘルプが見られる。
著者がよく使うのはマクロの実行

```
.x hoge.C
```

やマクロのロード

```
.L hoge.C
```

それにシェルエスケープ

```
.! <shell command>
```

などである。コマンドラインの実行結果をリダイレクトするには、例えば、

```
gDirectory->ls(); > my.log
```

とか

```
gDirectory->ls(); >> my.log
```

とする。

```
.class <class name>
```

は、クラスのヘルプとして使える。
RCINT はデバッガーとしても使えるが、著者は使いこなせていない。

第4章　ROOT を使ったアプリケーションの開発

4.1　作法について

4.1.1　アプリケーション

ROOT の独立したアプリケーションを作る場合は、

```
#include "TROOT.h"

TROOT MyTinyApp("Smallest", "The Smallest ROOT Program");

Int_t main() { return 0; }
```

のように、TROOT オブジェクトをグローバルスコープで、あらゆる ROOT オブジェクト (すなわち基底クラスである TObject クラスの派生クラスのオブジェクト) を作る前に作らなくてはならない。また、全ての ROOT オブジェクトが消えてなくなるまで、消えてはならない。これは、TROOT オブジェクトが全ての ROOT オブジェクトの生成消滅をカスタム new とカスタム delete を通して管理するものだからである。TROOT オブジェクトはスタックに作らねばならない。これは、アプリケーションの終了時に確実にそのデストラクターが呼ばれることを保証するためである。TROOT クラスはシングルトンである。どの ROOT アプリケーションにもただ一つ存在し、いつも gROOT ポインターによってアクセスできる。このポインターにはすでに何度もお世話になった。

4.1.2　コンパイルとリンク

コンパイルフラッグ (CXXFLAGS) には、ROOT のヘッダーのパスが含まれること。これには root-config コマンドを使うのが良い。

```
CXXFLAGS = `root-config --clfags`
```

また、リンクフラッグには ROOT のライブラリー

```
ROOTLIBS = `root-config --libs`
```

を加えなくてはいけない。グラフィックを使う場合は

```
ROOTLIBS = `root-config --glibs`
```

とする。

4.1.3　クラスの拡張と Dictionary

全ては **TObject** から

C++ インタープリターを内蔵することと、基底クラスを共有することの強力な点は、オブジェクトの入出力のためのストリームメソッドが自動的に生成できることである。ROOT のクラスを拡張する際にプログラマーがなすべきことは、まずユーザー定義のクラスヘッダーに

```
class MyClass : public TObject {
  ....
  ClassDef(MyClass,1)  // The class title
}
```

などと、書いておくこと。ここで、1 はバージョン ID である。バージョン ID はランタイムタイプ ID に必要であり、クラスのデーターメンバーが変更を受けたら更新する。バージョン ID を 0 にしておくと、オブジェクト I/O が不要と見なされ、ストリーマーメソッドは自動生成されない。

クラスの実装では、

```
ClassImp(MyClass)
```

などと、書いておくこと。

もう一つのポイントは、かならずデフォールトコンストラクター (引数なしで呼べるコンストラクター) を用意しておくこと。この際、デフォールトコンストラクターでは、データーメンバーなるポインターにメモリーを確保してはいけない。ROOT ファイルが読み込まれる際に、まず、デフォールトコンストラクターが呼ばれ、それからデーターが読み込まれるわけだが、その際にそのポインターが上書きされ、もともとそれが指していたオブジェクトは見失われ、メモリーリークを生じる。

次に LinkDef.h なる名前で

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class MyClass;

#endif
```

のようなファイルを用意する。

これは、Makefile の中で

```
MyClassDict.cxx: MyClass.h LinkDef.h
      @echo "Generating dictionary ..."
      rootcint -f MyClassDict.cxx -c MyClass.h LinkDef.h
```

のように使い、Streamer や Showmember などを自動生成してもらう。

後は、MyClass.o と MyClassDict.o を含めて、共有ライブラリーを作れば良い。

自分のクラスを含む共有ライブラリーのロード

上で、libMyClass.so という名前で共有ライブラリーを作ったものとすると、

```
gSystem->Load("<hoge>/libMyClass.so");
```

とすることで、ROOT の対話セッションの中で自分のクラスが使えるようになる。

html の自動生成

ROOT の WEB ページに行くと、ROOT のクラスの html 版のマニュアルがある。

```
http://root.cern.ch/root/html/ClassIndex.html
```

である。クリッカブルで、親クラスやデーターメンバーになっているクラスの構造を知ることができとても重宝する。実は、これらの索引は自動生成されたものである。自分でつくったクラスについても索引があると便利なので (年をとるとすぐに自分で書いたコードも自分が書いたということすら忘れてしまう) 作り方を知っていると幸せになれる。

やりかたは、例えば html.C とかいうマクロを用意して実行するだけである。

```
$ root -b -q html.C
```

html.C の中身については例題をあげるにとどめる。

```
{
  gROOT->Reset();

  gSystem->Load("../lib/libMyClass.so");

  THtml html;
  html.SetOutputDir("./html/");     // 出力先を html にする。

  html.SetSourceDir("./src/");      // ソースのありかを指定。

  // ここから自分のつくったクラスを html オブジェクトに登録していく

  html.MakeClass("MyClass");
  html.MakeClass("MyClass1");
     ......
     ......
     ......
     ......

  // 登録終了

  html.MakeIndex();  // USER_Index.html という索引の生成

  Char_t *cmd="cd ./html; mv USER_Index.html MyClass.html";
  gSystem->Exec(cmd);  // インデックスの名前を変えておく
}
```

これで、html というサブディレクトリーに html 形式の自分のクラスの索引がつくられる。

```
netscape file://`pwd`/html/MyClass.html
```

とすれば幸せになれる。

コーディング上の習慣

ROOT のプログラミングでは

Identifier	Convention	Example
Classes	Begin with T	THashTable
Non-class types	End with _t	Simple_t
Enumeration types	Begin with E	EColorLevel
Data members	Begin with f for field	fViewList
Member functions	Begin with a capital	Draw()
Static variables	Begin with g	gSystem
Static data members	Begin with fg	fgTokenClient
Locals and parameters	Begin with lower case	seed, thePad
Constants	Begin with k	kInitialSize
Template arguments	Begin with A	AType
Cetters and setters	Begin with Get, Set, or Is (boolean)	SetLast(), GetFirst(), IsDone()

とすることが推奨されている。アンダースコアー "_" は #define のマクロ以外では使わない。

また、マシンに依存しない基本型として

```
Char_t
UChar_t
Short_t
UShort_t
Int_t
UInt_t
Long_t
ULong_t
Float_t
Double_t
Bool_t
```

の使用が推奨されている。

4.2 使ってみよう、便利なクラス

とてもじゃないが全てを説明しきれない。これは、ROOT の提供するクラスが膨大なことと、そしてそれを著者が全く使いこなせていないせいである。クラスのマニュアル (html が自動生成できる) を見るか、ソースコードを直接眺めるのがベスト。また、

```
$ROOTSYS/test
```

にある例題は非常に参考になる。

4.2.1 TString と TObjString

TString は、ROOT の提供する、文字列を扱うクラスである。

```
TString s1("hoge");    // "hoge" という内容の TString を作成
TString s2("hoge");    // "hoge" という内容の TString をもう 1 つ作成
cout << s1 << endl;    // "hoge" をプリント
cout << s1[1] << endl; // "o" をプリント
cout << s1+s2 << endl; // "hogehoge" をプリント
s1.ToUpper();          // s1 を大文字に
cout << s1+s2 << endl; // "HOGEhoge" をプリント
s1[1] = 'o';           // "HOGE" の "O" を小文字に
cout << s1 << endl;    // "HoGE" をプリント
```

などは、ごくごく単純な使い方であるが、すでに十分便利であることがわかる。TString には、実に多くの機能があるので各自研究されたい。

TString は、TObject を継承していないので、そのままではオブジェクト I/O にはむかない。Streamer が自動生成されないからである。そのため、TObjString がある。これは、TString を TObject にするためのラッパーである。

```
TObjString os("hogehoge");   // "hogehoge" という内容の TObjString を作成
Char_t *data = os.GetName(); // "hogehoge" へのポインターをもらう
TString s = os.GetString();  // "hogehoge" を TString としてもらう
os.SetString("foobar");      // "hogehoge" を "foobar" に変更
TString &sr  = os.String();  // TString への参照をもらう
```

のように使う。TString への参照をもらっておけば、後は TString でできることがいろいろできる。

4.2.2 コンテナクラス

TObjArray は、TObject へのポインターの配列である。TObject クラスの子孫であれば、何でもつめることができる。要素要素で別のクラスであっても良い。TObjArray 自体、TObject の子孫なので、配列要素になれる。したがって、複雑なネスト構造がつくれる。TObjArray は、TList、THashList、TClonesArray、TBtree、TSortedList などの仲間で、Collection とか Container とか呼ばれている。

普通の配列でなく、TObjArray を使う利点はいろいろある。配列に要素を加えていく (Add メソッドを使う) 際に、配列が小さければ自動的に拡張してくれる。また、Remove メソッドで、要素を削除したり、削除後空いたところを Compress したりできる。配列要素を一括して delete できる (Delete メソッドを使う)。TIter、At( ) または [ ]、など配列要素にアクセスする標準的なインターフェースが提供されている。要素の並べ代えも簡単にできる。

使い方は、

```
TObjString os1("hoge");
TObjString os2("hogehoge");
TObjString os3("owari");
TObjArray  oa(1);                       // 大きさ 1 の配列を作成
oa.Add(os1);                            // 配列に os1 を入れる
oa.Add(os2);                            // 配列に os2 を追加
oa.Add(os3);                            // 配列に os3 を追加
TObjString *sp = (TObjString *)oa[0]; // s1 を取得 ( cast が必要 )
oa.Remove(s2);                          // s2 を削除
TIter next(oa);                         // iterator の作成
while((sp = (TObjString *)next())) {  // cast が必要
```

```
    cout << sp << endl;                // s2 はプリントされない
  }
  cout << oa[1] << endl;               // oa[1] はいない ( 怒られる )
  oa.Compress();                       // 圧縮
  cout << oa[1] << endl;               // "owari" をプリント ( 怒られない )
```

配列の拡張は時間がかかるので、普通は最初に十分な大きさの領域を確保する。また、決まった大きさの要素の配列に対しては、TClonesArray を使うのが経済的である。さらに使い方を学ぶには

```
$ROOTSYS/test/tcollex.cxx
```

が非常に参考になる。

4.3 プログラム中でインタープリターを使う

C++ プログラムをストリングに収め、それをプログラム中からインタープリターで実行させることもできる。Shell でいうところの eval のようなものである。

例えばこんな感じ。

```
Char_t *cmd = "h1->Draw()";
gROOT->ProcessLine(cmd);
```

これは、実行すべきコマンドがプログラム実行時に変わるような場合にとても便利。

4.4 システムコール

システムコマンドをプログラム中で実行したければ Exec メンバー関数を使う。例えば

```
Char_t *cmd="cd ..; ls";
gSystem->Exec(cmd);
```

のような感じ。

4.5 ROOT の拡張の実際

4.5.1 例題：TH1E クラスの実装

例題。疲れてきたのでまたいずれ書くことにする。

第5章　付録

5.1　ダサダサのマクロ

5.1.1　rebin.C

```
TH2D *Rebin(TH2 *h, Int_t n = 1, Int_t m = 1)
{
    Int_t  nxbins = h->GetXaxis()->GetNbins();
    Axis_t xmin   = h->GetXaxis()->GetXmin();
    Axis_t xmax   = h->GetXaxis()->GetXmax();
    Axis_t dx     = (xmax-xmin)/nxbins;
    Int_t  nxbso  = nxbins;
    nxbins /= n;
    dx     *= n;
    xmax = xmin + nxbins*dx;

    Int_t  nybins = h->GetYaxis()->GetNbins();
    Axis_t ymin   = h->GetYaxis()->GetXmin();
    Axis_t ymax   = h->GetYaxis()->GetXmax();
    Axis_t dy     = (ymax-ymin)/nybins;
    Int_t  nybso  = nybins;
    nybins /= m;
    dy     *= m;
    ymax = ymin + nybins*dy;

    Char_t newname[256];
    sprintf(newname,"%s_chbn",h->GetName());
    TH2D *hnew = new TH2D(newname,h->GetTitle(),nxbins,xmin,xmax,
                                                nybins,ymin,ymax);

    Stat_t z;
    Int_t i, j;
    for (i=0; i<=nxbins+1; i++) {
        Int_t ilo, ihi;
        if (i > 0 && i <= nxbins) {
            ilo = n*(i-1) + 1;
            ihi = n*i;
        } else if (i == 0) {
            ilo = ihi = 0;
        } else {
```

```
            ilo = n*nxbins + 1;
            ihi = nxbso;
        }
        for (j=0; j<=nybins+1; j++) {
            Int_t jlo, jhi;
            if (j > 0 && j <= nybins) {
                jlo = m*(j-1) + 1;
                jhi = m*j;
            } else if (j == 0) {
                jlo = jhi = 0;
            } else {
                jlo = m*nybins + 1;
                jhi = nybso;
            }
            z  = h->Integral(ilo,ihi,jlo,jhi);
            if (z > 0) {
                hnew->Fill(hnew->GetXaxis()->GetBinCenter(i),
                           hnew->GetYaxis()->GetBinCenter(j),z);
            }
        }
    }
    return hnew;
}
```

5.1.2 blow.C

```
void Blow(TH1 *h1, Axis_t mn, Axis_t mx, Char_t *axis = "X")
{
        if (axis == "X") {
                h1->GetXaxis()->SetRange(h1->GetXaxis()->FindBin(mn),
                                         h1->GetXaxis()->FindBin(mx));
        } else {
                h1->GetYaxis()->SetRange(h1->GetYaxis()->FindBin(mn),
                                         h1->GetYaxis()->FindBin(mx));
        }
}
```

5.2 ソースからの ROOT のコンパイル

5.2.1 コンパイルの手順

著者が linuxppc 2k でコンパイルした時を例にとる。開発環境は

```
gcc*-2.95.3-2am
glibc-2.1.3-15d
binutils-2.10.0.9-0a
```

```
XFree86-3.3.6-8a
freetype-1.3.1-1a
Mesa-3.2-3a
```

である。

コンパイルの手順は、ROOT 2.25.xx でがらりと変わった。

```
$ tar -zxvf <somewhere>/root_v2.25.02.source.tar.gz
$ tar -zxvf <somewhere>/ttf_1.1.tar.gz
$ su
# mv ttf/fonts /usr/share/fonts/ttf
# pushd /cern/pro/lib
# ln -s libpythia6134.a libPythia.a
# exit
$ cd root
$ export ROOTSYS=‘pwd‘
$ export LD_LIBRARY_PATH=$ROOTSYS/lib
$ export PATH=$ROOTSYS/bin:$PATH
$ patch -p1 -s < <somewhere>/root_v2.25.00-config.patch
$ patch -p1 -s < <somewhere>/root_v2.25.00-g2c.patch
$ patch -p1 -s < <sowewhere>/root_v2.25-x3d.patch
$ ./configure linuxppcegcs
$ make
$ rm histpainter/src/THistPainter.o
$ make OPT='-O1' all-histpainter
$ make
```

ここで、root_v2.25.00config.patch は、Mesa のヘッダーを探すパスの設定で内容は

```
--- 2.25/configure.ORIG Wed Jun 21 01:42:21 2000
+++ 2.25/configure      Mon Jun 26 22:13:38 2000
@@ -251,7 +251,7 @@
 openglincdirs="$OPENGL $OPENGL/include /usr/include /usr/local/include \
                /usr/include/Mesa /usr/local/include/Mesa /usr/Mesa/include \
                /usr/local/Mesa/include /usr/Mesa /usr/local/Mesa /opt/Mesa \
-               /opt/Mesa/include"
+               /opt/Mesa/include /usr/X11R6/include"

 if [ -z "$openglincdir" ]; then
    openglincdir=NO
```

また、root_v2.23x3d.patch は 16bpp の X で取り敢えず x3d が使えるようにするパッチで、内容は

```
--- 2.25/x3d/src/x3d.c.ORIG     Wed May 17 02:00:45 2000
+++ 2.25/x3d/src/x3d.c  Mon Jun 26 22:27:29 2000
@@ -1049,9 +1049,15 @@

 /* An 16 bit TrueColor ? */
```

```
+#if defined(__linux__)
+#define SIXTEEN  15
+
+       if(XMatchVisualInfo(g->dpy, screen, 15, TrueColor, &vInfo)){
+#else
 #define SIXTEEN  16

        if(XMatchVisualInfo(g->dpy, screen, 16, TrueColor, &vInfo)){
+#endif
          g->depth = SIXTEEN;
        }else{
```

g2c パッチについては、普通のプラットフォームでは忘れていいだろう。linuxppc の環境では、最適化の問題あり、HISTPAINTER_HistPainter.cxx は、"−O2" では正常動作しなかった。

また、上記の TrueType フォントの場所の変更にともない、∼/.rootrc を変更する必要がある。

```
.....
#Unix.*.Root.TTFontPath:     $(ROOTSYS)/ttf/fonts
Unix.*.Root.TTFontPath:     /usr/share/fonts/ttf
.....
```

5.2.2 rmkdepend のパスの問題

ROOT は X の makedepend の C++ 版として、rmkdepend というコマンドを提供している。が、標準のものは、探すべきヘッダーのパスにシステム標準パスの一部が含まれておらず、警告を出す。煩わしいので著者の環境では

```
--- 2.25/build/Module.mk.ORIG   Wed May 17 02:00:52 2000
+++ 2.25/build/Module.mk        Mon Jul  3 15:39:53 2000
@@ -17,7 +17,14 @@
 #RMKDEPCFLAGS := -DINCLUDEDIR=\"/usr/include\" -DOBJSUFFIX=\".obj\"
 RMKDEPCFLAGS := -DINCLUDEDIR=\"/usr/include\" -DOBJSUFFIX=\".o\"
 else
-RMKDEPCFLAGS := -DINCLUDEDIR=\"/usr/include\" -DOBJSUFFIX=\".o\"
+GCCLIBDIR    := $(shell gcc -v 2>&1 | grep '/usr' | sed -e 's;.*\(/usr/lib/gcc-lib/.*\)/specs;\1;')
+PREINCDIR    := \"$(GCCLIBDIR)/include\"
+INCLUDEDIR   := \"/usr/include\"
+POSTINCDIR   := \"$(shell strings $(GCCLIBDIR)/cpp | grep include/g++)\"
+RMKDEPCFLAGS := -DPREINCDIR=$(PREINCDIR) \
+                -DINCLUDEDIR=$(INCLUDEDIR) \
+                -DPOSTINCDIR=$(POSTINCDIR) \
+                -DOBJSUFFIX=\".o\"
 endif

 ##### bindexplib #####
```

のようなパッチをあてて、rmkdepend をコンパイルし直して使っている。

関連図書

[1] "ROOT, Which Even Monkeys Can Use", Y.Shirasaki and O.Tajima, (1999)

[2] "Babar C++ Course", P.F.Kunz

[3] "ROOT Course", F.Rademakers