高エネルギー宇宙物理学のための ROOT 入門
ROOT for High-Energy Astrophysics

奥村 曉
名古屋大学太陽地球環境研究所
oxon@mac.com

2012 年 5 月 9 日

目次

1　はじめに

1.1　ROOT とは

ROOT とは、CERN（欧州原子核研究機構）によって開発されているソフトウェア・ライブラリ群の名称です。高エネルギー物理学のデータ処理・データ解析を主目的として、1994 年から René Brun と Fons Rademakers によって開発が始まりました。2011 年現在、高エネルギー物理学分野での標準解析ツールとしての地位を確立しています。また近年では、宇宙線や宇宙放射線（X 線、ガンマ線）といった他分野でも、データ解析に使われるようになってきました。物理の観測対象によらず、「イベント」の概念を持つ物理学実験においては、ROOT は不可欠な存在になっています。

1.2　ROOT と C++

ROOT は C++ というプログラミング言語で記述されています。1980 年代頃まで科学計算の主流は FORTRAN でした。例えば、PAW、Display 45、GEANT3 といった高エネルギー物理学で使われてきたソフトウェアは FORTRAN で書かれていました。1990 年代に入り、より大規模で柔軟性の高いソフトウェアを構築する必要に迫られたとき、開発者が選択した言語は C++ でした。C++ は C 言語を基にしたオブジェクト指向プログラミング（object oriented programming）が可能な言語であり、高エネルギー物理学業界での標準的な言語になっています。

ROOT を深く理解し正しく使用するためには、どうしても C++ を学習する必要があります。特に ROOT が標準で備える機能に飽き足らず、自分で機能拡張を目指すようなユーザは、C++ の知識が不可欠です。

本格的な C++ の解説は専門書に譲ることにして、本書でも C++ の簡単な説明を第 3 章でします。C++ の基礎知識がある人は、読み飛ばして構いませんが、C++ やプログラミングの初心者には、取っ掛かりとして役立つでしょう。

1.3　高エネルギー宇宙物理学での ROOT の利用例

To be written…

1.4　本書の目的

本書の第一の目的は、ROOT を使ったデータ解析を行えるようになることです。ROOT の機能を全て網羅することはできませんので、説明できるのは基本的かつ必須な機能に絞られてしまいます。しかし、ひとたび ROOT の基本さえ理解すれば、発展的な内容は自分で学習を進めることが可能なはずです。

第二の目的は、ROOT の学習を足がかりにして C++ や Python といったプログラミング言語に慣れ親しむことです。自分でプログラミングができるようになれば、より複雑なデータ解析や、ソフトウェアの構築を自ら行えるようになるでしょう。

これらの目的に合致した適切な文書は、ROOT の公式マニュアル以外にはあまり見当たりません。しかし初心者にとって、ROOT のマニュアルを最初の入門書として読み進めることが適切だとは言えません。インターネット上にも

ROOT に関する情報は多く存在しますが、断片的な情報が多く、体系的にまとめられたものはやはり無いようです。研究室に ROOT の熟練者がいれば、後輩達に指導を行うこともできるでしょうが、同じ説明を毎年全国各地で繰り返すのも時間の無駄です。本書では、共通に使用できる ROOT の入門書として使われることを目指しています。

筆者は ROOT や C++ の多くを独学で勉強しました。当時の自分を振り返り、「こんな入門書があれば分かりやすかったのに」という思いをもとに本書を執筆しました。筆者の独学による狭い視点のため、本書の説明が我流であったり間違っていたりする可能性が多々あります。もしお気づきの点がありましたら、ぜひご指摘いただければ幸いです。また追加してほしい話題や説明がありましたら、ご要望を頂ければ出来る限り対応します。

また本書を作成する際の LaTeX ファイルや例題として登場する C++ のコードなどは、全て http://svn.code.sf.net/p/rheap/code/trunk/RHEAP/src/ から入手可能です。もし必要であれば参照して下さい。また本書の最新版は http://svn.code.sf.net/p/rheap/code/trunk/RHEAP/RHEAP.pdf から入手可能です。

1.5 ROOT の学習方法

物理の勉強方法は個々人で様々なように、ROOT の学習方法にも決まったやり方は存在しません。しかし、初心者の場合にはある程度の指針があったほうが学習しやすいでしょう。以下に、いくつかの学習方法を紹介します。

1.5.1 マニュアル・入門書を読む

ROOT の最も体系的に書かれた教科書は、公式のマニュアル “ROOT Users Guide” です。http://root.cern.ch/drupal/content/users-guide から PDF が入手可能です。500 ページある長いものであり説明も多岐にわたるため、「どこから ROOT を始めたらいいのだろう」という初心者に読みこなすのは大変です。しかし、世の中に存在する ROOT の説明書で最も詳しいものですので、英語の勉強がてら読み進めると良いでしょう。

もっと簡単なものが欲しいという向きには、日本語の以下のものが有名です。

- 『ROOT Which Even Monkey Can Use』Shirasaki and Tajima
- 『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

両者ともインターネット上で検索すれば PDF や PS ファイルが見つかるはずですので、必要があれば検索して下さい。もちろん、本書はこれらの入門書を越えることを目指して執筆されています。

1.5.2 実際に書いてみる

マニュアルや入門書を片手に、実際に自分で例題や解析プログラムを書いて実行しましょう。最初は人の真似をしながら、そして、自分の解析目的に応じて、より複雑なプログラムを書いてみましょう。最初は沢山のエラーや予期しない動作に悩まされるはずです。しかし、そのような苦労をして、最終的に ROOT や C++ を習得するしかありません。失敗は成功への第一歩です。沢山間違えましょう。

1.5.3 ソースコードを読む

マニュアルや入門書に書かれていない機能を使いこなすのは、なかなか難しいものです。自分の作成したヒストグラムを使って複雑な処理をしたい場合、実はその機能は既に ROOT が持っている機能かもしれません。そのような場合は、ROOT のソースコードを読んでみましょう。既に実装された機能が見つかるかもしれません。

ROOT が内部で実際にどのような処理をしているかを知りたい場合も、ソースコードを絶対に読まなくてはいけません。例えば、1 次元ヒストグラムの平均値を ROOT から取り出したとします。このとき、平均値を ROOT はどのよう

に計算しているのでしょうか。平均値はビン幅に影響されるのでしょうか、されないのでしょうか。データ解析をするということは、ROOT をブラックボックスとして使用するのではなく、このような内部の処理をユーザの責任で理解していなくてはいけません。

ソースコードを読むには、ROOT のレファレンスガイドを http://root.cern.ch/root/html/ClassIndex.html から参照するとよいでしょう。先ほどの平均値の問いに答えるためには、http://root.cern.ch/root/html/TH1.html から 1 次元ヒストグラムのクラスである TH1 の説明に行きましょう。そこからどんどんリンクを辿って行けば、答えが見つかるはずです。

1.5.4 インターネットを利用する

インターネットが発達し、欲しい情報は簡単に手に入るようになりました。特にコンピュータやプログラミングに関する知識は、書籍の情報量を圧倒します。その分、情報が断片的であったり、情報の質がバラバラだったりします。初心者の悩みを解決する情報は、大抵インターネット上に転がっています。エラーが出た場合にはそのエラーメッセージで検索しましょう。また、単純に「ROOT」と検索すると、管理者ユーザの意味での「root」が大量にヒットしてしまいます。「CERN」という単語を混ぜる等して工夫して下さい。

また、「〜〜するには、どのようにしたらよいか」のような疑問は、「how to」のような語をつけて検索するとよいでしょう。例えばヒストグラム同士の割り算の仕方を調べたければ、「how to divide histogram root cern」などと検索すれば答えが見つかるはずです。

調べても分からないこと、マニュアルやソースコードの説明が不十分な場合には、ROOT のメーリングリスト（http://root.cern.ch/drupal/content/roottalk-digest）や掲示板（http://root.cern.ch/phpBB2/）を利用することもできます。これらに自ら投稿しなくても、眺めたり検索するだけで有用な情報を得ることができるはずですので、是非活用して下さい。質問する場合は、http://root.cern.ch/drupal/content/roottalk-rules を熟読しましょう。

1.5.5 C++ の勉強をする

既に書きましたが、ROOT と C++ は切り離せない関係にあります。ROOT を深く学ぶためには、C++ を知らなくてはいけません。あなたが疑問に思った点は、実は ROOT に関することではなく C++ 自体かもしれません。研究室や図書館にあるもので構いませんので、適宜 C++ の本を参照するようにしましょう。C++ を理解できれば、ROOT を学ぶ速度は劇的に改善するはずです。

2　ROOT を始めよう

2.1　ROOT のインストールと初期設定

2.1.1　必要な環境

本書を読み進めるためには、当然ながら ROOT を実行するためのコンピュータ環境が必要です。ROOT がサポートする OS は多岐にわたりますが、2011 年現在、Mac OS X か Linux を使用するのが、この業界では標準です。そのため本書では OS X か Linux の使用を前提とします[*1]。どちらかの OS を使う限り、作業内容はほぼ同じはずです。本書に書かれている ROOT での作業を全て行うためには、自分のコンピュータに GCC がインストールされている必要があります[*2]

ある程度のコマンドラインの使用経験を前提としていますので、`ls` や `cd` のような単純なコマンド、「スーパーユーザー」、「管理者」、「ディレクトリ」といった用語の説明は省略します。この章の記述は、コマンドライン操作の初心者には不親切かもしれません。しかし、この章を読み進めることができなければ、恐らく ROOT を使いこなせるようにもなりません。コンパイルができなかった、ROOT がうまく起動しなかった、この章に書いてあることが分からなかったという場合は、自力でインターネットや書籍、研究室の同僚を頼りに解決して下さい。

2.1.2　ダウンロード

ROOT は `http://root.cern.ch/` からダウンロードすることができます。既にコンパイルされているバイナリ版と、コンパイルされていないソースコードがあります。前者はコンパイルする必要がないので簡単ですが、本書では教育目的のため、ソースコードをダウンロードして、自分でコンパイルすることにします。

2009 年 10 月時点で、ROOT の最新バージョン 5.25/02 です。5.24 や 5.22 のような偶数番号のものは pro バージョンと呼ばれ、安定版の扱いです。5.25 のような奇数番号のものは dev バージョンと呼ばれ、新機能の取り込み等が行われているため、不安定な可能性があります。経験的には、pro でも dev でも、色々とバグが潜んでいるので大差ありません。ここでは、5.24/00 をダウンロードすることにします。

上記 URL を辿ると、ダウンロードのページがあります。リンクをクリックしてダウンロードすることも可能ですが、ここでは直接ターミナルからダウンロードします。

```
$ cd ~
$ curl -O ftp://root.cern.ch/root/root_v5.24.00.source.tar.gz
```

などとして、好きなディレクトリにダウンロードして下さい。今の例では、root_v5.24.00.source.tar.gz がホームディレクトリに保存されます。

[*1] Windows や Solaris でも ROOT は動作します。

[*2] GCC についての説明や、そのインストール方法は本書の範囲内ではありません。GCC のインストール方法が分からなければ、「GCC インストール OS X」などで検索して下さい。

2.1.3 コンパイル

ROOT のコンパイル作業やインストール先は、好きな場所でやって構いません。もしあなたがコンピュータの管理者権限を持っているならば、/usr/local の下にインストールするのが標準的です。次のように、ターミナルから好きなディレクトリに移動し、ソースコードを展開して下さい。[*3]

```
$ cd /usr/local
$ sudo tar zxvf ~/root_v5.24.00.source.tar.gz
$ cd root
$ sudo ./configure
$ sudo make
```

root という名前のディレクトリが作成されるので、そこに移動してコンパイルを行います。一般的なソフトウェアのコンパイルと同様に、configure スクリプトを実行します[*4]。make を実行した後に make install を実行したくなりますが、本書の説明通りに作業を進める場合は、別にやる必要はありません。複数コアの CPU を使える場合には、コアの数に応じて最後の行に-j 4 のような引数をつけると、コンパイルが早くなります[*5]。

ここまででエラーが出なければ成功です。もし不運にもエラーが出てしまった人は、コンパイル済みのバイナリをダウンロードしましょう。以下は、OS X 版をダウンロードした場合の例です。

```
$ cd ~
$ curl -O ftp://root.cern.ch/root/root_v5.24.00.macosx105-i386-gcc-4.0.tar.gz
$ cd /usr/local
$ sudo tar zxvf ~/root_v5.24.00.macosx105-i386-gcc-4.0.tar.gz
```

2.1.4 最低限の初期設定

ソースをコンパイルしたり、バイナリ版をダウンロードしただけでは ROOT は使えるようになりません。ROOT の実行ファイルやライブラリの場所を設定する必要があります。以下の 4 行を~/.bashrc に書き足して下さい。

```
export ROOTSYS=/usr/local/root
export PATH=$ROOTSYS/bin:$PATH
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
export PYTHONPATH=$ROOTSYS/lib:$PYTHONPATH
```

これは、ログインシェルに bash を使っている人の場合です。もし zsh を使っている場合は、~/.zshrc に書いて下さい。書き終わったら、

```
$ source ~/.bashrc
```

としましょう。もし csh や tcsh を使っている場合は、~/.cshrc や~/.tcshrc に以下の 5 行を書き足して下さい。

```
setenv ROOTSYS /usr/local/root
```

[*3] この例のように、先頭に$がついている行は、ターミナルで入力していることを表します。実際にターミナルから入力する内容は、$とその直後の半角スペースを取り除いた内容ですので、注意して下さい。

[*4] OS X 10.5 以前を使用していて、Python から ROOT を使いたい人は、./configure のあとに macosx という引数をつけて下さい。32 bit 用のバイナリが作成されます。OS X 標準で付属の Python は 32 bit のバイナリなので、ROOT も 32 bit にしておく必要があります。OS X 10.6 では 64 bit の Python が標準で付属しているため、macosx を付ける必要はありません。

[*5] Mac OS X で-j 4 をつけると、コンパイルに失敗する場合があります。その場合は、引数なしで make を実行して下さい。

```
setenv PATH ${ROOTSYS}/bin:${PATH}
setenv LD_LIBRARY_PATH ${ROOTSYS}/lib:${LD_LIBRARY_PATH}
setenv PYTHONPATH ${ROOTSYS}/lib:${PYTHONPATH}
rehash
```

そして同様に

```
$ source ~/.cshrc
```

としましょう。source コマンドを実行する必要があるのは、初めて.bashrc や.cshrc の設定を ROOT 用に変更したときだけです。次にターミナルを開くときには自動的に処理されますので、source コマンドを打つ必要はありません。

2.1.5 動作確認

それでは早速、ROOT を起動してみましょう。

```
$ root
```

と打ってみて下さい。図 2.1 が画面に現れるとともに、ターミナルには以下のように出力されます。

```
   ****************************************
   *                                      *
   *        W E L C O M E  to  R O O T    *
   *                                      *
   *   Version   5.24/00      29 June 2009 *
   *                                      *
   *  You are welcome to visit our Web site *
   *          http://root.cern.ch         *
   *                                      *
   ****************************************

ROOT 5.24/00 (trunk@29257, Jun 30 2009, 09:23:51 on macosx)

CINT/ROOT C/C++ Interpreter version 5.17.00, Dec 21, 2008
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
```

もしも root というコマンドが見つからないというエラーが出た場合は、PATH 環境変数の設定が正しく行われていません。またライブラリが見つからないというエラーが出た場合は、LD_LIBRARY_PATH の設定が正しく行われていません。.bashrc や.cshrc に打ち間違いがないか再確認しましょう。

最後の行の

```
root [0]
```

は、ROOT が入力待ちをしている状態です。これをプロンプト（prompt）と呼びます。ここに色々なコマンドを打つことによって ROOT を操作します。ひとまず

```
root [0] .q
```

と入力しましょう（Quit の q です）。これで ROOT が終了します。ここまでで、ROOT の起動と終了ができるようになりました。毎回起動の度に起動画面とバージョン情報が出るのが鬱陶しい場合、引数をつけて

```
$ root -l
```

として起動しましょう。すぐに ROOT のプロンプトが表示されるはずです。毎回引数をつけるのが面倒な場合は、

```
alias root="root -l"
```

を~/.bashrc に、もしくは

```
alias root "root -l"
```

を~/.cshrc に書き足すとよいでしょう。

2.1.6 チュートリアルで遊ぶ

ROOT で作業できる内容は様々です。何ができるかを言葉で説明するよりは、どんな図を作ることが可能か眺めるのが手っ取り早いでしょう。まずは、ROOT のチュートリアルで遊んでみましょう。$ROOTSYS/tutorials には様々な例題が置かれています。このディレクトリに移動すると、少し起動後の出力が変わります。

```
$ cd $ROOTSYS/tutorials
$ root

Welcome to the ROOT tutorials


Type ".x demos.C" to get a toolbar from which to execute the demos

Type ".x demoshelp.C" to see the help window

==> Many tutorials use the file hsimple.root produced by hsimple.C
==> It is recommended to execute hsimple.C before any other script

root [0] .x demos.C
```

最後の行は、demos.C という名前のファイルを実行せよという命令です（eXecute の x）。.q に続いて、基本中の基本コマンドなので覚えましょう。このような複数の ROOT の命令が記述されたファイルを、スクリプト（script）やマクロ（macro）と呼ぶことがあります[*6]。

このコマンドを実行すると、いくつかボタンが表示されるはずです。このうち、“hsimple” と書かれたボタンをクリックしてみましょう。図 2.2 のようなウインドウが表示されるはずです。他にも沢山の例が実行可能ですので、全てのボタンをクリックして遊んでみて下さい。終了のときは、やはり.q と打ちます。

もし、あなたのアカウントが$ROOTSYS/tutorials に書き込み権限を持たない場合、hsimple.root を開けないというエラーが出るでしょう。そのような場合は、書き込み権限を持つユーザで作業をするか、$ROOTSYS/tutorials をディレクトリごと自分のホームディレクトリの下にコピーして、そこで作業をしましょう。

[*6] ROOT は CINT というライブラリによって、C++ のソースファイルをコンパイルすることなく実行することが可能です。そのため、shell、Python、Perl などのスクリプト言語と同様にスクリプトと呼びます。「ROOT ソース」と言う場合には ROOT 本体のソースコードを、「ROOT スクリプト」と言う場合には解析用にユーザが書いたプログラムを指すことが多いと思います。

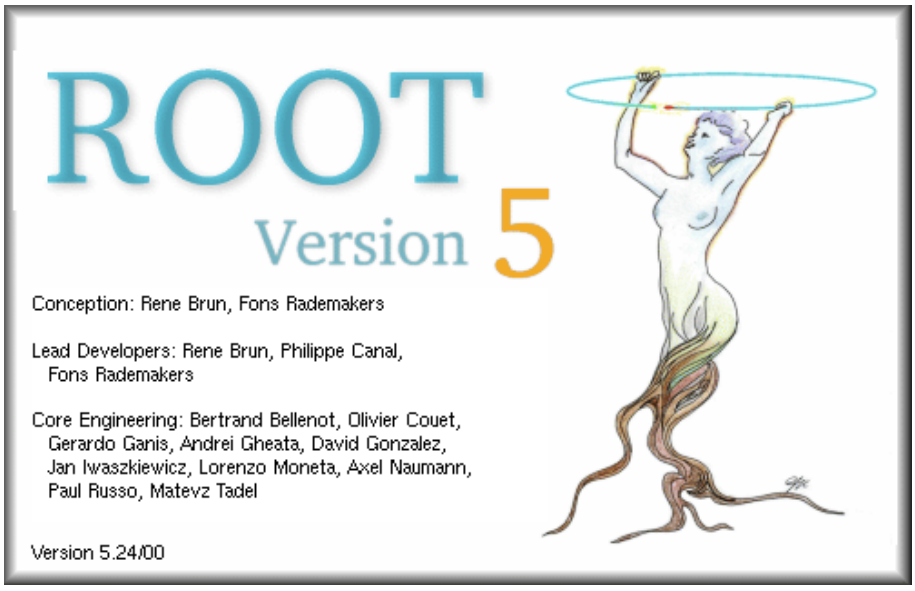

図 2.1 ROOT の起動画面

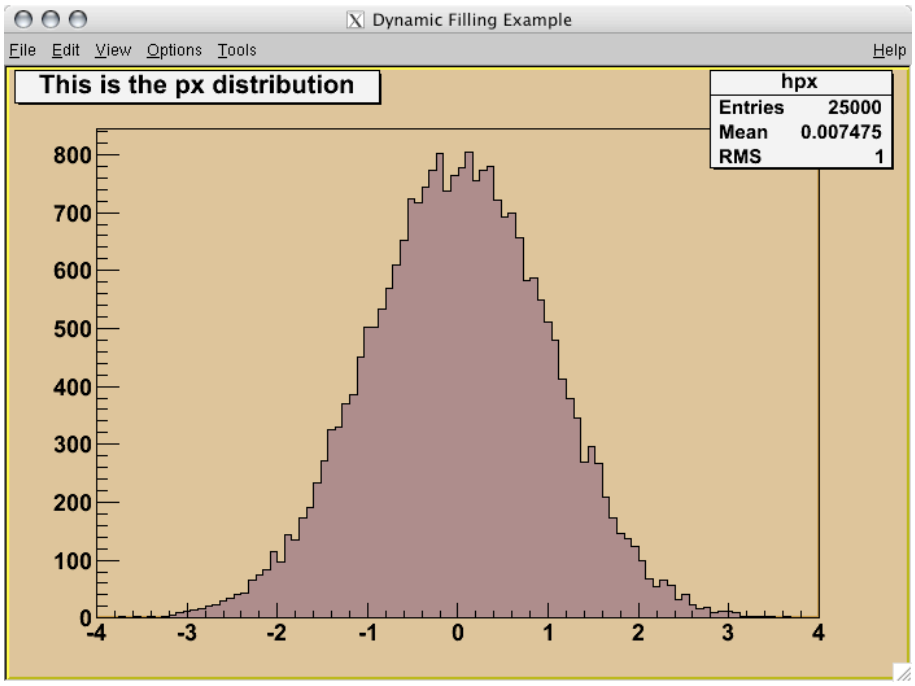

図 2.2 hsimple の実行結果

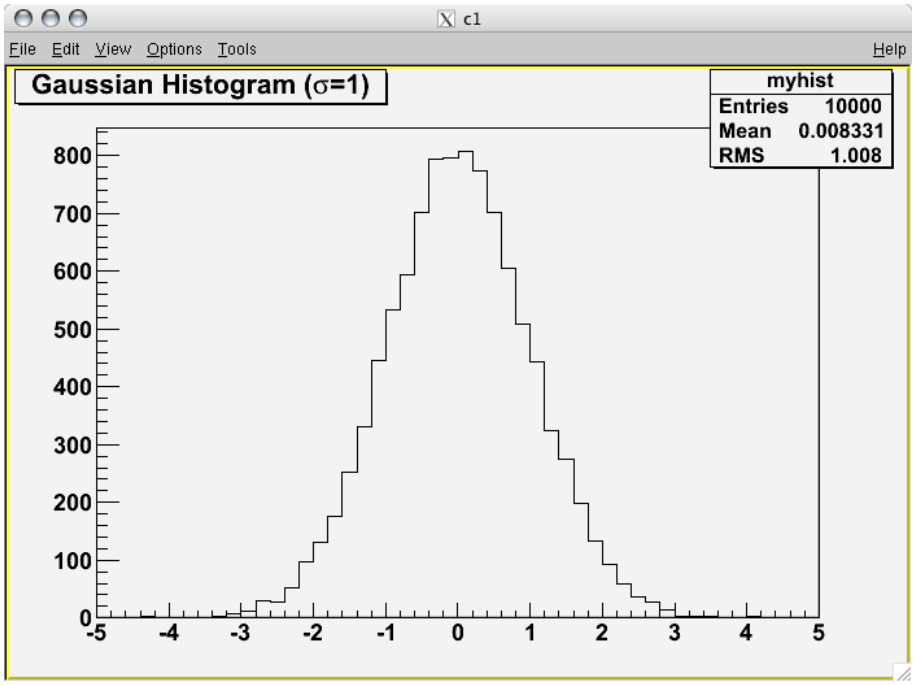

図 2.3 標準偏差 $\sigma = 1$、標本数 $N = 10000$ のヒストグラム

2.2 ROOT の操作に慣れる

まずは、簡単な例を走らせることで ROOT の操作方法に慣れましょう。ここでは、標準偏差 $\sigma = 1$ の正規分布に従った、標本数 $N = 10000$ のヒストグラム（histogram）を作ってみます。

2.2.1 コマンドラインからの操作

まずは ROOT を起動し、

```
root [0] TH1D* hist = new TH1D("myhist", "Gaussian Histogram", 50, -5, 5)
root [1] hist->FillRandom("gaus", 10000)
root [2] hist->Draw()
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

という 3 行を打ってみて下さい。最後の行は ROOT が自動的に出力しますが、エラーではないので今は気にしないで下さい。図 2.3 のような実行結果が画面に現れます。

このままでは何が起きたか分からないので、簡単に説明します。C++ やプログラミングの用語が混ざりますが、各自で調べつつ読んで下さい。これらの 3 行は、実は C++ の言葉で書かれています。まず

```
root [0] TH1D* hist = new TH1D("myhist", "Gaussian Histogram", 50, -5, 5)
```

の部分では、`TH1D` クラス（class）のインスタンス（instance）を作成しています。このインスタンスの名前は、この例では hist としています。`*`や `new` は、おまじないだと思っておいて下さい[*7]。この hist が、ヒストグラムの実体になり

[*7] コンピュータ関連の説明文書の中で、「おまじない」という言葉がたまに出てきます。「それが何者であるか（今は）理解する必要はないが、こうしておくと正しく動作する」ようなものを「おまじない」と呼ぶ習慣があります。とりあえず気にするなということです。

ます。"myhist"や"Gaussian Histogram"という文字列、50、−5、5 という数字が図 2.3 のどの箇所に対応するか、自分で考えて下さい。このヒストグラムは、この時点では作りたてなので、当然中身は空っぽです。そこで

```
root [1] hist->FillRandom("gaus", 10000)
```

として、正規分布に従う数字 10000 個を乱数で発生させ、ヒストグラムに詰めています。最後に

```
root [2] hist->Draw()
```

としてヒストグラムを画面に描画させています。

2.2.2 スクリプトを使った操作

今回の例はたったの 3 行ですので入力は簡単です。しかし、このようなヒストグラムを何百回と繰り返し作成することを想像してみて下さい。実際のデータ解析では、取得した大量のデータに同じ処理をしたい場合があります。そのようなときに、同じコマンドを何度も繰り返し入力するのは非効率的です。そこで、先ほどの 3 行を 1 つのファイルにまとめて書いてしまいましょう。コード 2.1 の内容を入力したテキストファイルを first_script.C というファイル名で好きな場所に保存して下さい[*8]。このようなファイルを、ROOT のスクリプトファイル、マクロファイルなどと呼びます。

first_script.C を保存したディレクトリに移動し、その場所で ROOT を立ち上げ、次の行を実行します。

```
root [0] .x first_script.C
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

先ほどと同様の実行結果（図 2.3）が得られるはずです。ここで、再び.x というコマンドが登場しました。first_script.C を実行せよという意味です。first_script.C に書かれた内容が逐一実行されたため、図 2.3 の結果が得られたわけです。

1 度実行するだけでは、せっかく別ファイルにしたありがたみが分かりません。そこでコード 2.2 のように少し書き直して、first_script2.C というファイルを作成してみて下さい。先ほどと同様に実行してみましょう。ただし今回は

```
root [0] .x first_script2.C(500, 100000)
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

としてみます。少し先ほどの例とは表示されるヒストグラムが変化したはずです。2 つの数字を何通りか試して、何が起きてるか確かめて下さい。何度も実行すると、

コード 2.1 first_script.C

```
1 void first_script()
2 {
3   TH1D* hist = new TH1D("myhist", "Gaussian Histogram (#sigma=1)", 50, -5, 5);
4   hist->FillRandom("gaus", 10000);
5   hist->Draw();
6 }
```

[*8] 日本語の文章の書き方は個人で色々な癖があるように、プログラミング言語の記述にも人それぞれの流儀があります。本書で採用しているコードの流儀は、かなり著者の好みが反映されています。

コード 2.2 first_script2.C

```
1 void first_script2(int nbins, int nevents)
2 {
3   TH1D* hist = new TH1D("myhist", "Gaussian Histogram", nbins, -5, 5);
4   hist->FillRandom("gaus", nevents);
5   hist->Draw();
6 }
```

```
Warning in <TROOT::Append>: Replacing existing TH1: myhist (Potential memory leak).
```

という ROOT の警告（warning）が出るはずですが、今は無視しましょう。

ここまでの作業を振り返ってみます。最初に ROOT のプロンプトから入力した行は 3 行だけでしたが、コード 2.1 と 2.2 では上に 2 行、下に 1 行追加され、合計 6 行になっています。この余計な部分を含めて、関数（function）と呼びます。`void` はおまじないです。`int nbins` や `int nevents` の部分は、引数（argument）と呼ばれるものです。このように特定の機能を関数化することによって、汎用性が高くなります。

ROOT のスクリプトファイルでは、そのファイル名と同一の関数がファイル内に存在すると、その関数が実行されます。したがって、コード 2.2 に書いた関数名を `first_script2` から例えば `foo`[*9]に変更してしまうと、

```
root [0] .x first_script2.C(500, 100000)
Error: Function first_script2(500,1000000) is not defined in current scope  :0:
*** Interpreter error recovered ***
```

というエラーが出てヒストグラムが表示されません。もしファイル名と関数名を別々のものにしたければ、次のやり方も可能です。

```
root [0] .L first_script2.C
root [1] foo(500, 100000)
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

`.L` コマンドで `first_script2.C` をロード（Load の L）します。そのスクリプトファイルの内容を ROOT が呼び出せるようにする作業のことです。`first_script2.C` は既にロードされましたので、その中に書かれていた関数 `foo` に引数を与えて直接呼び出すことができます。また、ひとつのスクリプトの中に複数の関数を記述しても問題ありません。ロードした後に、好きな関数を呼び出して下さい。

2.2.3 図を保存する

図 2.2 や 2.3 の例は、筆者の使用している OS X 上でスクリーンショットを撮ったものです。論文用の図を作成する場合は、ウインドウ上のリサイズボックスやタイトルバーは必要ありません。LaTeX 文書で一般的に使われる、EPS 形式で出力結果を保存してみましょう。`first_script.C` の実行結果が表示されたら、ウインドウ左上にある “File” メニューから “Save As...” を選択し、好きな場所に好きな名前で出力結果を保存しましょう。図 2.4 は、この手順で図 2.3 を EPS 形式で保存し直したものです。この PDF 文書を拡大しても、図が綺麗なことが分かります。学会発表の資料等で使う場合には、PNG や GIF 形式のほうが便利かもしれません。いくつか保存形式が選べますので、試してみ

[*9] `foo` や `bar` というのは、とりあえずの適当な名前として、プログラミングの話題をするときによく出てきます。日本語だと、「ほげ」や `hoge` という単語もよく使われます。

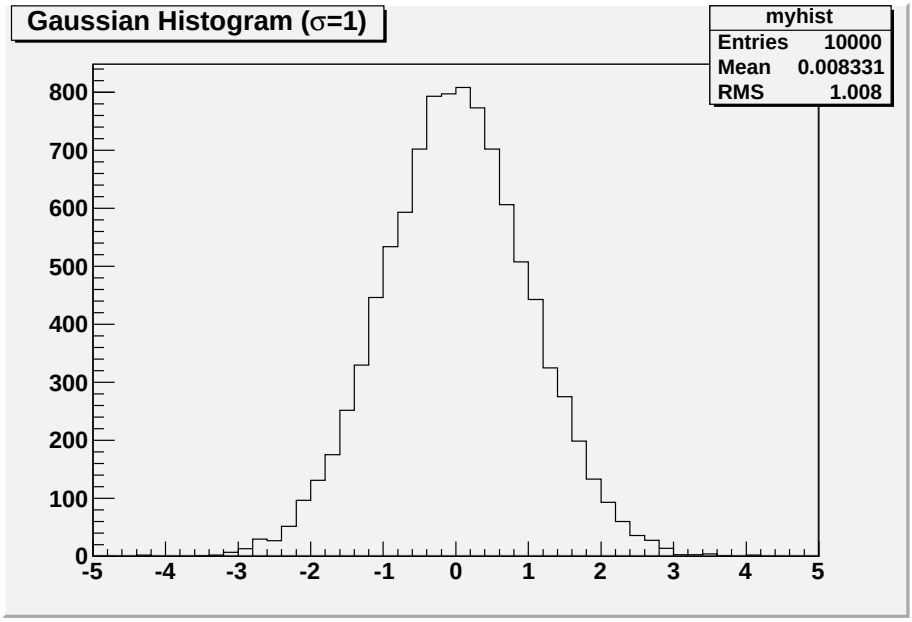

図 2.4 図 2.3 を、EPS 形式で保存し直したもの

ましょう。JPEG 形式は図の細部が潰れますので、お勧めしません。GIF 形式は最大で 256 色までしか使用できないので、そのうち凝った図を作る場合には PNG のほうが良いでしょう。

2.2.4 タブ補完を使う

ROOT のプロンプトでコマンドを打つ場合、キーボードのタブキーを押すことで、補完することができます。携帯電話の予測変換のようなものです。例えば、

```
root [0] TH1D* hist = new TH1D("myhist", "Gaussian Histogram (#sigma=1)", 50, -5, 5)
root [1] hist->FillRandom("gaus", 10000)
root [2] hist->Draw()
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [3] hi
```

とまで入力したところで、タブキーを押してみましょう。自動的に

```
root [3] hist
```

と補完されます。次に、

```
root [3] hist->Get
```

まで入力して再度タブを押してみましょう。次のような候補が大量に現れるはずです。

```
GetArray
GetAt
GetSum
GetSize
```

これは、インスタンス `hist` に操作可能な機能のうち、`Get` で始まるものの一覧です。C++ の言葉で言い換えると、クラス `TH1D` のメンバ関数（member function）のうちゲッター（getter）の一覧です。候補を眺めてみると、`GetMean`

や GetRMS[*10] というメンバ関数が見つかるはずです。何をする関数か、一目瞭然でしょう。それでは

```
root [3] hist->GetMe
```

まで打ち、タブキーを押しましょう。

```
root [3] hist->GetMean
GetMean
GetMeanError
```

の 2 候補にまで絞られます。今は GetMean を試したいので、

```
root [3] hist->GetMean(
```

とまで打って再度タブを打ちましょう。今度は、

```
Double_t GetMean(Int_t axis = 1) const
```

と表示されます。これは、TH1D::GetMean[*11] の引数の説明が表示されたものです。引数には整数（integer）値を取り、そのデフォルト（default）値が 1 だという意味です（1 は X 軸を意味しています）。何も打たなくても X 軸を指定するデフォルト値が入っているので、

```
root [3] hist->GetMean()
(const Double_t)8.33116798564013245e-03
```

と入力します。出力された値は、このヒストグラムの平均値です。ウインドウの右上に既に表示されている値と同一なはずです。同様にして、TH1D::GetRMS() も試してみましょう。

今度は

```
root [4] hist->Set
```

でタブ補完をすると、Set で始まる関数がたくさん表示されます。これらのメンバ関数はセッター（setter）と呼ばれ、ゲッターと対をなすものです。ゲッターとセッター以外にも多くのメンバ関数が存在しますが、ここでは説明しません。試しに

```
root [4] hist->SetLineColor(2)
root [5] hist->SetXTitle("x")
root [6] hist->SetYTitle("Number of Events")
root [7] hist->Draw("e")
```

としてみましょう。出力結果と見比べて、何が起きたか分かるはずです。先ほどまでは TH1D::Draw の引数に何も与えていませんでしたが、今度は"e"という引数がついています。なぜこれで動作したかというと、これも、引数のデフォルト値が存在していたためです。

```
root [8] hist->Draw(
```

でタブキーを押して、意味を理解して下さい。first_example.C では、引数のデフォルト値を設定していません。そのため、引数を必ず両方とも指定しないと正しく動作しません。

[*10] ROOT や PAW で「RMS」と言った場合、これは二乗平均ではなく標準偏差を指します。統計学の用語としては不適切なのですが、歴史的理由で RMS という言葉を使い続けているそうです。そのためこの業界では RMS という用語を間違って覚えている人が沢山いますが、優しく注意してあげて下さい。

[*11] この書き方は、クラス TH1D のメンバ関数 GetMean という意味です。

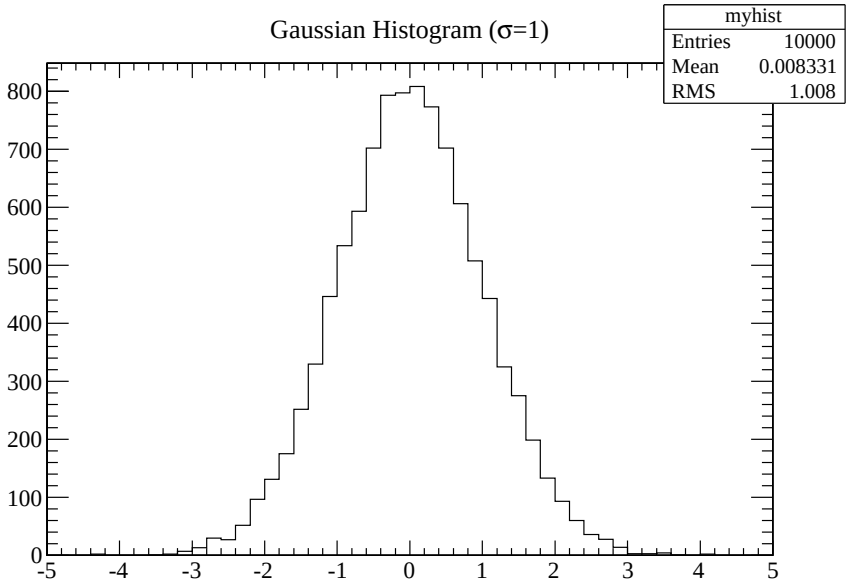

図 2.5　`~/.rootlogon.C` を使って図 2.4 の見栄えを変更したもの

2.3　やっておきたい初期設定

さて、図 2.4 を見ると、背景色が薄い灰色です[*12]。これでは通常の論文で使用するには問題があります。研究室内のみで使うような図だとしても、プリンタのインクの無駄です。ヒストグラムのタイトル部分を見ると、“σ” と他の文字のフォントが一致していません。文章自体のフォントに対しても太すぎます。これでは格好悪いので、少し ROOT の設定を変更してみます。

コード 2.3 を、`~/.rootlogon.C` として保存して、再度 ROOT を起動しましょう[*13]。その名が示す通り、ROOT を起動したとき（ログオンしたとき）に読み込まれるファイルです。再び `first_script.C` を走らせると、図 2.5 のような結果が得られます。まだまだ見栄えを変更可能ですが、ひとまずは良しとします。もし図 2.4 のほうが好みであれば、コード 2.3 の不要な行を消していきましょう。学会の発表資料では太い文字のほうが可読性が高いので、デフォルトのフォント（Helvetica）のままでも良いでしょう。どんな設定項目があるかは、`http://root.cern.ch/root/html/TStyle.html` を参照して下さい。

[*12] ROOT 5.30 から、見た目のデフォルト設定が変更になりました。そのため新しい ROOT を使っている場合には、この節の記述は古い可能性があります。

[*13] コード 2.1 では、`hist` というインスタンス（のポインタ）を自分で作りました。しかし、`gROOT` や `gStyle` というインスタンスは作った記憶がありません。これは ROOT が内部的に既に持っている、グローバル（global）インスタンスです。

コード 2.3 .rootlogon.C の例

```
{
  // You MUST do the following settings regardless of your taste
  gROOT->SetStyle("Plain"); // Set the overall design to "Plain" mode
  gStyle->SetTitleBorderSize(0); // Remove the boarder from title panels
  gStyle->SetFrameFillColor(0); // Set the frame color to white
  gStyle->SetCanvasColor(0); // Set the canvas color to white
  gStyle->SetPadBorderSize(0); // Remove uneccesary margin in pads
  gStyle->SetPalette(1); // Rainbow color palette

  // Change the default position of titles to top-center (optional)
  gStyle->SetTitleAlign(22);
  gStyle->SetTitleX(0.5);
  gStyle->SetTitleY(0.95);

  // Show the ticks on right Y axes and top X axes (optional)
  gStyle->SetPadTickX(1);
  gStyle->SetPadTickY(1);

  // Set the default font to Times (optional)
  Int_t fontid = 132;
  gStyle->SetStatFont(fontid);
  gStyle->SetLabelFont(fontid, "XYZ");
  gStyle->SetLabelFont(fontid, "");
  gStyle->SetTitleFont(fontid, "XYZ");
  gStyle->SetTitleFont(fontid, "");
  gStyle->SetTitleOffset(1.2, "XYZ");
  gStyle->SetTextFont(fontid);

  gStyle->SetFuncWidth(2); // Set the default line width of functions to 2
  gStyle->SetLegendBorderSize(0); // Remove the legend boarder
}
```

3　C++ の基礎

この章では、プログラミングや C++ の初心者向けに、C++ の簡単な解説を行います。第 2 章の内容や用語がチンプンカンプンだった人は、この章を読んでから先に進むのが良いでしょう。筆者が重要だと思う箇所だけを抜粋して解説しますので、網羅的な C++ の解説書も参照することをお勧めします。

ただし、いきなり一般書籍を購入しても当たり外れがあります。特に、C++ と銘打っていても、コードの書き方の癖が C 言語に近い書物はやめたほうが無難です[*1]。例えば

```
int i;
for(i = 0; i < 100; i++){
  // some codes...
}
```

のように書かれている本よりも

```
for(int i = 0; i < 100; i++){
  // some codes...
}
```

と書かれている本を選んで下さい。また、なるべくクラス（class）の概念が早めに登場する本が良いでしょう。いきなり一般書籍を読み進める前に、まずは本書を眺めて C++ 言語やプログラミングとは何かを掴んで下さい。プログラミングの専門家を目指すわけではないので、研究室では「使って覚える」を実践するべきです。インターネット上で日本語で閲覧可能なものでは、以下の 2 つをお勧めします。

- C++ 入門
  http://www.asahi-net.or.jp/~yf8k-kbys/newcpp0.html
- ATLAS Japan C++ Course
  http://www.icepp.s.u-tokyo.ac.jp/~sakamoto/education/atlasj/cplusplus/index.html

この他にも「C++ 入門」などでインターネットを検索すれば大量に出てきますので、好みに合うものを選択してください。

C++ が分かれば ROOT は理解しやすいのですが、ユーザの利便性を考えて、ROOT には独自の仕様が存在します。このような両者の違いについても、この章では説明します。

[*1] C++ の入門書ではありませんが、『Numerical Recipes in C++』は悪書の例です。

コード 3.1 hello_world.cxx

```
1 #include <cstdio>
2
3 int main()
4 {
5   printf("Hello World!\n");
6
7   return 0;
8 }
```

3.1 Hello World!

まず好きなエディタを使って、コード 3.1 を hello_world.cxx というファイル名で保存して下さい[*2]。次にターミナルから、

```
$ g++ hello_world.cxx
$ ./a.out
```

と打ちます[*3]。

```
Hello World!
```

と表示されるはずです。この一連の作業は、

1. あなたが hello_world.cxx というプログラムをエディタで作成し
2. あなたが g++ というコマンド[*4]に hello_world.cxx をコンパイルするように指示を出し
3. g++ が hello_world.cxx をコンピュータが実行できるファイル a.out に変換し
4. あなたが a.out を実行し
5. “Hello World!” という文字列が出力された

ということです。これがプログラミングをするという作業の基本的な流れです。

a.out という変な名前は、g++ が作成する実行ファイルのデフォルトの名前です。いくつもプログラムを作成したら、区別できなくなってしまいます。そこで g++ に引数をつけて

```
$ g++ hello_world.cxx -o hello_world
$ ./hello_world
```

とすれば、好きな名前で実行ファイルを作成してくれます。

さて、コード 3.1 の解説です。まずこのプログラムは、“Hello World!” と呼ばれる、初心者の解説向けによく用いられるものです[*5]。このプログラムの主な作業は

```
  printf("Hello World!\n");
```

[*2] Emacs、vi、gedit、TextEdit など何でも構いません。

[*3] もし g++ が見つからないという内容のエラーが出たら、お使いのコンピュータに GCC をインストールして下さい。「Scientific Linux GCC intall」「OS X GCC intall」などで検索すれば方法は見つかるはずです。Scientific Linux の場合は yum コマンドを使って GCC を入れることができます。また OS X の場合は Xcode を Apple からダウンロードしてインストールして下さい。

[*4] GNU Compiler Collection（GCC）に含まれる、C++ 用のコンパイラです。

[*5] http://ja.wikipedia.org/wiki/Hello_world などに解説あり。

の部分が担っています。printf という関数（function）を使って、“Hello Wolrd!” という文字列を出力させています[*6]。日本語で「関数」と言うと、普通は $y = f(x)$ のような数学の関数を想像するでしょう。しかしプログラミングの世界の関数は、必ずしも数字を扱うものではありません。“function” という単語は「機能」という訳語も持ちます。特定の機能をもった命令の集まりが関数です。

printf という名前には、意味があります。“print” という単語と “format” という単語を組み合わせたものです。文字列の書式を整えて出力する関数なので[*7]、このような名前になっています。関数の名前は、何をする機能を持っているか分かるようになっていなくてはなりません。数学では $y = f(x)$、$y = g(x)$ のように抽象的な名前を使いますが、プログラムの中で

```
  g("Hello World!\n");
```

と書かれていては、可読性が悪くなります。

さて、“\n” という 2 文字は何でしょうか。これは改行を表す特殊文字です。2 文字で 1 文字だと考えて下さい。試しにコード 3.1 から “\n” を取り除いてコンパイルし、実行してみて下さい。改行の有無で出力結果が変わります。

この “Hello World!” という文字列のことを、printf に渡した引数（ひきすう、argument）と言います[*8]。好きな文字列を渡すことによって、出力結果が変わります。“Hello World!” という文字列しか出力できない関数を作るのではなく、このように引数を変更することで結果を変更できるのが、関数を用意することの最大の利点です。

それではなぜ、この printf という関数を我々は使うことができるのでしょうか。その答えはコード 3.1 の先頭にあります。この行はインクルード（include）文と呼ばれ、他のファイルに既に存在している printf を見えるようにしています[*9]。したがって、この箇所を

```
//#include <cstdio>
```

のようにコメントアウト（comment out）[*10]してコンパイルすると、

```
$ g++ -Wall hello_world.cxx
hello_world.cxx: In function ‘int main’():
hello_world.cxx:5: error: ‘printf was not declared in this scope
```

とエラーを吐きます。コンパイラは printf が一体なんなのか分からなくなるわけです。

このエラーメッセージを読むと、“In function ‘int main()’” と書かれています。そうです、この main というのも関数です。この関数のことを main（メイン）関数と呼びます。main 関数は、プログラム中に必ず存在しなくてはいけません。main 関数の中に書かれた内容が、プログラムの実行時に呼び出される決まりになっているからです。「関数の中」とは { から } の中を指しています。この括弧を書くことで、コンパイラは main 関数の範囲を理解できるようになります。

main の前にある int（integer の int）は、main 関数の返り値（return value）の型（type）を示しています。一般的に、関数は呼び出されると何か値を返します。数学の二次関数 $y = f(x) = ax^2 + bx + c$ であれば、関数 f に x という値を入れると、$ax^2 + bx + c$ を返しますね。これと同じです。main 関数の返り値は、必ず整数（integer）でなくてはいけません。この返り値が 0 であれば、main 関数が正常終了したということを示します。最後に

```
  return 0;
```

[*6] C++ では std::cout を紹介するべきですが、ROOT の Form 関数や Python の文字列操作で printf に近い操作が出てくるため、あえて printf を使っています。

[*7] この書式の機能は今は使っていません。

[*8] 文字列は常に二重引用符で囲む必要があります。二重引用符自体を文字列の中で使用する場合には、"\""のように、バックスラッシュを前方に置きます。

[*9] cstdio は “C”、“standard”、“input/output” の合成語です。C 言語の時代に作られた、標準入出力のためのライブラリです。

[*10] 先頭が “//” で始まる行は、全てコンパイラに無視されます。

コード 3.2 triple.cxx

```
1  #include <cstdio>
2
3  int triple(int v)
4  {
5    return 3*v;
6  }
7
8  int main()
9  {
10   int before = 15;
11   int after  = triple(before);
12
13   printf("Before: %d\n", before);
14   printf("After : %d\n", after);
15
16   return 0;
17 }
```

と書くことで、確かに 0 を返す設計になっています。返り値は、return を使って返すことができます。もしこれを 0 ではなく−1 などにすれば、このプログラムは以上終了したと OS 側が判断します。

3.2 型と関数

前節の説明では、型、関数、引数、返り値という用語がでてきました。ここでは、もう少し例を挙げて説明します。コード 3.2 を作成してコンパイルし、実行してみましょう。15 という整数値を持つ変数（variable）before を

```
  int before = 15;
```

のようにして作成しています。数学の変数は、その中身をいつでも整数や無理数や複素数に変更できます。しかし C++ の場合は、その変数の種類を後から変更できません。変数 before の中身は、いつでも整数値です。先頭の int は、変数 before が常に整数値を持つという意味です。これを型（type）と呼びます。つづいて= 15 というのが出てきますが、これは「before に 15 を代入する」という意味です。「before と 15 は等しい」という意味ではありません。ここは、数学の等号と使用方法が異なります。もしこの直後に

```
before = before*before + 1;
```

という文を足すと、before の値が 226 に変更されます。before の値が二次方程式 $x = x^2 + 1$ の解に自動的に変更されたりはしないのです。C++ の=は、数学の等号ではなく代入を表します。

関数 triple は、引数に与えられた整数値 v を、3 倍して返す関数です[*11]。整数を 3 倍してもやはり整数なので、返り値は int になっています。このような関数を作ってしまえば、

```
  int after = triple(before);
```

として変数 after に関数の返り値を代入することができます。この右辺が呼ばれると、処理は関数 triple の行に飛び、

[*11] C++ では、四則演算の記号 +、−、×、÷ はそれぞれ +、−、*、/ で表します。

```
return 3*v;
```

で値が返されて、左辺に代入されます。

そして最後に、変数 before、after の中身を

```
printf("Before: %d\n", before);
printf("After : %d\n", after);
```

で出力させています。ここでは、printf の “format” の機能を利用しています。%d という文字列は、int の変数を文字列に整形して出力するという意味です。printf はこのように、複数の引数（可変長引数）を取ることが可能です。

int 以外にも、C++ には何種類か型があります。代表的なものが、double（ダブル）です。double 型は、int と異なり小数を使うことができます。なぜ double という名前かと言うと、同じように小数を扱う型に float（フロート）があるからです。double は float に比べてメモリを 2 倍（double）消費します。しかしその分、精度がよくなります[*12]。

さて、コード 3.2 の例では、triple 関数は引数と返り値が int 型でした。もし before の値が 16.9 だとすれば、after の値は期待通りに動作しません。そこで引数と返り値を double に変更したものを追加したのが、コード 3.3 です。同名の triple 関数が 2 つありますが、片方は int 用で片方は double 用です。このように同じ名前の関数に対して、異なる引数を与えることで処理内容を変更することを、関数のオーバーロード（overload）と言います。コンパイラが引数の違いを適切に見つけ出し、異なる処理をしてくれます。

コード 3.3 はコード 3.2 と違い、main の前に実体を伴わない 2 つの関数が書かれています。これは関数の前方宣言（forward declaration）と呼ばれるものです。今はなくても構いませんが、ヘッダーファイルを分割して書くようになるときには必須の作業です。このように返り値と引数だけ先に書いておくことで、実際の関数の中身を知らなくても（関数が後で定義されていても）、コンパイラは main 関数の中で triple が出てきても問題なく処理を続行できるようになります。プログラムの可読性を高めるという意味もあります。main 関数の後に、実際の triple 関数の中身が記述されています。これを、関数の定義（definition）と呼びます。

新たな printf の使い方として、

```
printf("Before: %f\n", before_d);
printf("After : %f\n", after_d);
```

のように %d ではなく %f というのが登場しています。これは、double 型を整形するための特殊な文字列です[*13]。

なぜこれらの例で、2 倍にする関数ではなく 3 倍にするものを選んだかというと、double という単語が C++ の予約語（reserved word）だからです。C++ が既に確保している名前を、ユーザが勝手に使うことはできません。printf のように一般的に使われる関数名も、使わないことが推奨されます。もしあなたが “f” という文字を出力する関数を printf という名前で作成したら、他の人は混乱するでしょう。

3.3 if 文と関係演算子

ここまでで、関数の簡単な使い方が分かりました。しかし、四則演算程度しかまだやり方をしりません。2 つの数字の大小を比べるにはどうしたら良いでしょうか。この節では、関係演算子と if 文の使い方を説明します。

[*12] C++ で扱われる小数には精度がつきまといます。例えば $\frac{3}{2}$ を小数に直すと、どれだけ小さいほうの桁を見ても 0 が続きます。しかしコンピュータ上にこのような無限の精度を持つ数を定義すると、メモリが無限大必要になります。これでは実現不可能なため、ある程度の精度を犠牲にします。例えば円周率を double 型で扱いたい場合には、有効桁数が 15 桁しかありません。 3.14159265358979 までしか精度が保証されないのです。float にした場合はさらに精度が下がり、7 桁しか有効桁数がありません。したがって、1.3×10^{20} と 3.4×10^{-13} の引き算をそのまま C++ で実行しても、数学的に正しい数字は得られないので注意が必要です。

[*13] 他にもフォーマットの種類がいくつか存在しますので、「printf」で検索してみてください。

コード 3.4 には、新しい関数 min と max を作りました。初めて、

```
  if(v1 < v2){
    ret = v1;
  } else {
    ret = v2;
  }
```

という if 文が出てきます。これを日本語訳すると、「もし（if）v1 が v2 よりも小さければ、ret に v1 を代入しなさい。そうでなければ（else）v2 を代入しなさい」となります。<は関係演算子や比較演算子（relational operator、comparison operator）と呼ばれるものです。もし左辺が小さければ 1 を、そうでなければ 0 を返します。つまり

```
int a = 10 < 20;
int b = 10 > 20;
```

とすれば、a と b の値はそれぞれ 1 と 0 になります。if 文の中身（{ と } で囲まれた部分）は、条件式が 1 のときだけ実行されます。v1 が小さいときは返り値として v1 を返して min 関数は終了します。

コード 3.3 triple2.cxx

```
1  #include <cstdio>
2
3  int triple(int v);
4  double triple(double v);
5
6  int main()
7  {
8    int before_i = 15;
9    int after_i  = triple(before_i);
10
11   printf("Before: %d\n", before_i);
12   printf("After : %d\n", after_i);
13
14   double before_d = 16.9;
15   double after_d  = triple(before_d);
16
17   printf("Before: %f\n", before_d);
18   printf("After : %f\n", after_d);
19
20   return 0;
21 }
22
23 int triple(int v)
24 {
25   return 3*v;
26 }
27
28 double triple(double v)
29 {
30   return 3*v;
31 }
```

コード 3.4 minmax.cxx

```
1  #include <cstdio>
2
3  double min(double v1, double v2);
4  double max(double v1, double v2);
5
6  int main()
7  {
8    printf("%f is smaller\n", min(39.2, 48.5));
9    printf("%f is larger\n",  max(103.8, -3.2));
10
11   return 0;
12 }
13
14 double min(double v1, double v2)
15 {
16   double ret;
17   if(v1 < v2){
18     ret = v1;
19   } else {
20     ret = v2;
21   }
22
23   return ret;
24 }
25
26 double max(double v1, double v2)
27 {
28   return v1 > v2 ? v1 : v2;
29 }
```

<があれば、当然>も存在します。同様に、 <=と>=も存在します。それぞれ見たままの通りで、<、>、≤、≥ を表します。一見その機能が分かりにくい、==と!=も存在します。前者は両辺が等しいときに 1 を返し、後者は等しくないときに 1 を返します[*14]。

>を使って、max 関数も定義しました。しかし今度は

```
  return v1 > v2 ? v1 : v2;
```

という、わけの分からない記号が並んでいます。これは 3 項演算子と呼ばれる記法で、最初は分かりにくいですが慣れると min 関数の中身よりも分かりやすいはずです。?と:で、これは 3 つの部分に分かれています。日本語に訳すと、「もし v1 が v2 よりも大きければ ? v1 を返す : そうでなければ v2 を返す」となっています。

3.4 for 文

なぜコンピュータを使ってプログラミングをするのかと言えば、人間には手に負えない、複雑な計算をする必要があるからです。特に同じ作業を繰り返す場合には、コンピュータを利用すると劇的に処理速度が向上します。そこで

[*14] 文字列の比較は、比較演算子ではできません。これは 3.7 節で説明します。

コード 3.5 pi.cxx

```
1  #include <cstdio>
2
3  double pi(int n);
4
5  int main()
6  {
7    for(int i = 1; i <= 100; i++){
8      printf("n = %d: pi = %f\n", i, pi(i));
9    } // i
10
11   return 0;
12 }
13
14 double pi(int n)
15 {
16   int total = 0; // number of points inside a unit circle
17   double d = 1./n; // grid length of points
18
19   for(int j = -n; j < n; j++){
20     double y = d*(j + 0.5);
21     for(int i = -n; i < n; i++){
22       double x = d*(i + 0.5);
23       if(x*x + y*y < 1.){ // check if (x, y) exists inside a unit circle
24         total += 1;
25       } // if
26     } // i
27   } // j
28
29   return total/double(n*n);
30 }
```

登場するのが for（フォー）文です。for 文の使い方を学ぶために、非常に初等的な手段で円周率 π を計算してみましょう。

コード 3.5 では、$-1 < x < 1$、$-1 < y < 1$ の範囲に等間隔で並ぶ、$4n^2$ 個の格子点の場所を計算しています。これらの点のうち、半径 1 の単位円の内部に存在する個数を数え上げると、$\simeq \pi n^2$ 個になるはずです[*15]。

以下の、main の中身が for 文です。

```
  for(int i = 1; i <= 100; i++){
    printf("n = %d: pi = %lf\n", i, pi(i));
  } // i
```

最初の行を日本語にすると、「i が 1 の状態から開始し、100 以下の間だけ以下の作業を繰り返しなさい。ただし、1 度繰り返した直後に、i は 1 ずつ増やせ」となります[*16]。つまり、i が 1 から 100 まで変化します。i++ という表現は始めて出てきました。これは

[*15] $n \to \infty$ の場合、本当に π に収束するかは真面目に考えていません。

[*16] i は “index” の “i” です。for 文は入れ子にすることができ、そのような場合は j や k をその後の添え字として使います。ただし、小文字の L l を k の後に使うのは避けて下さい。数字の 1 と視覚的に区別が難しいためです。

```
i = i + 1;
```

や

```
i += 1;
```

と同じ意味を持ちます[*17]。

for 文の文法さえ分かれば、pi 関数の中で使われている for 文も理解できるでしょう。関係演算子が<の場合と<=の場合で繰り返し回数が変わることに注意して読んで下さい。

ここまでで、あなたは四則演算を使った膨大な計算をできるようになりました。例えば

$$\sin(x) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{3.1}$$

のような手では面倒な計算も（double の精度で）可能になります。試してみて下さい。

[*17] なぜ複数の書き方が可能かというと、「1 だけ増加させた」ということをより明示的にするためです。

3.5 クラス

int 型と double 型の使い方は覚えました。しかし、(x, y, z) や (p_x, p_y, p_z, E) のようなベクトルを考えた場合、変数を 3 つや 4 つも自分で管理するのは面倒です。2 つのベクトルの足し算をして新しいベクトルを作るときに、

```
double x1 = 1.5, y1 = 2.3, z1 = -0.4;
double x2 = -3.1, y2 = 5.6, z2 = 1.9;
double x3 = x1 + x2, y3 = y1 + y2, z3 = z1 + z2;
```

と書くと見づらいです。可能ならば

```
Vector3D v1(1.5, 2.3, -0.4);
Vector3D v2(-3.1, 5.6, 1.9);
Vector3D v3 = v1 + v2;
```

のように書けたほうがすっきりします。これがクラス（class）の発想です。自分で好きな型を作ることが可能になります。

それでは、早速クラスを自分で作ってみましょう。ソースコードを眺めながら、クラスとは何かを理解してください。コード 3.6、3.7、3.8 は、3 次元ベクトルを扱うためのクラスの作成例です。virtual という予約語が出てきますが、今はこれを無視して読み飛ばしてください。3.10 節で説明します。

コード 3.6 Vector3D.h

```
1  #ifndef VECTOR_3D
2  #define VECTOR_3D
3
4  class Vector3D
5  {
6  private:
7    double fX;
8    double fY;
9    double fZ;
10
11 public:
12   Vector3D();
13   Vector3D(double x, double y, double z);
14   Vector3D(const Vector3D& other);
15   virtual ~Vector3D();
16
17   virtual double X() const {return fX;}
18   virtual double Y() const {return fY;}
19   inline virtual double Z() const;
20   virtual void   Print() const;
21
22   Vector3D& operator=(const Vector3D& other);
23   Vector3D  operator+(const Vector3D& other);
24   Vector3D  operator-(const Vector3D& other);
25   double    operator*(const Vector3D& other);
26 };
27
28 double Vector3D::Z() const
```

```
29 {
30   return fZ;
31 }
32
33 #endif // VECTOR_3D
```

コード 3.7 Vector3D.cxx

```
1  #include "Vector3D.h"
2  #include <cstdio>
3
4  Vector3D::Vector3D()
5  {
6    fX = fY = fZ = 0;
7  }
8
9  Vector3D::Vector3D(double x, double y, double z)
10 {
11   fX = x;
12   fY = y;
13   fZ = z;
14 }
15
16 Vector3D::Vector3D(const Vector3D& other)
17 {
18   fX = other.fX;
19   fY = other.fY;
20   fZ = other.fZ;
21 }
22
23 Vector3D::~Vector3D()
24 {
25 }
26
27 void Vector3D::Print() const
28 {
29   printf("(x, y, z) = (%lf, %lf, %lf)\n", fX, fY, fZ);
30 }
31
32 Vector3D& Vector3D::operator=(const Vector3D& other)
33 {
34   if (this != &other) {
35     fX = other.fX;
36     fY = other.fY;
37     fZ = other.fZ;
38   }
39
40   return *this;
41 }
42
43 Vector3D Vector3D::operator+(const Vector3D& other)
44 {
```

```
45   return Vector3D(fX + other.fX, fY + other.fY, fZ + other.fZ);
46 }
47
48 Vector3D Vector3D::operator-(const Vector3D& other)
49 {
50   return Vector3D(fX - other.fX, fY - other.fY, fZ - other.fZ);
51 }
52
53 double Vector3D::operator*(const Vector3D& other)
54 {
55   return fX*other.fX + fY*other.fY + fZ*other.fZ;
56 }
```

コード 3.8 Vector3D_main.cxx

```
1  #include <cstdio>
2  #include "Vector3D.h"
3
4  int main()
5  {
6    Vector3D v0; // default constructor
7    Vector3D v1(1.5, 2.3, -0.4); // constructor with arguments
8    Vector3D v2 = Vector3D(-3.1, 5.6, 1.9); // operator=, constructor
9    Vector3D v3 = v1 + v2; // operator=, operator+
10   Vector3D v4(v1 - v2); // copy constructor, operator-
11   double product = v1*v2; // operator*
12
13   v0.Print();
14   v1.Print();
15   v2.Print();
16   v3.Print();
17   v4.Print();
18   printf("v1*v2 = %f\n", product);
19
20   return 0;
21 }
```

3 つのファイルに分割したのは、可読性と可搬性[*18]を高めるためです。もし全てを 1 つのファイルに書いてしまうと、せっかく作った Vector3D というクラスを他のプログラムで使うのが面倒になります。クラスの記述と main 文を分けておくことで、他の main 文を書いたときにも、クラスの記述を何度も繰り返す必要がなくなります。Vector3D.h と Vector3D.cxx は、それぞれヘッダーファイル（header file）、ソースファイル（source file）と呼ばれます[*19]。まずは新しく作ったこのプログラムをコンパイルして、コンパイル済みの実行ファイルを走らせてみましょう。

```
$ g++ -c Vector3D.cxx
$ g++ -c Vector3D_main.cxx
$ g++ Vector3D.o Vector3D_main.o -o Vector3D
```

[*18] 他のプログラムでも使い回しが効くという意味です。

[*19] C++ のヘッダーとソースの拡張子には、いくつかの流儀があります。例えば ROOT では、“.h” と “.cxx” という拡張子をそれぞれに使っています（ただし、スクリプトファイルには区別のために “.C” を採用しています）。また Geant4 では、“.hh” と “.cc” を使っています。C++ のソースコードの拡張子には、他にも “.C”、“.c++”、“.cpp” なども世の中では使われています。どの拡張子を使うかは本質的な問題ではありません。本書では、ROOT の流儀に合わせて “.h” と “.cxx” を採用します。

```
$ ./Vector3D
```

1 行目と 2 行目では-c オプションをつけて、Vector3D.cxx と Vector3D_main.cxx をコンパイルだけしています。これらはコンパイル後に拡張子が.o のオブジェクトファイル（object file）に変換されます。「コンパイルだけ」というのは、実行ファイルを作成しないということです。Vector3D.cxx には main 文が存在せず、また Vector3D_main.cxx にはクラスの中身が書かれていないので、そのままでは実行ファイルが作成できません。3 行目で 2 つのオブジェクトファイルを結合し、Vector3D という実行ファイルが作成されます。今のコードでは必要性をあまり感じませんが、膨大な量のソースコードをコンパイルするときには、このような分割コンパイルは必須です。修正個所だけコンパイルし直すことで、時間を節約できるからです。

3.5.1 クラス宣言

それでは、Vector3D クラスの説明に移ります。コード 3.6 では、クラス Vector3D の宣言（declaration）を行っています。「宣言」とは、クラスの基本仕様を書く作業のことです。C++ では int や double のような基本的な型しか持っていませんので、あなたが新しいクラスを作るときには、それがどんなものであるかを教えてやる必要があります。

クラスの宣言に最低限必要な部分は、次の箇所だけです。他の箇所は、そのクラスがどんな性質を持つかを記述するためのものですので、以下の記述だけでは何の役にも立たないクラスができあがります。

```
#ifndef VECTOR_3D
#define VECTOR_3D

class Vector3D
{
};

#endif // VECTOR_3D
```

最後のセミコロンを忘れやすいので注意してください。

#で始まる行は、おまじないです。色々なファイルから Vector3D.h を何度もインクルードすると、あたかも Vectr3D クラスが何度も宣言されたように見え、コンパイルエラーが起きます。これを防ぐために、VECTOR_3D と文字列をここでは定義しています。#ifndef VECTOR_3D は、「もし VECTOR_3D が定義されていなかったら（IF Not DEFine）」という意味です。VECTOR_3D が定義されていないときだけ、#endif までの内容が実行されます。つまり、VECTOR_3D を#define し、クラス宣言を行います。この仕組みを「インクルードガード（inclue guard）」と呼びます[*20]。

3.5.2 メンバ変数

クラスが保持する情報は、メンバ変数（member variable）に格納されます。コード 3.6 では、fX、fY、fZ[*21] という 3 つのメンバ変数が存在します。今はメンバ変数に double 型しか使っていませんが、int を使ったり、他のクラスを使うことも可能です。今回は 3 次元ベクタを記述するためのクラスの例ですので、XYZ 座標をこれらのメンバ変数

[*20] VECTOR_3D という文字列は、VECTOR3D でも HOGE でも、別に好きなもので構いません。ただし、コードを読む人が分かりやすいもので、なおかつ、他のプログラムで使われていなさそうな名前にしてください。#endif の後のコメント行は無くても構いませんが、長いコードの場合は、何に対応する#endif なのかを分かりやすくするため、このような書き方をすることがあります。

[*21] 変数名の最初の f は、メンバ変数と他の変数の区別をしやすくするためのものです。これは ROOT で使われる変数名の命名規則ですが、他にも mX や m_x と書いたり（member の m）、単に x とする文化もあります。

が double 型で保持します。これらメンバ変数の直前に書かれている private: は、「次の変数はプライベート変数だよ」という目印です。個人情報のようなもので、特別に公開する手段を持たない限り、本人以外は外から知ることができません。

3.5.3 コンストラクタ

Vector3D()、Vector3D(double x, double y, double z)、Vector3D(const Vector3D& other) の 3 つの関数は、コンストラクタ（constructor）と呼ばれる特殊な関数です。クラス名と同じ関数名になっています。これらの関数がいつ使われるかというと、クラスを実際に使用し始める瞬間です。コード 3.8 では v0 という変数[*22]を

```
Vector3D v0;
```

のようにして作成しています。このように作成した変数では、Vector3D() のほうのコンストラクタが呼び出されます。このような引数を持たないコンストラクタを、デフォルトコンストラクタ（default constructor）と呼びます。また

```
Vector3D v1(1.5, 2.3, -0.4);
```

のように引数をつけて変数を作成した場合は、Vector3D(double x, double y, double z) のほうが呼び出されます。それぞれのコンストラクタの実体はコード 3.7 に書かれており、それぞれのコンストラクタで fX、fY、fZ 全てにゼロを代入するか、与えられた引数を代入していることが分かります。コンストラクタはクラスが使われる瞬間に呼び出されるため、一般的にはメンバ変数などの初期化に使われます。Vector3D では、(x, y, z) をデフォルトコンストラクタで $(0, 0, 0)$ に設定するか、与えられた 3 変数を代入することによって、初期化しています。

もしあなたがどちらのコンストラクタも作らないと、コンパイラは自動的にデフォルトコンストラクタを作るので注意が必要です。しかし引数を持つコンストラクタを 1 つでも作れば、コンパイラはデフォルトコンストラクタを作成しません。そのため、今回の例ではデフォルトコンストラクタと引数を持つコンストラクタを両方作成しています[*23]。コンパイラにデフォルトコンストラクタを自動生成させた場合、メンバ変数はその型やクラスの初期値を持ちます。この例では、fX、fY、fZ はどれも double 型なので、初期値はゼロになります。

デフォルトコンストラクタの使用方法は、以下の 2 通りの書き方がありますが、機能としては全く同一です。

```
Vector3D v0;
Vector3D v0();
```

コンストラクタも関数的に振る舞うため本来は () の部分が必要なのですが、引数を持たないデフォルトコンストラクタでは省略することができます。

3.2 節で説明したように、一般的な関数は返り値を持ちます。しかしコンストラクタは特殊な関数であり、返り値は持ちません。返り値を持たない関数には通常 void を付ける必要がありますが、コンストラクタは返り値を持たないのが分かっているので、void を書く必要はありません。

これらコンストラクタと別に

```
Vector3D(const Vector3D& other);
```

として宣言されているコンストラクタをコピーコンストラクタ（copy constructor）と呼びます。既に存在しているインスタンスから、全く同じ内容のインスタンスを作成するときに使用します。コード 3.8 の例では、

[*22] インスタンス（instance）とも呼びます。

[*23] このようにデフォルトコンストラクタを作らなくても、Vector3D v0(0, 0, 0) と書けば全く同じ結果が得られます。しかし後々複雑なプログラムを書くようになると、どんな値を詰めるか考えないで、ひとまず変数を用意する場面が出てきます。このようなときにデフォルトコンストラクタは必要になります。

```
Vector3D v4(v1 - v2);
```

が該当します。v1 - v2 の部分や&の意味は節 3.5.5 で説明しますが、ここでは引数に Vector3D が与えられ、その中身が v4 にそのままコピーされます。

この例ではコピーコンストラクタを本当は自作する必要はありません。なぜなら、コピーコンストラクタを明示的に作成しなかった場合、これもコンパイラによって自動生成されるからです。コード 3.7、3.6 では、わざわざ自分で書きましたが、その必要はありません。特に、メンバ変数の中身をそのままコピーするだけのコピーコンストラクタならば、自分でコードを書かずにコンパイラに任せてしまいましょう。人間がやるとバグの元になります。

3.5.4 メンバ関数

プライベートなメンバ変数とコンストラクタを持つだけでは、そのクラスが持つ情報にユーザはアクセスすることができません。そこで、3.5.2 節に書いたように、「特別に公開する手段」が必要となります。コード 3.6 に書かれた X()、Y()、Z() の 3 つの関数は、メンバ変数 fX などを取り出すための関数で、その中身は短く、単純にメンバ変数の値を返しています。const 修飾子というものが出てきていますが、これはクラスの持つメンバ変数の値を変更しないという目印です。つけなくてもこの例では大差ありませんが、おまじないです。X() のような短い関数は、可読性の落ちない限り、このようにヘッダーの中に書いてしまうことが頻繁に行われます。ヘッダーの中に書き込むことで、処理速度の向上が見込まれるためです。これを関数のインライン化と言います。また、inline という予約語を使うことでインライン関数をクラス定義の外側に書くことができます。コード 3.6 では、Vector3D::Z() だけクラス定義の外側でインライン化させています。

メンバ変数の情報を取り出す以外にも、様々な動作をメンバ関数にさせることが可能です。コード 3.6 ではさらに、Print() というメンバ関数を追加しています。実行ファイル Vector3D を走らせれば分かるように、これは 3 次元ベクタの中身を表示するための関数です。この関数の実体は、やはりコード 3.7 に書かれています。

コンストラクタやメンバ関数をソースファイルに記述するとき、

```
void Vector3D::Print() const
```

のような記述方法をします。Print() という関数がどのクラスのメンバ関数なのかをはっきりさせるため、Vector3D::という所有格を明示する記述が必要になります。この Print() というメンバ関数ではメンバ変数を表示するだけなので、やはり const を付けておきましょう。

コード 3.7 で定義した Print() や X() という Vector3D クラスのメンバ関数は、

```
Vector3D v(1., 2., 3.);
double x = v.X();
v.Print();
```

のようにして、変数 v の後に “.” と関数名を繋げることにより、呼び出すことが可能です。この “.” は、日本語の「の」だと思えば良いでしょう。上記の例では、「変数 v の X() を呼び出す」「変数 v の Print() を呼び出す」のように理解してください。

3.5.5 演算子オーバーロード

せっかく 3 次元ベクタを扱うクラスを作っても、数学的な演算ができなければ役に立ちません。そこで、C++ には演算子オーバーロード（operator overloading）という機能があります。ベクタの加減や、内積といった演算が直感的に行えれば、ややこしいコードを何度も書く必要はなくなります。コード 3.8 では、ベクタの加減算と内積を行っていま

す。コンピュータには Vector3D 同士の足し算とは一体何をするべき作業なのか分かりません。そのため、+ や*演算子を使ったときにどのような処理を実行するべきかは、ユーザが決定してやる必要があります。これが演算子オーバーロードです。

コード 3.6 と 3.7 には、operator という文字の入った関数が出てきます。3.7 に定義された、Vector3D 同士の足し算の定義を見てみましょう。

```
Vector3D Vector3D::operator+(const Vector3D& vec)
{
  return Vector3D(fX + vec.X(), fY + vec.Y(), fZ + vec.Z());
}
```

3 次元ベクタ同士を足せば、その結果は当然 3 次元ベクタになります。したがって、operator+ の返り値は当然 Vector3D になります。また、operator+ は関数の形をしていますが、実際に使うときは

```
Vetor3D v3 = v1 + v2;
```

のように使います。

```
Vetor3D v3 = v1.operator+(v2);
```

のようには使わないので注意が必要です。

さて、operator+ の引数の記述はこれまで見たことがない形式です。まず const 修飾子がここでも出てきています。これは、引数 vec の中身を一切変更しませんという宣言です。v1 と v2 の足し算をしている最中に、v2 の中身が変わったりしたら困るからです。また、引数の型が Vector3D なのは当然です。Vector3D 同士の足し算を定義しているからです。& という初めて使う記号がその直後についています。これは、参照渡し（call by reference）と呼ばれる方法です。&がない場合、C++ では引数のコピーをその都度作成し、その関数を実行し終えると自動的にそのコピーが破棄されます。今はたった 3 つのメンバ変数しか持っていませんが、メンバ変数に長い文字列や画像データを持つクラスでは、いちいちコピーを作成しているとコンピュータ資源の無駄になります。そこで、&をつけた場合には、その関数は引数の本体を参照するようになります。

さて、+、−、*に加えて、代入演算子=がコード 3.8 では使われています。これは他の演算子と異なり、返り値が Vector3D ではなく Vector3D&です。また this という予約語が使われています[*24]。this は、インスタンスが自分自身を指し示すポインタです。したがって、

```
  if (this != &other) {
    fX = other.fX;
    fY = other.fY;
    fZ = other.fZ;
  }
```

の部分は、

```
v1 = v1;
```

のような、自分自身への代入を無駄に実行したときに読み飛ばされるようになっています。上記のような代入がされただけでは、返り値を持つ必要がありません。if 文の中の代入操作さえ終われば、左辺の v1 が何を返そうが、何も起きないからです。しかし、C++ では

```
v1 = v2 = v3;
```

[*24] 3.7 を読んでから、再度この段落を読んでみてください。

のような書き方もできます。この場合には、v2 に v3 が代入された後、v2 が自分自身への参照を返してくれないと、v1 への代入が行えません。そのため、operator=の返り値は Vector3D&でなくてはならないのです。

代入演算子はコピーコンストラクタと同様、明示的に書かれていなければコンパイラが自動生成します。デフォルトの代入演算子では、メンバ変数の内容を全く同一にコピーしたものを代入先に渡します。この例ではわざと自分で書きましたが、コピーコンストラクタと同様、コンパイラ任せにできる場合は書く必要はありません。

3.5.6 継承

クラスの面白い機能に継承（inheritance）があります。あるクラスの機能をそのまま受け継いだ（継承した）、他のクラスを作る機能です。ここでは、ローレンツベクタを例に考えることにします。ローレンツベクタは、空間情報 (x, y, z) に加えて、時間という新たな変数 t が追加されます。したがって、先ほど作成した Vector3D を継承した、変数 t を持つクラスを作れば、色々なコードを再利用できます。コード 3.9、3.10、3.11 に、Vector3D を継承した新しいクラス LorentzVector の例を示します。前と同様に、

```
$ g++ -c Vector3D.cxx
$ g++ -c LorentzVector.cxx
$ g++ -c LorentzVector_main.cxx
$ g++ LorentzVector.o Vector3D.o LorentzVector_main.o -o LorentzVector
$ ./LorentzVector
```

とすれば実行可能です。Vector3D.cxx のコンパイルも必要なことに注意してください。

コード 3.9 LorentzVector.h

```
1  #ifndef LORENTZ_VECTOR
2  #define LORENTZ_VECTOR
3
4  #include "Vector3D.h"
5
6  class LorentzVector : public Vector3D
7  {
8  private:
9    double fT;
10
11 public:
12   LorentzVector();
13   LorentzVector(double x, double y, double z, double t);
14   LorentzVector(const LorentzVector& other);
15   virtual ~LorentzVector();
16
17   virtual double T() const {return fT;}
18   virtual void   Print() const;
19
20   LorentzVector& operator=(const LorentzVector& other);
21   LorentzVector  operator+(const LorentzVector& other);
22   LorentzVector  operator-(const LorentzVector& other);
23   double         operator*(const LorentzVector& other);
24 };
25
26 #endif // LORENTZ_VECTOR
```

コード 3.10 LorentzVector.cxx

```
1  #include "LorentzVector.h"
2  #include <cstdio>
3  #include <cmath>
4
5  LorentzVector::LorentzVector() : Vector3D()
6  {
7    fT = 0;
8  }
9
10 LorentzVector::LorentzVector(const LorentzVector& other) : Vector3D(other)
11 {
12   fT = other.fT;
13 }
14
15 LorentzVector::LorentzVector(double x, double y, double z, double t)
16   : Vector3D(x, y, z)
17 {
18   fT = t;
19 }
20
21 LorentzVector::~LorentzVector()
22 {
23 }
24
25 void LorentzVector::Print() const
26 {
27   printf("(x, y, z, t) = (%lf, %lf, %lf, %lf)\n", X(), Y(), Z(), fT);
28 }
29
30 LorentzVector& LorentzVector::operator=(const LorentzVector& other)
31 {
32   if (this != &other) {
33     Vector3D::operator=(other);
34     fT = other.fT;
35   }
36
37   return *this;
38 }
39
40 LorentzVector LorentzVector::operator+(const LorentzVector& vec)
41 {
42   return LorentzVector(X() + vec.X(), Y() + vec.Y(), Z() + vec.Z(), fT + vec.fT);
43 }
44
45 LorentzVector LorentzVector::operator-(const LorentzVector& vec)
46 {
47   return LorentzVector(X() - vec.X(), Y() - vec.Y(), Z() - vec.Z(), fT - vec.fT);
48 }
49
50 double LorentzVector::operator*(const LorentzVector& vec)
```

```
51 {
52   return X()*vec.X() + Y()*vec.Y() + Z()*vec.Z() - fT*vec.fT;
53 }
```

コード 3.11 LorentzVector_main.cxx

```
1  #include "LorentzVector.h"
2  #include <cstdio>
3
4  int main()
5  {
6    LorentzVector v0; // default constructor
7    LorentzVector v1(1.5, 2.3, -0.4, 4.2); // constructor with arguments
8    LorentzVector v2 = LorentzVector(-3.1, 5.6, 1.9, -3.8); // operator=, constructor
9    LorentzVector v3 = v1 + v2; // operator=, operator+
10   LorentzVector v4(v1 - v2); // copy constructor, operator-
11   double product = v1*v2; // operator*
12
13   v0.Print();
14   v1.Print();
15   v2.Print();
16   v3.Print();
17   v4.Print();
18   printf("v1*v2 = %f\n", product);
19
20   return 0;
21 }
```

コード 3.9 では Vector3D のときと同様に、LorentzVector クラスの宣言を行っています。前と違うところは、Vector3D を継承している点です。

```
class LorentzVector : public Vector3D
```

と書くことで、Vector3D を継承したクラスになります。public はおまじないです。継承される側のクラスを基底クラス（base class）、親クラス、スーパークラス（super class）と呼び、また継承する側を派生クラス（derived class）、子クラス、サブクラス（sub class）と呼びます。

LorentzVector クラスには、新たに fT というメンバ変数が追加されています。fX などは、一切宣言されていません。他の変数は既に Vector3D が持っており、その変数ごと LorentzVector は継承しているので、宣言し直す必要がないからです。

fT を追加したのであれば、これを取り出すための関数も必要になります。これも同様に、X() などは既に Vector3D から継承済みなので、T() だけ追加すれば良いことになります。他のメンバ変数は private として Vector3D で宣言されていました。そのため、LorentzVector のメンバ関数からは、fX などは X() などを通じないと取り出せなくなっています。そのため LorentzVector::operator+ などの定義では、fX を直接触らずに X() を使っています。

コンストラクタは、新たに書き直しが必要です。コンストラクタは fX、fY、fZ、fT の全てを初期化する必要があるので、Vector3D のコンストラクタをそのまま使うことはできません。コード 3.10 では、fT の初期化作業が、Vector3D のコンストラクタに比べて増えています。ここで注目して欲しいのが、コード 3.10 のコンストラクタのうち、

```
LorentzVector::LorentzVector(const LorentzVector& other) : Vector3D(other)
{
  fT = other.fT;
}
```

や

```
LorentzVector::LorentzVector(double x, double y, double z, double t)
  : Vector3D(x, y, z)
```

で使われている、単独のコロン（:）の後ろの部分です。このような書き方をすると、LorentzVector のうち Vector3D に由来する部分を Vector3D のコンストラクタを使って初期化することができます。いちいち、fX などへの代入作業を書き直す必要がなくなります。

また、Print() や演算子も fT に関する記述を追加する必要があるので、新たに書き直しています。Print() のように、親クラスの持つメンバ関数と外見上全く同じメンバ関数を作成することを、関数のオーバーライド（override）と言います。fX、fY、fZ は private として宣言しました。そのため、LorentzVector からは X() などの関数を使わないとアクセスできなくなっていることに注意してください。

3.6 配列

int や double という型を使って、整数や有限桁の少数を扱うことができました。メモリの許す限り、好きなだけ変数を用意して計算をすることができます。しかし、変数が数百にもなると、もはや人力で変数を管理するのは困難です。また重複しないように変数名を考えることすらできません。そこで、関連した変数をひと塊にすることができます。それが配列（array）です。

3.7 ポインタと参照

3.8 文字列

3.9 new と delete

3.10 virtual

3.11 スコープ

3.12 プログラムの書き方

4　ROOT における C++

4.1　ROOT とは何か

4.2　ROOT と C++ の違い

4.3　CINT

4.4　ACLiC

4.5　ROOT 固有の部分

5　ヒストグラム

5.1　ヒストグラムとは何か

ヒストグラム（histogram、度数分布図）は、ある物理量を複数回測定したとき、測定値の分布がどのようになっているかを表すときに頻繁に使われます。身近な例では、図 5.1 に示すような人口の年齢分布などに使われます。

ヒストグラムを使うと、その測定対象がどのような値を取りやすいのかが、一目瞭然になります。図 5.1 の元データは、総務省統計局のまとめた国勢調査の結果です。コード 5.1 のような、単なる数字の羅列を見ただけでは、このデータがどのような特性を持っているのかを視覚的に認識することは大変困難です。どのような年齢層に人口が偏っているのか、東京のような都市部と鳥取のような地方では、人口分布の特徴がどうなっているのか、こういう情報はヒストグラムにして比較するのが一番です。図 5.1 と図 5.2 は、コード 5.2 で作成しました。

ヒストグラムを「読む」上で大切な点は、棒の 1 本ずつの面積が意味を持つということです。図 5.1 を見ると、0〜5 歳の人口は全国平均で約 4.5% になっています。ただし、縦軸の値は「%」ではなく「%/5 year」になっていることに注意してください。1 つの棒の幅が 5 年間分あるので、縦軸の値に 5 年間をかけて、単位が「%」になった人口の割合

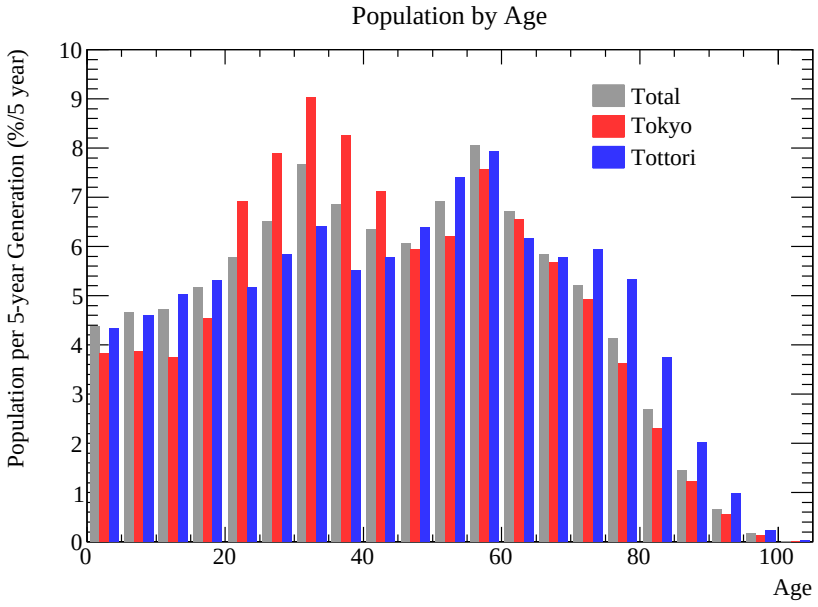

図 5.1　2005 年国勢調査を元にした、全国合計、東京都、島根県の年齢別人口分布。コード 9.3 で同じ結果を得られる。データは http://www.e-stat.go.jp/SG1/estat/List.do?bid=000001007609&cycode=0 から入手可能。

コード 5.1 population.dat

```
Total Tokyo Tottori
5578087 476692  26333
5928495 481382  27945
6014652 466593  30545
6568380 562968  32239
7350598 859742  31331
8280049 981230  35464
9754857 1121689 38890
8735781 1026016 33490
8080596 885146  35032
7725861 736656  38768
8796499 770054  44873
10255164  938669  48068
8544629 813422  37384
7432610 705944  35001
6637497 612400  36028
5262801 451357  32420
3412393 285738  22804
1849260 151770  12294
840870  68497 5951
211221  17606 1459
25353 2215  156
```

が出てくるわけです[*1][*2]。

5.1.1 折れ線グラフとの違い

ヒストグラムの用途は、ある測定値の範囲にどれだけの事象（イベント）が存在するかを図示することです。したがって、測定値には幅が存在し、特定の測定値で代表することはできません。先ほどの図 5.1 の例では、最初の棒は 0〜5 歳の人口を表していました。縦軸の値は、中心値の 2.5 歳を代表するものではないことに注意してください。

従って、図 5.1 を図 5.2 のように折れ線グラフにして表示するのは誤りです。折れ線グラフにする場合は、1 つ 1 つの点の座標がともに（誤差の範囲内で）意味のある 1 つの数値でなくてはいけません。折れ線グラフを使用するのは、原則として線分の傾きに意味がある場合に限ります。

5.1.2 ビン

図 5.1 の横軸は、0〜105 歳を 21 の区間に分けてあります。このような小分けした区間のことを、ビン（bin）と呼びます。またそれぞれのビンの幅が 5 歳分に相当し、これをビン幅（bin width）と呼びます。この例の 21 という数を、ビン数などと呼ぶことがあります。同じデータに対してビン数を変化させても、ヒストグラムの総面積は一定であることに注意してください。

[*1] 新聞などで見かける図表の多くは、縦軸の単位を省略して単純に「%」を使うことが多いですが、我々のように物理量を単位を含めて正確に扱う場面では、分母が何であるのか注意してください。

[*2] 図 5.1 では、3 つのヒストグラムを並べて表示するために棒の幅を 5 年間よりも細くしています。5 年間分の太さにするほうがより正確な表現ですが、この図では（本来の幅が常識で判断できるため）見やすさを優先してあります。

コード 5.2 population.C

```
1  void population()
2  {
3    ifstream fin("population.dat");
4
5    const Int_t kHistN = 3;
6    const Int_t kBinsN = 21;
7    TH1D* hist[kHistN];
8
9    for(Int_t i = 0; i < kHistN; i++){
10     string str;
11     fin >> str;
12     hist[i] = new TH1D(str.c_str(), str.c_str(), kBinsN, 0, 105);
13   } // i
14
15   for(Int_t j = 0; j < kBinsN; j++){
16     for(Int_t i = 0; i < kHistN; i++){
17       Int_t pop;
18       fin >> pop;
19       hist[i]->SetBinContent(j + 1, pop);
20     } // i
21   } // j
22
23   TCanvas* can1 = new TCanvas("can1", "histogram");
24
25   TH1D* frame = new TH1D("frame", "Population by Age;Age;Population per 5-year
         Generation (%/5 year)", kBinsN, 0, 105);
26   frame->SetMaximum(10);
27   frame->Draw();
28
29   TLegend* leg1 = new TLegend(0.65, 0.7, 0.85, 0.85);
30   leg1->SetFillStyle(0);
31
32   Int_t kColor[kHistN] = {kGray + 1, kRed - 4, kBlue - 4};
33
34   for(Int_t i = 0; i < kHistN; i++){
35     hist[i]->Scale(100./hist[i]->GetEffectiveEntries()); // normalize
36     hist[i]->SetLineColor(kColor[i]);
37     hist[i]->SetFillColor(kColor[i]);
38     hist[i]->SetBarWidth(0.28);
39     hist[i]->SetBarOffset(0.08 + 0.28*i);
40     hist[i]->Draw("bar same");
41     leg1->AddEntry(hist[i], hist[i]->GetTitle(), "f");
42   } // i
43   leg1->Draw();
44
45   TCanvas* can2 = new TCanvas("can2", "graph");
46   frame->Draw();
47   TGraph* graph[kHistN];
48   TLegend* leg2 = new TLegend(0.65, 0.7, 0.85, 0.85);
49   leg2->SetFillStyle(0);
```

```
50
51   for(Int_t i = 0; i < kHistN; i++){
52     graph[i] = new TGraph();
53     for(Int_t j = 0; j < kBinsN; j++){
54       graph[i]->SetPoint(j, hist[i]->GetBinCenter(j + 1), hist[i]->GetBinContent(j +
             1));
55     } // j
56     graph[i]->SetLineColor(kColor[i]);
57     graph[i]->SetMarkerColor(kColor[i]);
58     graph[i]->Draw("same");
59     leg2->AddEntry(graph[i], hist[i]->GetTitle(), "l");
60   } // i
61   leg2->Draw();
62 }
```

5.2 1 次元ヒストグラム

5.3 2 次元ヒストグラム

5.4 3 次元ヒストグラム

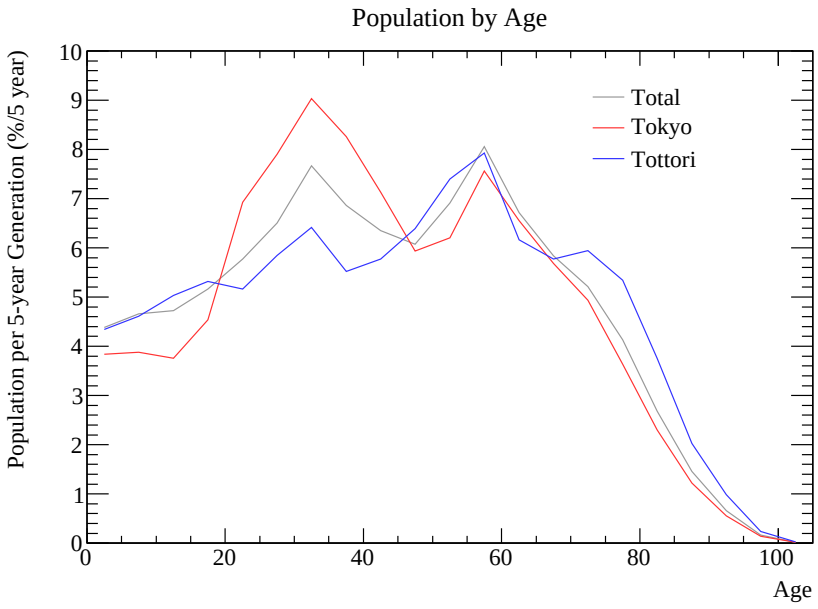

図 5.2 図 5.1 の間違った表示方法の例

6　グラフ

7 Tree

8 Python

8.1 なぜ Python を使うのか

Python とは、プログラミング言語の 1 つです。第 3 章で説明した概念は、ほぼそのまま Python でも通用します。C++ と異なる点は、例えば以下のようなものが挙げられます。

- コンパイルしなくてもコードを実行可能な、スクリプト言語と呼ばれるものです。これは ROOT の CINT に似ています。
- 様々なモジュールが標準で用意されており、C++ に比べると、手軽に様々な機能を使うことができます。例えば、オプション解析の機能が標準で利用可能です。
- 物理、天文業界で Python を利用する研究者が近年増えており、データ解析に必要な外部モジュールが多く用意されています。
- Linux/Mac/Windows といった OS 環境を気にせずに使うことができます。

特に 3 つ目は重要で、ROOT、IRAF、DS9、Geant4、FITS といったものを、Python という 1 つの言語の中で同時に扱えるようになります。もちろん、これらの機能を C++ から呼び出してコンパイルすることも可能です。しかし、リンクすべきライブラリやヘッダーファイルを把握し、どのような環境でも確実に動作するプログラムを組むのは大変なことです。Python であれば、OS を意識せずに様々な機能を簡単に使うことができます。

例えば、エネルギー、座標、時間などで構成される光子イベントが FITS のバイナリテーブルで用意されているとしましょう。このイベントのエネルギー分布を ROOT のヒストグラムに詰めたいと思った場合、以下のような簡単なコードで作業が終了します。もしあなたがこのコードを見て、短くて簡単だと感じるならば、ぜひ Python に挑戦してみましょう。

```
>>> import ROOT
>>> import pyfits
>>> hist = ROOT.TH1D("hist", "Energy distribution;Energy (MeV)", 100, 0, 1e3)
>>> energies = pyfits.open("event_list.fits")[0].data.field("ENERGY")
>>> for i in range(energies.size):
...     hist.Fill(energies[i])
>>> hist.Draw()
```

8.2 Python のインストール

8.3 追加しておきたいモジュール

Python のモジュールの追加方法は簡単です。なぜなら、追加方法が標準的なものに統一されているからです。もし foo というモジュールがあったとします。次のように、ダウンロードしてきたファイルのディレクトリに移動して、含まれる `setup.py` を引数つきで実行するだけです。

```
$ tar zxvf foo-1.2.3.tar.gz
$ cd foo-1.2.3
$ sudo python setup.py install
```

この setup.py は configure スクリプトや Makefile のようなものです。普通のモジュールには、必ず含まれています。

- PyROOT
- NumPy
  http://numpy.scipy.org/
- PyFITS
  http://www.stsci.edu/resources/software_hardware/pyfits
- python-sao
  http://code.google.com/p/python-sao/
- coords
  https://www.stsci.edu/trac/ssb/astrolib/
- pywcs
  https://www.stsci.edu/trac/ssb/astrolib/

8.4 Python の基本

8.5 PyROOT

8.5.1 C++ から Python へ

8.5.2 メモリ管理

8.6 PyFITS

9 様々な技

ROOT でそこそこ格好良い図を作るには、ある程度の知識と慣れが必要になります。ここでは、いくつかのスクリプトとその出力結果を例示し、ROOT で望み通りの図を作るにはどうすれば良いかを紹介します。

9.1 色関連

9.1.1 自前のカラーパレットを定義する

ROOT では、2 次元ヒストグラムや 2 次元グラフの「高さ」を表現する手段として、色を用いることができます。これは第 5 章でも説明しました。ROOT はいくつかのカラーパレット（color palette）を用意してくれていますが、それらの実用性は乏しいと言わざるを得ません。例えば図 9.1 の左上に示したような、デフォルトのカラーパレットを使っている人はほとんど見かけません。また階調数が小さめに設定されているため、滑らかな色表現には向きません。唯一よく使われているのが、以下の設定です。

```
root [0] gStyle->SetPalette(1)
```

レインボーカラー（rainbow color）などと呼ばれることがあります。他にデフォルトで用意されているパレットについては、http://root.cern.ch/root/html/TColor.html#TColor:SetPalette を参照してください。

コード 9.1 では、自分好みのカラーパレットを作る方法を示しています。原理は単純で、作りたいパレットに応じて、TColor::CreateGradientColorTable を呼び出すための関数を用意するだけです。いくつか例を書きましたが、原理は一緒なので関数 BPalette() の解説のみをします。

コード 9.1 color_def.C

```
1  void BPalette()
2  {
3    static const Int_t kN = 100;
4    static Int_t colors[kN];
5    static Bool_t initialized = kFALSE;
6
7    Double_t r[]    = {0., 0.0, 1.0, 1.0, 1.0};
8    Double_t g[]    = {0., 0.0, 0.0, 1.0, 1.0};
9    Double_t b[]    = {0., 1.0, 0.0, 0.0, 1.0};
10   Double_t stop[] = {0., .25, .50, .75, 1.0};
11
12   if(!initialized){
13     Int_t index = TColor::CreateGradientColorTable(5, stop, r, g, b, kN);
14     for (int i = 0; i < kN; i++) {
15       colors[i] = index + i;
16     } // i
17     initialized = kTRUE;
```

```
18    } else {
19      gStyle->SetPalette(kN, colors);
20    } // if
21  }
22
23  void GrayPalette()
24  {
25    static const Int_t kN = 100;
26    static Int_t colors[kN];
27    static Bool_t initialized = kFALSE;
28
29    Double_t r[]    = {0., 1.};
30    Double_t g[]    = {0., 1.};
31    Double_t b[]    = {0., 1.};
32    Double_t stop[] = {0., 1.};
33
34    if(!initialized){
35      Int_t index = TColor::CreateGradientColorTable(2, stop, r, g, b, kN);
36      for (int i = 0; i < kN; i++) {
37        colors[i] = index + i;
38      } // i
39      initialized = kTRUE;
40    } else {
41      gStyle->SetPalette(kN, colors);
42    } // if
43  }
44
45  void GrayInvPalette()
46  {
47    static const Int_t kN = 100;
48    static Int_t colors[kN];
49    static Bool_t initialized = kFALSE;
50
51    Double_t r[]    = {1., 0.};
52    Double_t g[]    = {1., 0.};
53    Double_t b[]    = {1., 0.};
54    Double_t stop[] = {0., 1.};
55
56    if(!initialized){
57      Int_t index = TColor::CreateGradientColorTable(2, stop, r, g, b, kN);
58      for (int i = 0; i < kN; i++) {
59        colors[i] = index + i;
60      } // i
61      initialized = kTRUE;
62    } else {
63      gStyle->SetPalette(kN, colors);
64    } // if
65  }
66
67  void RBPalette()
68  {
```

```
69    static const Int_t kN = 100;
70    static Int_t colors[kN];
71    static Bool_t initialized = kFALSE;
72
73    Double_t r[]    = {0., 1., 1.};
74    Double_t g[]    = {0., 1., 0.};
75    Double_t b[]    = {1., 1., 0.};
76    Double_t stop[] = {0., .5, 1.};
77    if(!initialized){
78      Int_t index = TColor::CreateGradientColorTable(3, stop, r, g, b, kN);
79      for (int i = 0; i < kN; i++) {
80        colors[i] = index + i;
81      } // i
82      initialized = kTRUE;
83    } else {
84      gStyle->SetPalette(kN, colors);
85    } // if
86  }
```

次の 4 行が、実際に色の設定をする部分です。

```
    Double_t r[] = {0., 0.0, 1.0, 1.0, 1.0};
    Double_t g[] = {0., 0.0, 0.0, 1.0, 1.0};
    Double_t b[] = {0., 1.0, 0.0, 0.0, 1.0};
    Double_t stop[] = {0., .25, .50, .75, 1.0};
```

最初の 3 行で RGB 各色の輝度情報を設定します。例えば R = G = B = 1 であれば白、R = G = B = 0 であれば黒、R = G = 1、B = 0 であれば黄色といった具合です。次の行は、それらの色がヒストグラムの最小値（≡ 0）から最大値（≡= 1）のどこに相当するかを決めています。

グラデーションを ROOT に登録する作業は、

```
    Int_t index = TColor::CreateGradientColorTable(5, stop, r, g, b, kN);
```

で行います。最後の引数は、階調の数です。これを大きくすればより滑らかなグラデーションになります。これらの関数が複数回呼び出されても速度低下を招かないように、関数内静的変数を用いていることに注意してください。

ガンマ線のカウントマップでは、よくこの `BPalette()` が使われます[*1]（図 9.1 下段）。明るいところを強調し、暗いノイジーな箇所を目立たなくするためでしょう[*2]。`GrayPalette()` と `GrayInvPalette()` は、それぞれ黒から白、白から黒へのグラデーションです（図 9.1 中段）。`RBPalette()` は、青、白、赤と変化するグラデーションです。世の中であまり使われていませんが、2 次元ヒストグラムの残差を見せるときなどに筆者は使っています。

実際に作成するスクリプトでこのようなパレットを設定するためには、どこかでこれらの関数を定義しておいて、ヒストグラムを描く前に呼び出して下さい。例えば

```
root [0] .L color_def.C
root [1] BPalette()
```

などとすれば良いでしょう。 /.rootlogon.C に

```
gROOT->LoadMacro("color_def.C");
```

[*1] `BPalette` という名前は、DS9 のパレットの名前に基づいています。

[*2] カラーパレットの使い方で、（良い意味でも悪い意味でも）図の印象ががらりと変わるということを心に留めておいてください。

という 1 行を加えて、起動時に読み込ませておくのでも大丈夫です。

9.1.2 複数のカラーパレットを同時に使う

自分の好きなようにカラーパレットを作成しても、複数のカラーパレットを同時に使うためには小技が必要です。例えば以下を実行した後に、can1 をクリックしてみてください。

```
root [0] TH2D* h2 = new TH2D("h2", "", 3, -1, 1, 3, -1, 1)
root [1] h2->Fill(0, 0)
root [2] TCanvas* can1 = new TCanvas("can1", "can1")
root [3] gStyle->SetPalette(1)
root [4] h2->Draw("colz")
root [5] TCanvas* can2 = new TCanvas("can2", "can2")
root [6] BPalette()
root [7] h2->Draw("colz")
```

クリックする直前まではレインボーパレットだったのに、クリックすると BPalette() の設定に変わってしまうはずです。これは、ROOT がクリックを検知した後に再描画を開始するためですが、その時点でグローバルに持っているパレットの情報が BPalette() に書き換えられてしまっているからです。これを回避するのが、コード 9.2 です。TExec を「重ね塗り」することによって、再描画の直前にパレットの設定を強制的に実行することができます。

コード 9.2 multi_palette.C

```
1  void multi_palette()
2  {
3    gStyle->SetOptStat(0);
4
5    TCanvas* can = new TCanvas("can", "can", 400, 600);
6    can->Divide(2, 3, 1e-10, 1e-10);
7
8    TH2D* hist  = new TH2D("hist", "", 50, -5, 5, 50, -5, 5);
9    hist->SetContour(100);
10
11   for(int i = 0; i < 100000; i++){
12     double x = gRandom->Gaus();
13     double y = gRandom->Gaus();
14     hist->Fill(x, y);
15   } // i
16
17   gROOT->ProcessLine(".L color_def.C");
18
19   TExec* exe[6];
20
21   exe[0] = new TExec("ex0", "gStyle->SetPalette(0);");
22   exe[1] = new TExec("ex1", "gStyle->SetPalette(1);");
23   exe[2] = new TExec("ex2", "GrayPalette();");
24   exe[3] = new TExec("ex3", "GrayInvPalette();");
25   exe[4] = new TExec("ex4", "BPalette();");
26   exe[5] = new TExec("ex5", "RBPalette();");
27
28   for(int i = 0; i < 6; i++){
29     gPad = can->cd(i + 1);
```

```
30      hist->Draw("axis z");
31      exe[i]->Draw();
32      hist->Draw("same colz");
33      gPad->Update();
34    } // i
35  }
```

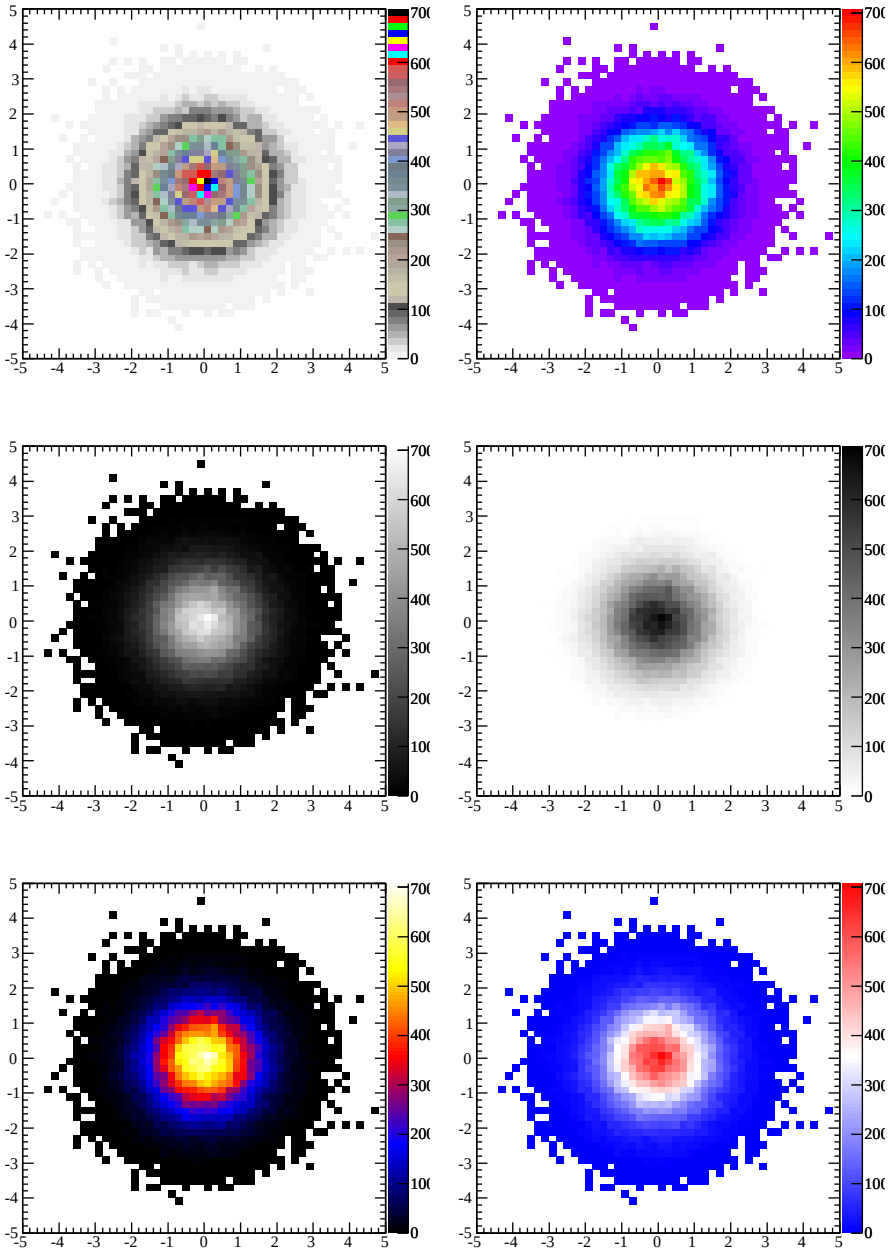

図 9.1 コード 9.2 の出力結果

9.1.3 塗りつぶしの色を変更する

図 9.1 で様々なパレットを例示しました。既にお気付きの通り、値が 0 のビンは ROOT は塗りつぶしません。そのため、空っぽのビンは全て白くなっています。図 9.1 の右上のように、パレットに白が含まれていない場合は、白いままのほうがヒストグラムの状態を把握するのに便利です。しかしパレット中に白が含まれている場合は、値が 0 なのか、他の値なのか判断できない場合が出てきます。そのような場合は、フレームの色を変更しましょう。

コード 9.3 frame_fill_color.C

```
1  void frame_fill_color()
2  {
3    gStyle->SetOptStat(0);
4
5    TCanvas* can = new TCanvas("can", "can", 400, 400);
6
7    TH2D* hist  = new TH2D("hist", "", 50, -5, 5, 50, -5, 5);
8    hist->SetContour(100);
9    hist->GetXaxis()->SetAxisColor(0);
10   hist->GetYaxis()->SetAxisColor(0);
11
12   for(int i = 0; i < 100000; i++){
13     double x = gRandom->Gaus();
14     double y = gRandom->Gaus();
15     hist->Fill(x, y);
16   } // i
17
18   gROOT->ProcessLine(".L color_def.C");
19   BPalette();
20   hist->Draw("colz");
21   gPad->Update();
22   TPaletteAxis* palette
23     = (TPaletteAxis*)hist->GetListOfFunctions()->FindObject("palette");
24   Int_t col = palette->GetValueColor(hist->GetMinimum());
25   hist->Draw("colz");
26   gPad->SetFrameFillColor(col);
27   gPad->Update();
28 }
```

コード 9.3 は、フレームの背景色を変更する方法です。出力結果は図 9.2 に示します。

```
  TPaletteAxis* palette
    = (TPaletteAxis*)hist->GetListOfFunctions()->FindObject("palette");
```

この部分では、ヒストグラムから TPaletteAxis[*3]のポインタを取得します。その直前の行で gPad を更新しないと 0 を返すので注意してください。

```
  gPad->SetFrameFillColor(col);
```

で、gPad の塗りつぶしの色を決定しています。この例では、最小値の色は黒になっています。

[*3] 図 9.2 の右端にあるグラデーション付き目盛りのことです。

9.2 キャンバス関連

9.2.1 描画領域の大きさを指定する

通常、新たなキャンバスを作成するときに大きさを指定する場合は、

```
root [0] TCanvas* can = new TCanvas("can", "can", 100, 100)
```

のように第 3、第 4 引数に横幅と高さを指定します。しかし予想外にも、図 9.3a に示すように、実はこの大きさは描画領域の大きさではありません。メニューバーは描画領域外周部まで含めた大きさなのです。したがって、実際に描画できる部分の大きさは、筆者の環境では 96×72 ピクセルになってしまいます。図 9.3b のように、描画領域を正確に 100×100 ピクセルにしたい場合は、コード 9.4 のような関数を用意しましょう。

コード 9.4 PreciseSizeCanvas.C

```
1  TCanvas* PreciseSizeCanvas(const char* name, const char* title,
2                             Double_t width, Double_t height)
3  {
4    TCanvas* can = new TCanvas(name, title, width, height);
5    can->SetWindowSize(width + (width - can->GetWw()),
6                       height+ (height- can->GetWh()));
7    gSystem->ProcessEvents();
8
9    return can;
10 }
```

次のように、PresiceSizeCanvas.C をロードしてから、PresiceSizeCanvas 関数を TCanvas のコンストラクタのように使用すれば、描画領域が丁度 400×400 ピクセルのキャンバスが得られます。

```
root [0] .L PresiceSizeCanvas.C
root [1] TCanvas* can = new PresiceSizeCanvas("can", "can", 400, 400)
```

もし、得られた描画領域が 400×400 でない場合は、お使いのコンピュータの画面の高さが 1000 ピクセル未満の可能性があります[*4]。 /.rootrc の

```
Canvas.UseScreenFactor:      true
```

という行を

```
Canvas.UseScreenFactor:      false
```

に変更するか、

```
root [0] gStyle->SetScreenFactor(1.)
```

を実行して再度試してみてください。

またコンピュータの処理速度によっては、TCanvas::SetWindowSize の結果が反映されるまでに次の処理が開始される場合があります。例えば

```
can->SaveAs("foo.png");
can->SaveAs("bar.png");
```

[*4] 画面が小さすぎると ROOT が勝手に判断し、TCanvas の大きさを自動調整するためです。

のように 2 回連続で PNG 画像を保存させると、2 つの画像のサイズがウインドウサイズ変更前と変更後になる場合があります。そのようなときは、

```
gSystem->Sleep(100);
```

のような行を足すことで、処理待ちをさせることも可能です。

9.3 グラフ関連

9.3.1 残差を表示する

コード 9.5 residual.C

```
1  void residual()
2  {
3    TGraphErrors* gra = new TGraphErrors();
4    gra->SetTitle(";Time (s);Voltage (V)");
5    for(int i = 0; i < 20; i++){
6      double x = i + 0.5;
7      double y = 5*sin(x);
8      double ey = gRandom->Gaus();
9      gra->SetPoint(i, x, y + ey);
10     gra->SetPointError(i, 0, 1);
11   } // i
12
13   TCanvas* can = new TCanvas("can", "can");
14   can->cd(1);
15   can->SetBottomMargin(0.3);
16
17   TF1* f1 = new TF1("f1", "[0] + [1]*sin(x)", 0, 20);
18   f1->SetParameter(0, 0.);
19   f1->SetParameter(1, 0.9);
20   gra->Fit("f1");
21   gra->GetXaxis()->SetLabelSize(0);
22   gra->GetXaxis()->SetTitleSize(0);
23   gra->GetXaxis()->SetLimits(0, 20);
24   gra->GetHistogram()->SetMaximum(9.);
25   gra->GetHistogram()->SetMinimum(-9.);
26   gra->Draw("ape");
27
28   TPad* pad = new TPad("pad", "pad", 0., 0., 1., 1.);
29   pad->SetTopMargin(0.7);
30   pad->SetFillColor(0);
31   pad->SetFillStyle(0);
32   pad->Draw();
33
34   TGraphErrors* res = new TGraphErrors();
35   res->SetTitle(";Time (s);#chi");
36
37   for(int i = 0; i < 20; i++){
38     double x = gra->GetX()[i];
```

```
39      double y = gra->GetY()[i];
40      double ey = gra->GetErrorY(i);
41      res->SetPoint(i, x, (y - f1->Eval(x))/ey);
42      res->SetPointError(i, 0, 1);
43    } // i
44
45    pad->cd(0);
46    res->GetXaxis()->SetLimits(0, 20);
47    res->GetHistogram()->SetMaximum(3.5);
48    res->GetHistogram()->SetMinimum(-3.5);
49    res->GetYaxis()->CenterTitle();
50    res->GetYaxis()->SetNdivisions(110);
51    res->Draw("ape");
52  }
```

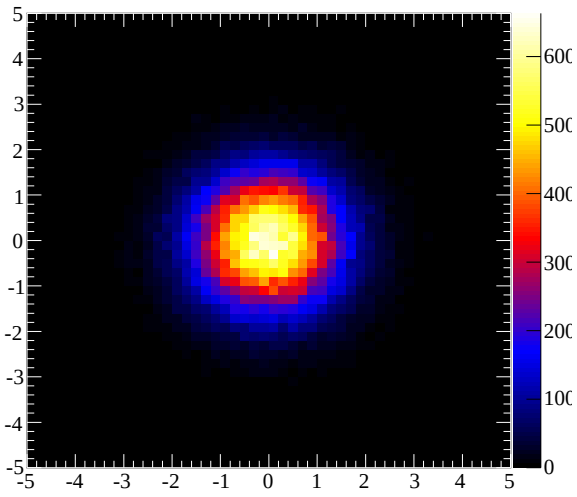

図 9.2 コード 9.3 の出力結果

(a) 96 × 72 ピクセル (b) 100 × 100 ピクセル

図 9.3 TCanvas の描画領域の違い

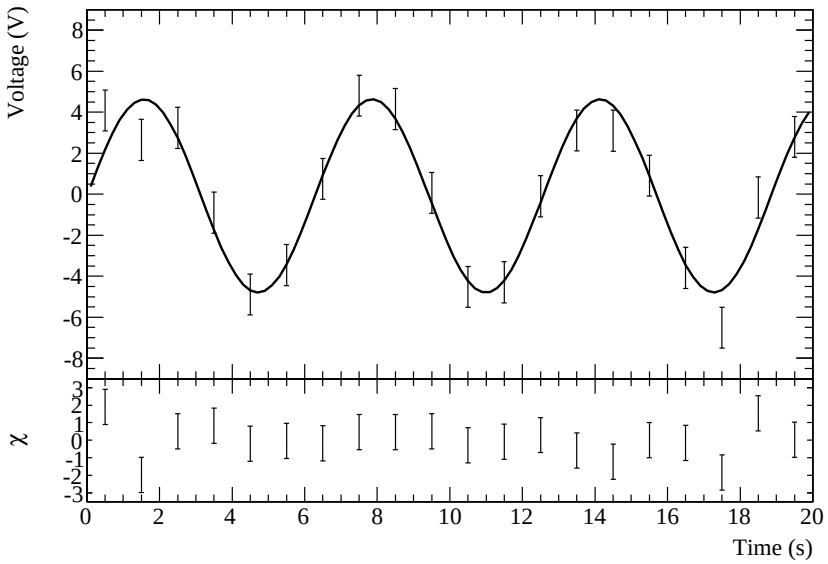

図 9.4 コード 9.5 の出力結果

付録 A　Mac での研究環境の構築

ここでは、Mac で研究環境を構築する祭に最低限やるべきこと、知っておくべきことを説明します。Mac に限らず Windows や Linux でも、計算機環境の設定は個々人の好みや研究内容によって大きく変わります。そのため、ここに書くことは参考程度にとどめて下さい。

A.1　英語環境にする

日本の研究室で Mac を使う場合、OS の言語環境を日本語にしている人が多いでしょう。しかし Mac を研究で使う場合には英語環境に変更することを強くお勧めします。大きく四つの理由があるからです。

まず第一に、インターネット上の Mac 関係の情報の多くが英語で書かれており、英語で検索した場合に見つかる情報の量と質は日本語の情報を圧倒するからです。使っている Mac で何か問題が生じた場合にエラーメッセージが日本語で表示されていると、それを検索語にしても辿り着ける情報には限りがあります[*1]。Apple 社はアメリカの企業であり Mac ユーザの多くが北米に集中しています。そのため Mac 関連の情報のやり取りの多くは英語でなされています。これは Mac に限らずコンピュータ関係全般に言えます[*2]

第二に、あなたは日本人のみと共同研究をするわけではないからです。もしあなたの Mac の画面を外国人が見ながら、もしくはあなたが外国人の Mac の画面を見ながら作業する時に、OS が英語環境になっていたほうが意志疎通が簡単になることは言うまでもないでしょう。また海外（特にアメリカ）の研究機関などで実験をするときに、現地で使用する Mac が英語環境になっているのは当然です。

第三の理由は、ファイル名やメニューの表示が英語になっているほうが作業効率が上がるからです。Mac だとホームディレクトリに「書類」や「デスクトップ」というディレクトリが存在しますが、英語環境ではそれぞれ「Documents」と「Desktop」です。Terminal.app からホームで `ls` すれば、実体が英語名だということがわかります。Finder.app からディレクトリの移動をしたいときに、英語環境であればホームを開いた状態で「do」と連打すれば「Documents」が選択された状態になります。また「de」と打てば「Desktop」が選択されます。日本語環境の場合にはこのようにはいきません。またメニューの「編集」は英語環境では「Edit」になっています。メニューで「Edit」を開いた状態で「co」と連打すれば「Copy」のところが選択された状態になるでしょう。

最後の理由は、できる限り英語に慣れ親しんだほうが良いからです。大学院に入りたての頃は、誰しも英語の読み書きと会話に苦労するでしょう。また日本人の多くの研究者はその後何十年も英語で苦労をし続けます。少しでも英語に慣れるため、常用する Mac くらい英語環境で使う意志を持ちましょう。

ただし、英語環境にすることで問題が生じる場合があります。例えば英語環境で FLASH を表示すると日本語が文字化けすることがあります。これは FLASH が既に廃れつつある技術であり、かつ FLASH の開発チームが能力不足だからです。他にも英語環境にすると日本語表示がうまくいかないソフトがあるかもしれませんが、そのようなソフトを使うのはやめましょう。日本語表示以外にも色々と問題を抱えている可能性があります。

[*1] LaTeX 関係などの情報だと日本語特有の問題も発生しうるので、そこは臨機応変に対応して下さい。

[*2] 恐らく唯一の例外が Ruby というプログラミング言語です。これは日本人により開発されたものが世界に広がった希有な例です。

図 A.1　「システム環境設定」から「言語とテキスト」を開く

図 A.1 から図 A.3 のように「システム環境設定」から英語環境に変更することができます。この設定後に起動したアプリケーションは、全て英語環境として起動されます。全てのメニューなどが英語で表示されるはずです。図 A.3 にある「単語区切り」の設定は忘れないようにして下さい。ダブルクリックで日本語文字列を選択する場合に、熟語やカタカナ語が一つの単語として認識されるようになります。

A.2　拡張子を表示する

Mac の初期設定では、ファイルの拡張子 (extension) が表示されません。図 A.4 のように Finder.app の「Preferences...」から表示する設定に変更しましょう。この表示をしないと、Terminal.app から操作するファイル名と Finder.app からファイル名が見かけ上一致しない場合があります。

A.3　キーボードの設定

もしあなたの Mac が US 配列のキーボードならば、Caps Lock キーは Control キーとして機能するように設定を変更しましょう。図 A.5 のように、System Preferences から「Keyboard」を開き、Caps Lock を Control にします。また、「Keyboard Shortcuts」のタブに移動し、ボタンなどの選択を全て tab キーで行えるようにします。このようにすることで、様々な画面操作をするときに、いちいちキーボードからトラックパッドやマウスへ手の移動をしなくて済むようになります。

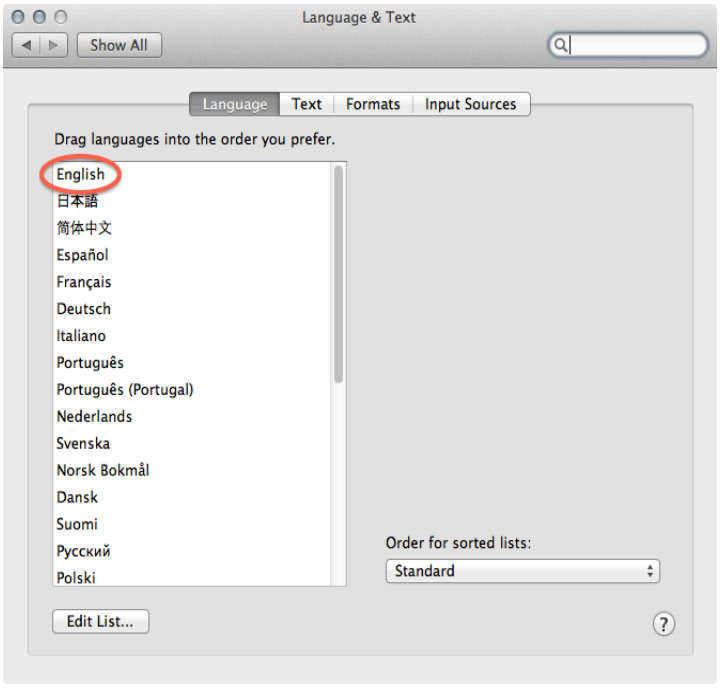

図 A.2　「日本語」ではなく「English」を先頭に持ってくる（この画面は既に英語環境になっている場合のもの）

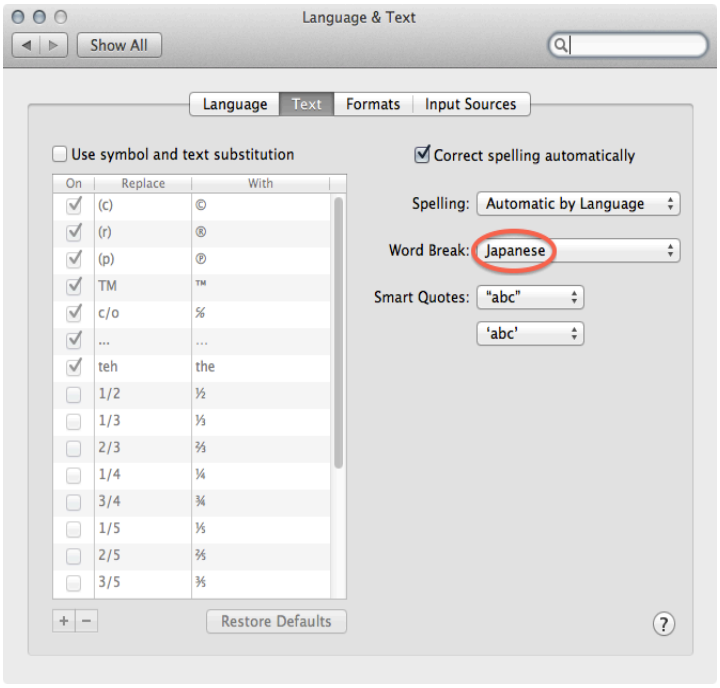

図 A.3　「単語区切り（Word Break）」を「Japanese」にする（この画面は既に英語環境になっている場合のもの）

図 A.4　「単語区切り（Word Break）」を「Japanese」にする（この画面は既に英語環境になっている場合のもの）

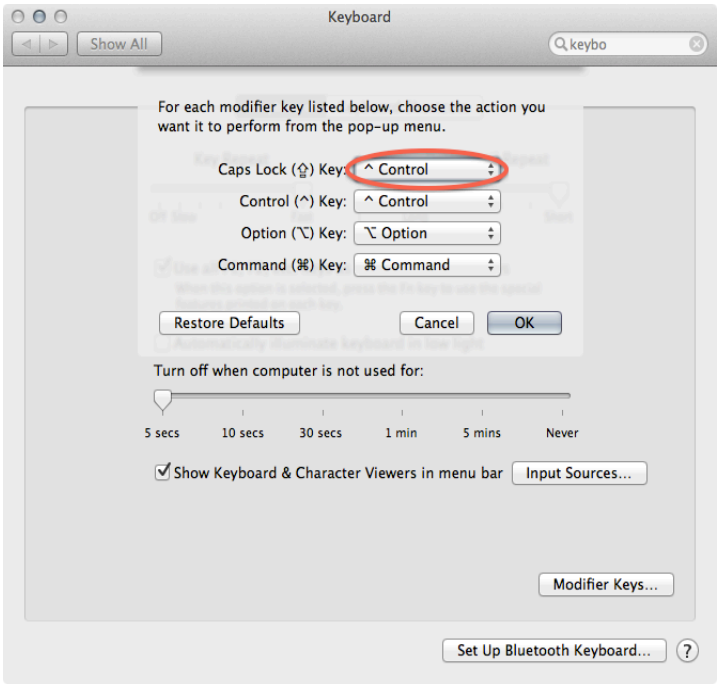

図 A.5　Caps Lock を Cntrol キーに変更する

A.4　Xcode

A.5　KeyRemap4MacBook

A.6　zsh

A.7　MacPorts

A.7.1　Emacs

A.7.2　LaTeX

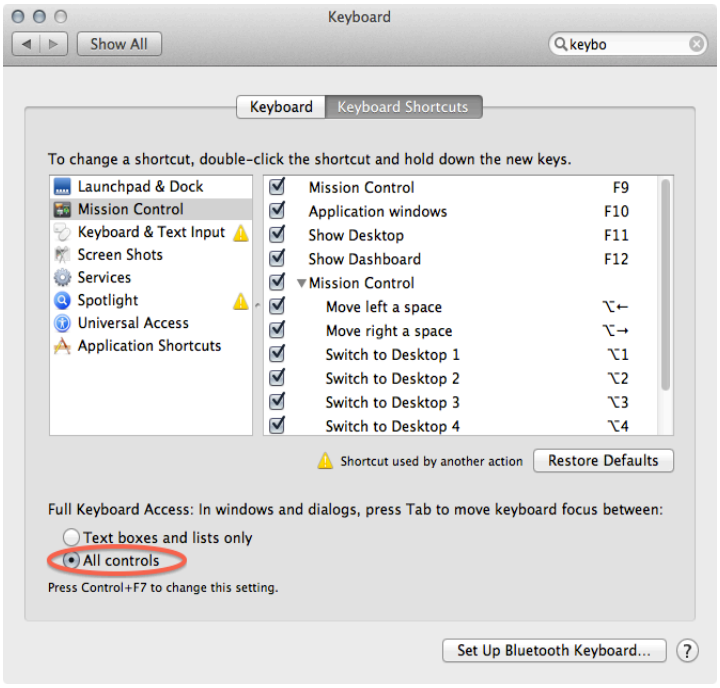

図 A.6　全てのボタンなどを tab キーで移動できるようにする