

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 暁

名古屋大学 宇宙地球環境研究所

oxon@mac.com

2019 年 4 月 23 日

本書の入手方法について

この入門書は [GitHub](https://github.com/akira-okumura/RHEA)^{*1} 上で公開されています。作成に用いた \LaTeX コード、図、また C++ のコードなどは、すべて <https://github.com/akira-okumura/RHEA> から入手可能です。

また、PDF として生成済みのものは <https://github.com/akira-okumura/RHEA/releases> で公開されています。必ずしも Git レポジトリの最新版とは対応していませんので、最新版は以下のコマンドを実行することで生成してください。git clone と make を実行すると、RHEA.pdf が作られます。make コマンド実行中に platex コマンドを使用しますので、日本語 \LaTeX 環境を用意してください (付録 A と B を参照)。

```
$ git clone https://github.com/akira-okumura/RHEA.git
$ cd RHEA
$ ls
Makefile  RHEA.bib  RHEA.tex  misc      tex
README.md RHEA.bst   fig       src
$ make
$ ls
Makefile  RHEA.bbl  RHEA.bst  RHEA.out  RHEA.toc  src
README.md RHEA.bib  RHEA.dvi  RHEA.pdf  fig       tex
RHEA.aux  RHEA.blg  RHEA.log  RHEA.tex  misc
$ open RHEA.pdf # macOS の場合
$ xdg-open RHEA.pdf # Linux の場合
```

本書の再配布について

PDF での再配布を禁止します。これは以下の理由のためです。

- 最新版でない PDF の流布を避け、ROOT などに関する情報をなるべく新しいもので伝えるため
- 読者に git や make コマンドに慣れてもらうため
- 注意書きや著作権表記などの変更が生じうるため

本書の著作権について

奥村暁が著作権を保持します。GitHub 上で \LaTeX コード等を公開していますが、これは著作権の放棄を意味しません。GitHub 上でのレポジトリの複製・改変は自由ですが、改変したものの GitHub 以外での再配布や、改変した内容で生成した PDF の再配布を禁止します。

間違いの指摘や改善の要望

多くの間違いや説明の足りない箇所が本書には含まれていると思います。その場合、ぜひ GitHub から [issue report](#)^{*2} を作成するか、同じく GitHub で pull request を出すか、もしくはメールなどでご連絡いただくと助かります。

© 2009–2019 Akira Okumura All Rights Reserved

*1 <https://github.com/>

*2 <https://github.com/akira-okumura/RHEA/issues>

著者について

奥村 暁（名古屋大学 宇宙地球環境研究所 講師）

1993 年 中学 1 年生で初めての Mac (LC475) を購入し、HyperCard で遊び始める

2004 年 ROOT の前身である PAW の利用を開始

2005 年 PAW の限界と時代遅れであることに気が付き ROOT へ移行

2007 年 ROOT を使った光線追跡ライブラリ ROBAST の開発を開始し、2010 年に一般公開^{*3}

2009 年 東京大学大学院理学系研究科物理学専攻で博士号取得

2012 年 名古屋大学 太陽地球環境研究所 助教

2015 年 名古屋大学 宇宙地球環境研究所 助教（改組による）

2018 年 名古屋大学 宇宙地球環境研究所 講師

2019 年現在、Cherenkov Telescope Array の小口径望遠鏡の開発で、焦点面カメラのデータ収集・カメラ制御ソフトウェア開発責任者などを担当している。専門は宇宙線物理学、ガンマ線天文学、光学系設計。2016 年から「宇宙素粒子若手の会」と連携し、「ROOT 講習会」を開催している。

電子メール oxon@mac.com

GitHub [@akira-okumura](#)

blog <http://oxon.hatenablog.com/>

Twitter [AkiraOkumura](#)

Flickr [akiraokumura](#)

個人ページ <https://www.isee.nagoya-u.ac.jp/~okumura/>

宇宙線物理学研究室 (CR 研) <https://www.isee.nagoya-u.ac.jp/CR/>

^{*3} <https://robast.github.io/>

目次

1	はじめに	1
1.1	ROOT とは	1
1.2	ROOT と C++	1
1.3	Python	1
1.4	高エネルギー宇宙物理学での ROOT の利用例	2
1.5	本書の目的	2
1.6	ROOT の学習方法	2
2	ROOT を始めよう	5
2.1	ROOT のインストールと初期設定	5
2.2	ROOT の操作に慣れる	14
2.3	やっておきたい初期設定	19
3	C++ の基礎	22
3.1	Hello World!	23
3.2	型と関数	25
3.3	if 文と関係演算子	27
3.4	for 文	28
3.5	クラス	31
3.6	配列	40
3.7	ポインタと参照	41
3.8	文字列	41
3.9	new と delete	41
3.10	virtual	41
3.11	スコープ	41
3.12	プログラムの書き方	41
4	ROOT における C++	42
4.1	ROOT とは何か	42
4.2	ROOT と C++ の違い	42
4.3	CINT	42
4.4	ACLiC	42
4.5	ROOT 固有の部分	42

5	ヒストグラム	43
5.1	ヒストグラムとは何か	43
5.2	1次元ヒストグラム	47
5.3	2次元ヒストグラム	48
5.4	3次元ヒストグラム	48
6	グラフ	50
7	Tree	51
8	Python	52
8.1	注意	52
8.2	なぜ Python を使うのか	52
8.3	簡単に試してみる	53
8.4	Python 3 のインストール	54
8.5	pip	55
8.6	IPython	56
8.7	追加しておきたいモジュール	56
8.8	Python の基本	57
8.9	PyROOT	57
8.10	PyFITS	57
9	様々な技	58
9.1	色関連	58
9.2	キャンバス関連	64
9.3	グラフ関連	65
10	統計学の基礎	68
10.1	参考書	68
10.2	基本的な用語	69
10.3	様々な確率分布	73
付録 A	パッケージ管理システム	75
A.1	Homebrew	75
A.2	yum	76
付録 B	L ^A T _E X 環境の構築	78
B.1	macOS での MacTeX のインストール	78
B.2	CentOS 7 での TeXLive 2018 のインストール	79
B.3	動作確認	81
付録 C	Mac での研究環境の構築	82
C.1	英語環境にする	82
C.2	拡張子を表示する	84
C.3	キーボードの設定	84

C.4	KeyRemap4MacBook	86
C.5	zsh	87
C.6	MacPorts	89
C.7	Emacs の操作体系に慣れる	93
C.8	C-Space C-w を macOS で使う	95
C.9	Xcode	96
C.10	知っている便利な macOS 特有のコマンド	96
付録 D	configure と Make	97
D.1	configure を使った ROOT のコンパイル (非推奨)	97
D.2	configure と Make の一般的な解説	98
付録 E	本文で登場した ROOT スクリプトの PyROOT 版	101
参考文献		103

1 はじめに

1.1 ROOT とは

ROOT (ルート) とは、CERN (欧州原子核研究機構、セルン、サーン) 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*1[1]。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。2019 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 ROOT は不可欠な存在になっています。

1.2 ROOT と C++

ROOT は C++ (シープラスプラス)*2 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 (フォートラン) 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、2019 年現在、高エネルギー物理学業界での標準的な言語のひとつになっています。

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 Python

また近年では C++ に代わって Python (パイソン) という言語を使用する研究者も増えてきました。できるかぎり高速な処理を行いたい場合やハードウェアの制御を行う場合を除いて、C や C++ よりも Python を選択する機会が多々あります。ROOT は Python の中からでも呼び出すことができるため、本書でも Python の簡単な解説を第 8 章で行います。

ただし、ROOT はもともと C++ の枠内で使うことを想定しているため、ROOT のマニュアルや C++ で書かれた例を読んでも、どのように Python に焼き直せば良いのか分からない場合があります。Python のみを学んで ROOT を使うことも可能ではありますが、C++ の基礎を学んでおいたほうが、より ROOT に親しめるでしょう。

研究グループや分野によっては ROOT の使用を徐々にやめ、matplotlib のような Python の解析環境に移行している

*1 <https://root.cern.ch/>

*2 日本語では「シーブラ」「シーブラブラ」と呼ばれることが多い。

ところもあります。Python ではオープンソース (open source) の開発コミュニティが ROOT よりも活発なため、より簡単に、より様々な機能を使うことができます。日本では ROOT を使い続けている研究グループが多いようですが、ROOT が解析手段の絶対的存在ではないということを知っておきましょう。

1.4 高エネルギー宇宙物理学での ROOT の利用例

To be written...

1.5 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 L^AT_EX ファイルや例題として登場する C++ のコードなどは、全て <https://github.com/akira-okumura/RHEA/tree/master/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 <https://github.com/akira-okumura/RHEA/releases> から PDF で入手可能です。

1.6 ROOT の学習方法

物理の勉強方法は個人で様々なように、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。

1.6.1 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 User's Guide』です。 <http://root.cern.ch/drupal/content/users-guide> から PDF が入手可能です。600 ページ超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が、どちらともかなり古いです。

- 『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。

1.6.2 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が、しかし、そのような苦勞をして、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。

1.6.3 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なりません。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なりません。

ソースコードを読むには、ROOT のリファレンスガイドを <https://root.cern.ch/doc/master/annotated.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が。

1.6.4 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が。

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 の公式フォーラム (<https://root-forum.cern.ch>) を利用することもできます。かつてはメーリングリスト (<https://groups.cern.ch/group/roottalk/default.aspx>) も使用されていましたが、今は公式フォーラムの利用が推奨されています。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、<https://root.cern.ch/guidelines-submitting-bug> を熟読しましょう。

1.6.5 C++ の勉強をする

既にかきましたか、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 に関するのではなく 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ず。

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（していた）オペレーションシステム（operation system、OS）は多岐にわたりますが、2019 年現在、Mac^{*1}か Linux^{*2}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 Mac か Linux の使用を前提とします^{*3}。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コンパイラ（compiler）がインストールされている必要があります^{*4}。

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が、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 <https://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。

2019 年 4 月時点で、ROOT 6 の最新安定バージョン 6.16/00 です。バージョン 5 からバージョン 6 に変わるとき、大きな変更がありました。そのため、研究室や実験グループによってはバージョン 5 系列を使っている人もまだまだ多いと思います。ROOT 5 ではバージョン 5.34/36 が最新です^{*5}。最初の数字はメジャーバージョン、次の数字は大きな機能追加ごとに増えるマイナーバージョン、スラッシュの後ろの数字はバグフィックスを示す番号です。

どのバージョンを選択するかは研究室の計算機環境や実験グループの解析ソフトなどに依存しますが、本書では 6.16/00 を前提として話を進めます。特に研究室でバージョン指定もなく初めて ROOT を使用する場合は、最新

*1 Apple 社の製造するコンピュータ Macintosh（通称 Mac）で使われている OS はバージョン 10.0 以降、永らく Mac OS X（マックオーエステン）と呼ばれてきました。しかしそのバージョンが 10.12 になってからは macOS と名称が変更されています。「Mac OS X」と「macOS」という二つの言葉が世の中では混在していますが、同じものを指すと考えてください。

*2 Redhat Enterprise Linux（RHEL）、Fedora、CentOS、Ubuntu、Scientific Linux など、一口に Linux と言っても様々な種類があります。これは Linux という言葉が OS の中核部分であるカーネル（kernel）を指すものであり、この Linux カーネルに様々なソフトウェアやライブラリを追加してひとまとめた配布形態（distribution）が複数あるためです。例えば「CentOS は Linux カーネルを使用した Linux ディストリビューションの 1 つである」というような表現をします。

*3 Windows や Solaris も ROOT 5 でサポートされていましたが、現在はおそらくサポートされていません。

*4 GCC や Clang といったコンパイラについての説明や、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 や Clang のインストール方法が分からなければ、「GCC インストール Linux」「Clang インストール Mac」などで検索して下さい。

*5 更新は 2016 年 4 月 5 日であり、今後は開発が継続されません。

バージョン <https://root.cern.ch/downloading-root> で確認し、それを使うようにしてください。

また使用している OS の公開日より古いバージョンの ROOT はコンパイルできない場合が多々ありますので、新しい ROOT のバージョンを確認するようにしてください。

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 https://root.cern.ch/download/root_v6.16.00.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_v6.16.00.source.tar.gz` がホームディレクトリ (`~`) に保存されます*6。

2.1.3 コンパイルとは

コンパイル (compile) とは、C++ などのプログラミング言語で書かれたソースコード (source code) をコンピュータが実行できるようにする作業のことです。C++ などで書かれたプログラムは人間が理解しやすいような文法で作られているため、これをコンピュータの中の CPU (central processing unit、中央処理装置) が理解できる形態に変換してやる必要があるのです。

コンパイルはコンパイラ (compiler) と呼ばれるソフトウェアを使って行います。Linux では GCC、macOS では Clang というコンパイラが多く使われています。しかし ROOT のように大きなソフトウェアになると、どのファイルをどの順序でコンパイルすれば良いのかや、どのようなライブラリが ROOT のコンパイルに必要なのかをコンピュータごとに調べる必要が出てきます。この作業を自動的に行うのが後述する CMake や configure です。これらのソフトウェアが Makefile を作成し、これにしたがってコンパイル作業が進みます。

2.1.4 CMake を使ったコンパイル (推奨)

最近の ROOT (6.08/00 以降) では、付録 D で詳述する configure コマンドではなく cmake コマンド*7を使ってビルドすることが推奨されており、configure は既にサポート対象外です。これは Geant4 でも同様で、最近のソフトウェアの流れです。CMake はまだ完全に市民権を得たわけではありませんが、徐々にソフトウェア開発で浸透しつつあります。

yum や homebrew などのパッケージ管理ソフトウェア (付録 A 参照) のコマンドを使うか、自分で CMake のページからダウンロードし、まずは CMake を導入します。ROOT 6.16/00 では CMake がバージョン 3.6 以上である必要があります。2019 年 4 月現在、Homebrew では CMake 3.13.0 が、CentOS 7.4 の Yum では CMake 3.6.3 が入手可能です。

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*8*9。

*6 ホームディレクトリとは、ユーザごとのファイルを置くためのディレクトリのことです。例えば Linux だと、ユーザ oxon の場合は `/home/oxon` が一般的にホームディレクトリになります。macOS の場合は `/Users/oxon` になります。これらは省略して `~oxon`、`~`、`$HOME` と書かれることがあります。HOME はホームディレクトリのパス (path) を記録する環境変数であり、`$` をつけることによってその値を取り出すことができます。

*7 <https://cmake.org/>

*8 この例のように本書で先頭に `$` がついてる行は、ターミナルで入力していることを表します。実際にターミナルから入力する内容は、`$` 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。

*9 管理者権限を持っていなかったり、`sudo` コマンドを使用できない場合、一般ユーザでもインストール可能なディレクトリを選択して下さい。


```
$ cd /usr/local
$ sudo tar zxvf ~/root_v6.16.00.source.tar.gz
```

root-6.16.00 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。まずは作業用のディレクトリ*10を作成して移動し、CMake を実行します。もし管理者権限を持たない一般ユーザとしてホームディレクトリなどで作業する場合、以降 sudo はつける必要はありません。

```
$ sudo mkdir root-6.16.00/obj
$ cd root-6.16.00/obj
$ sudo cmake ..
```

ここで obj というサブディレクトリを作るのは、ビルドのやり直しをする際に CMake や Make の作成したファイルを削除するのを簡単にするためというのが主な理由です。また Python 2 と Python 3 用にビルドするディレクトリを分けたいという場合にも、このようにビルド用のサブディレクトリを作っておくと便利です。

もし Python 2 ではなく Python 3 を使用した場合、節 2.1.5 で説明する cmake コマンドのオプションをつけてください。

CentOS 7 の場合、cmake3 というコマンドが yum でインストールされるため (付録 A 参照)、コマンドを cmake3 に変更して実行してください。

```
$ sudo mkdir root-6.16.00/obj
$ cd root-6.16.00/obj
$ sudo cmake3 ..
```

cmake コマンドは指定したディレクトリ (この場合は上位ディレクトリ*11) に存在する CMakeLists.txt というファイルの中身に従って、ROOT のビルドに必要なコンパイラやライブラリがインストールされているかを調べたり、その計算機環境に適切なコンパイルオプションを指定したりする作業を行います。問題がなければ、色々とターミナルに出力された最後に次の表示が出るはずですが。

```
-- Enabled support for:  asimage astiff builtin_afterimage builtin_clang
    builtin_davix builtin_freetype builtin_ftgl builtin_gl2ps builtin_glew
    builtin_llvm builtin_lz4 builtin_openssl builtin_tbb builtin_xxhash clad cling
    cocoa cxx11 davix exceptions explicitlink fitsio gdml http imt libcxx minuit2
    opengl pch python roofit shared sqlite ssl thread tmva tmva-cpu tmva-pymva vdt
    xml
.
.
.
-- Configuring done
-- Generating done
-- Build files have been written to: /usr/local/root-6.16.00/obj
```

これで /usr/local/root-6.16.00/obj/Makefile が生成されました。このファイルは、ROOT の多数のソースコードファイルをどのような手順でコンパイルし、どのようなライブラリとリンクすれば良いかが記述されています。

次に make コマンドを実行します*12。make コマンドは Makefile の中に書かれた作業手順を逐次実行するコマン

*10 obj というディレクトリ名は好きなもので構いません。

*11 上位ディレクトリは.. で表します。

*12 make の使用経験がある人は make の実行後に make install を実行したくなるかもしれませんが、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。

ドです。

```
$ sudo make -j 8
```

複数コアの CPU を使える場合には、コアの数に応じて最後に `-j 8` のような引数をつけると、コンパイルが早くなります^{*13}。

問題がなければ、色々とターミナルに出力された最後に次のような表示が出るはずですが、エラーを出さずに 100% の表示まで進み、`hsimple.C` の処理が完了していれば正常にビルドできています^{*14}。

```
Scanning dependencies of target hist2workspace
[100%] Building CXX object roofit/histfactory/CMakeFiles/hist2workspace.dir/src/
  MakeModelAndMeasurements.cxx.o
[100%] Building CXX object roofit/histfactory/CMakeFiles/hist2workspace.dir/src/
  hist2workspace.cxx.o
[100%] Linking CXX executable ../../bin/hist2workspace
[100%] Built target hist2workspace
[100%] Built target onepcm
Scanning dependencies of target hsimple
[100%] Generating tutorials/hsimple.root

Processing hsimple.C...
hsimple  : Real Time =  0.06 seconds Cpu Time =  0.05 seconds
(TFile *) 0x7f9f7cda9760
[100%] Built target hsimple
```

これまで見てきたように、CMake を使用したソフトウェアのビルドは、`cmake` コマンドを使い `CMakeLists.txt` に従った Makefile の生成、Makefile に従った `make` コマンドの実行という流れになっています。

`cmake` 実行時に何もオプションを渡さない場合、ROOT の標準機能のみがコンパイルされます。例えば発展的な機能である `minuit2` や `gdml` は「Enabled support」の一覧に入っていません。このような機能が必要となった場合、次のようにオプションをつけて実行してください。

```
$ sudo cmake .. -Dminuit2=ON -Dgdml=ON
```

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、macOS 版をダウンロードした場合の例です。コンパイル済みのものは最新版に更新されていない可能性もあるため、やはり自分でコンパイルするのが確実です。

```
$ cd ~
$ curl -O https://root.cern/download/root_v6.16.00.macosx64-10.14-clang100.tar.gz
$ cd /usr/local
$ sudo tar zxvf ~/root_v6.16.00.macosx64-10.14-clang100.tar.gz
```

コンパイル済みのバイナリは、全ての環境用に配布されているわけではありません。ROOT のダウンロードページ^{*15}から、自分の環境に対応したものをダウンロードしましょう^{*16}。

^{*13} 最近の CPU では実コア数に比べて論理コア数が多い場合があります。例えば Hyper-Threading という技術の含まれる Intel 製の CPU の場合、実コア数が 4 でも、論理コア数が 8 に見えるため、`-j 8` をつけると速度を最大化できます。

^{*14} 環境によっては `make` の処理の順番が前後する可能性があるため、必ずしもこれと完全に同じ表示ではないかもしれません。

^{*15} <https://root.cern.ch/content/release-61206>

^{*16} 2019 年 4 月現在、macOS Mojave (10.14)、macOS High Sierra (10.13)、RHEL7 クローン (CentOS Cern 7) などにバイナリ配布があります。

2.1.5 Python 2 と Python 3 の違い

CentOS 7 や macOS では、標準でインストールされている Python のバージョンは 2.7 です。このようなバージョン 2.x のものを Python 2 と呼びます。しかし 2019 年現在、多くの Python プロジェクトが Python 3 に移行しています。Python 2 と Python 3 では一部の互換性が失われており、新規に Python の学習を始める場合は Python 3 から学ぶほうが良いでしょう*17。

Python 2 がインストールされている環境、もしくは python コマンドが Python 2 を実行する環境では、ROOT は Python 2 を自動的に選択し Python 用のライブラリを構築します。そのため Python 3 を使用したい場合は、次のように明示的に Python 3 を ROOT ビルド時に指定する必要があります。

```
$ sudo cmake ../ -DPYTHON_EXECUTABLE=/usr/local/bin/python3
```

この例では、Python 3 の実行コマンドである python3 が /usr/local/bin/python3 としてインストールされていることを仮定しています。例えば Homebrew を使って macOS に Python 3 をインストールした場合がこれに該当します。

このように明示的に python3 を指定するのは、ROOT 6.16/00 では Python 2 と Python 3 の両方に対応した Python 用ライブラリを同時にビルドできないからです。そのため、Python 3 を指定した場合は Python 2 から ROOT を呼び出せませんし、その逆もまた然りです。

2.1.6 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。これは、コンピュータがあなたの設定を自動では理解してくれないからです。CMake を使ってビルドした場合は、次の 3 行を macOS の場合は ~/.bash_profile に、Linux の場合は ~/.bashrc に書き足して下さい*18*19。

```
cd /usr/local/root-6.16.00/obj
source bin/thisroot.sh
cd - > /dev/null
```

新しくターミナルを開いた場合、bash などのシェルが起動されます。この起動とともに各種設定が読み込まれ、その設定を ~/.bashrc などを書くことになっています。1 行目で ROOT をビルドしたディレクトリに移動し、2 行目でその下にあるシェルスクリプトを読み込み様々な環境変数を設定します。最後に 3 行目でもともといたディレクトリ*20に戻り、そのときに吐かれる余計な標準出力を /dev/null に渡し、ターミナルに何も表示させないようにします。

これらは、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、同じ内容を ~/.zshrc に書いて下さい。書き終わったら、

```
$ source ~/.bash_profile # macOS の場合
$ source ~/.bashrc # Linux の場合
```

*17 ただし、実験プロジェクトによっては Python 2 の使用が必須の場合もあるので先輩や教員に相談してください。

*18 もし /usr/local/root-6.16.00 以外の場所にインストールした場合は、ディレクトリを適宜変更してください。

*19 ~/.bashrc は対話的に bash を起動した際に読み込まれる設定ファイルです。一方、~/.bash_profile は login shell として bash が起動した際に読み込まれます。違いが知りたい場合は「bashrc bash_profile 違い」などで検索してください。

*20 -は前にいたディレクトリを指します。

とするか、新しいターミナルを開いてください。もし `csh` や `tcsh` を使っている場合は、`~/.cshrc` や `~/.tcshrc` に次の3行を書き足して下さい。

```
cd /usr/local/root-6.16.00/obj
source bin/thisroot.csh
cd - > /dev/null
```

そして同様に

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

とするか、新しいターミナルを開きましょう。`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。

以上の設定で、使用しているシェルの環境変数の内容が更新されます。`ROOTSYS` という変数が追加され（これは `/usr/local/root-6.16.00/obj` を指しているはずですが）、`PATH`、`LD_LIBRARY_PATH`、`PYTHONPATH` などが `ROOTSYS` 以下のディレクトリを含むように変更されています。次のコマンドの出力を確認してみましょう（**ROOT** をビルドする前から環境変数の設定がされている場合、表示が異なる可能性があります）。

```
$ echo $ROOTSYS
/usr/local/root-6.16.00/obj
$ echo $PATH
/usr/local/root-6.16.00/obj/bin
$ echo $PYTHONPATH
/usr/local/root-6.16.00/obj/lib
$ echo $LD_LIBRARY_PATH
/usr/local/root-6.16.00/obj/lib
```

2.1.7 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、ターミナルには次のように出力されます。

```
-----
| Welcome to ROOT 6.16/00                               https://root.cern /
|                                                       (c) 1995-2018, The ROOT Team |
| Built for macosx64 on Apr 03 2019, 14:43:00          |
| From tag , 23 January 2019                            |
| Try '.help', '.demo', '.license', '.credits', '.quit'/.q' |
|-----|
root [0]
```

もしも `root` というコマンドが見つからない (`command not found`) 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



図 2.1 ROOT の起動画面 (6.06/00 のもの)

```
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が、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。

Python から ROOT を使うには PyROOT が正しくビルドされている必要があります。次のように Python を起動して、ROOT モジュールを import できるか確認してください*21。

```
$ ipython
Python 2.7.10 (default, Oct 6 2017, 22:29:07)
Type "copyright", "credits" or "license" for more information.
```

*21 本書では普通の python コマンドではなく ipython を使用します。こちらのほうが色々便利であり、また Python に慣れた人の多くは ipython を使っていますので、節 8.4 を参照して ipython を導入してください。

```
IPython 5.4.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: import ROOT
```

Python 3が入っている場合には、次のようになります。環境によっては ipython3 というコマンドを代わりに実行する必要があるかもしれません。

```
$ ipython
Python 3.6.4 (default, Jan 6 2018, 11:51:59)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import ROOT
```

もしも次のようなエラーが表示された場合、PyROOT が正しくビルドされていない可能性があります。

```
In [1]: import ROOT
-----
ModuleNotFoundError                               Traceback (most recent call last)
<ipython-input-1-53ef8b0f0297> in <module>()
----> 1 import aho

ModuleNotFoundError: No module named 'ROOT'
```

特に CentOS 7 などの Linux の場合、Python 関係のヘッダーファイルなどがインストールされていることを確認してください。

```
$ yum info python-devel
```

もしインストールされていなければ、インストールした後に ROOT のビルドを cmake コマンドの実行からやり直します。また、PYTHONPATH の環境変数が正しく設定されているかも確認しましょう。

2.1.8 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^{*22}。

```
$ cd $ROOTSYS/tutorials
$ root

Welcome to the ROOT tutorials
```

^{*22} これは\$ROOTSYS/tutorials/rootlogon.C というファイルが存在するため、\$ROOTSYS/tutorials の中で ROOT を起動すると挙動が変わるからです。

```
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^{*23}。

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が、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 のようなウィンドウが表示されるはずですが、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。

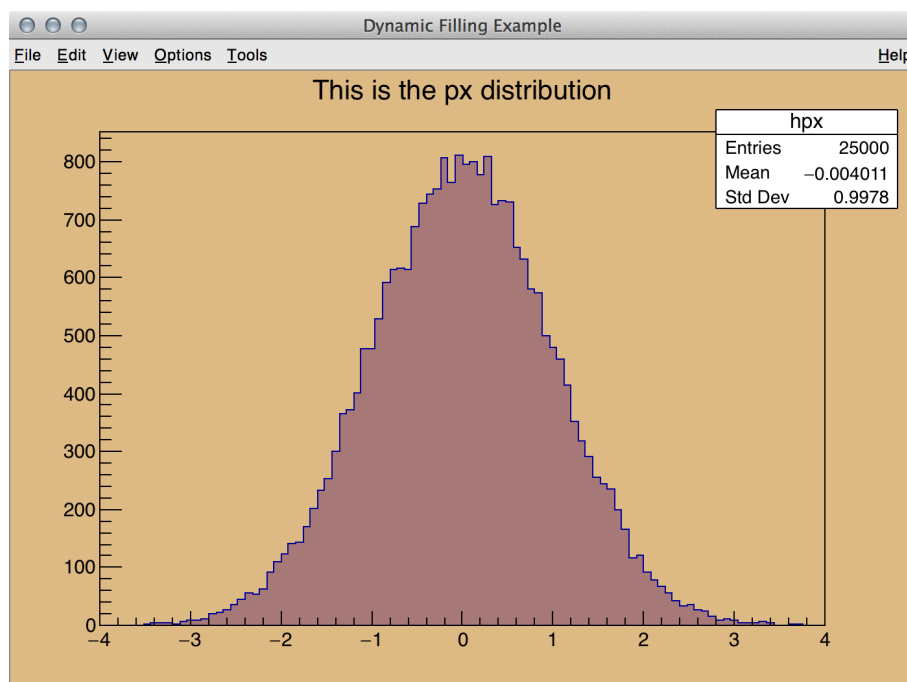


図 2.2 hsimple の実行結果

^{*23} ROOT 5 は CINT というライブラリによって、また ROOT 6 では Cling 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。

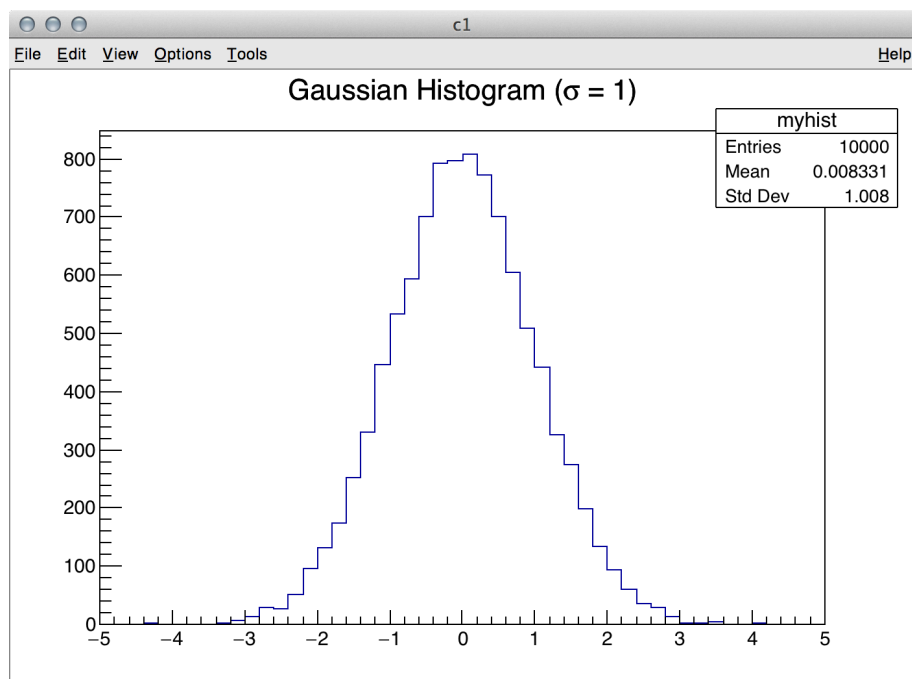


図 2.3 標準偏差 $\sigma = 1$ 、標本の大きさ $n = 10000$ のヒストグラム

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 (standard deviation) $\sigma = 1$ の正規分布 (normal distribution、Gaussian distribution) に従った、標本の大きさ (sample size) $n = 10000$ 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 (#sigma = 1)", 50, -5,
5)
root [1] hist->FillRandom("gaus", 10000)
root [2] hist->Draw()
Info in <TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

という 3 行を打ってみてください。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ください。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ください。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 (#sigma = 1)", 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^{*24}。この `hist` が、ヒストグラムの実体になります。`"myhist"`や`"Gaussian Histogram (#sigma = 1)"`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。

ROOT プロンプトで入力したコマンドは、`.q` や `.x` のような ROOT 固有のコマンドをのぞいて基本的には C++ の文法で記述する必要があります。しかし利便性を高めるためにいくつか C++ とは異なる文法でも動作するようになっています。C や C++ を既に知っている読者の場合、この節で取り上げた例の各行にセミコロン `;` が無いことに気がつくかもしれません。次の例のように、ROOT プロンプトではセミコロンの有無で表示が変わります。スクリプトファイルに書く際は、セミコロンを忘れないようにしてください。

```
root [0] int a = 1
(int) 1
root [1] int b = 2;
root [2]
```

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ください。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^{*25}。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
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

コード 2.1 first_script.C

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

^{*24} 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。

^{*25} 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     TH1D* hist =
3         new TH1D("myhist", "Gaussian Histogram (#sigma = 1)", nbins, -5, 5);
4     hist->FillRandom("gaus", nevents);
5     hist->Draw();
6 }

```

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ください。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)
Info in <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が、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、

```

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、その関数が値を返さないという意味です。 `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`^{*26} に変更してしまうと、

```

root [0] .x first_script2.C(500, 100000)
warning: cannot find function 'first_script2()'; falling back to .L

```

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)
Info in <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。

^{*26} `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。

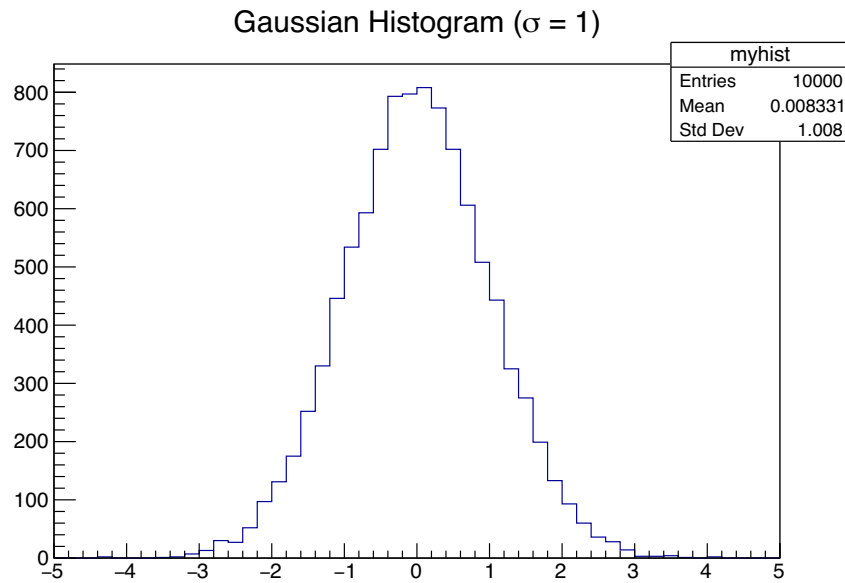


図 2.4 図 2.3 を、PDF 形式で保存し直したものの。拡大しても細部まで滑らかなことが分かる。

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 macOS 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。L^AT_EX 文書で一般的に使われる、PDF 形式で出力結果を保存してみましょう^{*27}。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 PDF 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。いくつか保存形式が選べますので、試してみましょう。JPEG 形式は図の細部が潰れますので、絶対に使わないでください。ウェブページに掲載する目的などでビットマップ画像が必要な場合は、PNG 形式 (.png) にしてください。GIF 形式は最大で 256 色までしか使用できないので、画面で見ている色と変わってしまう可能性があります。

メニューからいちいち保存するのは面倒ですので、スクリプトを実行したらキャンバスをプロンプトから PDF で保存してみましょう。

```
root [0] .x first_script.C
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [1] c1->SaveAs("his.pdf")
Info in <TCanvas::Print>:  pdf file hist.pdf has been created
```

ROOT の生成する横長の PDF には /Rotate 90 というタグが含まれています。dvi2pdf コマンド^{*28}や Apple Keynote の古いバージョンなどでこれを正常に処理できないことがあり、そのため作成した PDF が 90 度回転して表示されるという問題があります (ROOT の問題ではありません)。これを回避するためには、横長の図の場合に c1->SaveAs("hist.pdf") などと実行するのではなく、c1->Print("hist.pdf", "pdf Portrait") と

^{*27} 最近の L^AT_EX 環境では latex コマンドよりも pdflatex コマンドを使用して PDF ファイルを生成するのが一般的になりつつあります。そのため、論文出版社も EPS 形式ではなく PDF 形式で図を受け付けています。また EPS よりも PDF のほうがパソコン上での閲覧も楽なため、特に理由がない場合は PDF で保存しましょう。ただし、本書のような日本語文書では platex コマンドがまだまだ使われています。

^{*28} TeX Live 2018 以降では問題ありません。

実行しましょう。

```
root [1] c1->Print("hist.pdf", "pdf Portrait")
Info in <TCanvas::Print>: pdf file hist.pdf has been created
```

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。携帯電話の文字入力の予測変換機能のようなものです。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()
Info in <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
GetAsymmetry
GetAt
.GetAxisColor
```

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。候補を眺めると、`GetMean` や `GetStdDev`^{*29} 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、

^{*29} `GetRMS` というメンバ関数も見つかると思います。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たそうです (PAW で間違っって使ってしまった用語との互換性を保つため、ROOT でもわざと間違っった用語を使い続けた)。そのためこの業界では `RMS` という用語を間違っ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ください。最近の ROOT では、`GetRMS` と全く同じ働きをする関数として `GetStdDev` が用意されており、後者が本体です。

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

と表示されます。これは、`TH1D::GetMean`^{*30}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()
(Double_t) 0.00833117
```

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が、同様にして、`TH1D::GetStdDev()`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が、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点、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`の中の関数`first_example`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。

2.3 やっておきたい初期設定

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^{*31}。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。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。

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。ただし、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。このように数式を文字列中に組み込むとゴシック体では格好悪いので、好みによって Times に変更します。

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

*31 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

```

{
  // Recommended color settings for color vision deficiency
  gStyle->SetPalette(kBird); // Default in recent ROOT, but just in case.

  // 0 : White, 1 : K, 2 : R, 3 : G,
  // 4 : B, 5 : C, 6 : M, 7 : Y
  // Avoid bright yellow and green for better presentation.
  gROOT->GetColor(3)->SetRGB(0., .7, 0.); // Green (0., 1., 0.)->(0., .7, 0.)
  gROOT->GetColor(5)->SetRGB(1., .5, 0.); // Yellow (0., 1., 0.)->(0., .7, 0.)
  gROOT->GetColor(7)->SetRGB(.2, .6, .6); // Cyan (.4, 1., 1.)->(0.25, 0.25, 0.75)

  // Okumura's preference
  gStyle->SetPadBorderSize(0); // Remove unnecessary margin in pads
  gStyle->SetFuncWidth(2); // Set the default line width of functions to 2
  gStyle->SetLegendBorderSize(0); // Remove the legend boarder

  // 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)
  // This is optional, but recommended if you mix TLatex
  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->SetLegendFont(fontid);
}

```

ひとつ注意して欲しいのが、あなたの画面で見えている色の設定は、万人に同じように見えるわけではないということです。人間は色覚特性というものを持ち、例えば赤と緑の区別が困難な人も存在します。またパソコンの液晶ディスプレイで見えている状態と、学会会場のプロジェクタで映される色はコントラストが異なるため、明るい黄色や黄緑は認識しにくくなります。さらに白黒印刷した場合には、カラー印刷とは全く違う見え方をします^{*32}。そのため、なんでもかんでも好きな色に設定して良いわけではありません。どのような色覚特性であっても、どのような会場でも、どのような印刷方法でも色や線の区別をつけられるように心がけましょう。

ROOT 5.30 より古いバージョンを使っている場合、図 2.4 の背景色が薄い灰色になります。これでは通常の論文で

^{*32} 紙媒体の雑誌は多くのページが白黒印刷であり、また科研費や学振特別研究員の申請書は白黒印刷で審査員に送付されます。

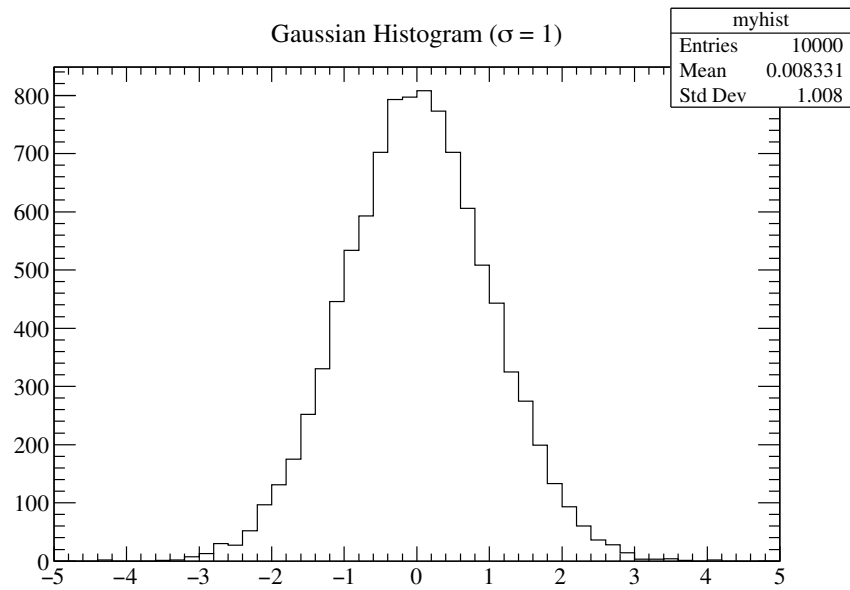


図 2.5 `~/rootlogon.C` を使って図 2.4 の見栄えを変更したもの

使用するには問題があります。研究室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ので、次の設定を追加してください。

```
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->SetPalette(1) ;// Rainbow color palette
```

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     printf("Hello World!\n");
5
6     return 0;
7 }
```

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

^{*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 もしくは Clang をインストールして下さい。「Scientific Linux GCC intall」「Mac Clang 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。Mac の場合は Xcode を App Store からダウンロードしてインストールして下さい（無料）。

^{*4} GNU Compiler Collection (GCC) に含まれる、C++ 用のコンパイラです。Mac で標準的に使用する Clang では clang++ というコマンドが代わりに使えますが、g++ コマンドが clang++ を実際には呼び出すため、GCC の入っていない Mac でも g++ コマンドを使うことができます。

^{*5} http://ja.wikipedia.org/wiki/Hello_world などに解説あり。

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

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、実行してみてください。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なりません。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

*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。

値を入れると、 $ax^2 + bx + c$ を返しますね。これと同じです。main 関数の戻り値は、必ず整数 (integer) でなくてはなりません。この戻り値が 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;
```

と書くことで、確かに 0 を返す設計になっています。戻り値は、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、このプログラムは異常終了したと 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、戻り値という用語ができました。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、戻り値は int になっています。このような関数を作ってしまうと、

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

として変数 after に関数の戻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、

```
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) を取るのが可能です。

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 という名前かというと、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

^{*11} C++ では、四則演算の記号 +、-、×、÷ はそれぞれ +、-、*、/ で表します。

コード 3.2 triple.cxx

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

ります*12。

さて、コード 3.2 の例では、triple 関数は引数と戻り値が 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。そこで引数と戻り値を 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。このように戻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こと、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。

*12 C++ で扱われる小数には精度がつきまっています。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、メモリが無尽大必要になります。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 3.3 triple2.cxx

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

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。

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 true を、そうでなければ false を返します。つまり

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

とすれば、a と b の値はそれぞれ true と false になります。if 文の中身 ({} と {} で囲まれた部分) は、条件式が true もしくは真 (非ゼロの値) 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

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     printf("%f is smaller\n", min(39.2, 48.5));
8     printf("%f is larger\n", max(103.8, -3.2));
9
10    return 0;
11 }
12
13 double min(double v1, double v2) {
14     double ret;
15     if (v1 < v2) {
16         ret = v1;
17     } else {
18         ret = v2;
19     }
20
21     return ret;
22 }
23
24 double max(double v1, double v2) { return v1 > v2 ? v1 : v2; }
```

終了します。

<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 true を返し、後者は等しくないときに true 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。これは三項演算子もしくは条件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、?: 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、非常に初等的手段で円周率 π を計算してみましょう。

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

```

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*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さない。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

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

や

*¹⁵ $n \rightarrow \infty$ 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 1 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 += 1;
```

と同じ意味を持ちます^{*17}。これをインクリメント演算子 (increment operator) と呼びます。

`i++` (後置演算、pre-increment) の代わりに `++i` (前置演算、post-increment) という書き方を次のようにする場合もあります。単純な `for` 文では動作結果に違いはありませんが^{*18}、コンパイラの生成する実行ファイルは若干高速になることがあるかもしれません。本書では `++i` を使用します。

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

`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

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

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。試してみてください。

^{*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。

^{*18} `y = ++x;` や `y = x++;` のように、代入を伴う場合は挙動が異なるので注意してください。前者は `x += 1; y = x;`、後者は `y = x; x += 1;` という意味になります。

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

```
29 #endif // VECTOR_3D
```

コード 3.7 Vector3D.cxx

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

コード 3.8 Vector3D_main.cxx

```
1 #include <cstdio>
2 #include "Vector3D.h"
3
4 int main() {
5     Vector3D v0; // default constructor
6     Vector3D v1(1.5, 2.3, -0.4); // constructor with arguments
7     Vector3D v2 = Vector3D(-3.1, 5.6, 1.9); // operator=, constructor
8     Vector3D v3 = v1 + v2; // operator=, operator+
```

```

9   Vector3D v4(v1 - v2);           // copy constructor, operator-
10  double product = v1 * v2;      // operator*
11
12  v0.Print();
13  v1.Print();
14  v2.Print();
15  v3.Print();
16  v4.Print();
17  printf("v1*v2 = %f\n", product);
18
19  return 0;
20 }

```

3つのファイルに分割したのは、可読性と可搬性^{*19}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^{*20}。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
$ ./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。修正箇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ないので、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。

```

#ifdef VECTOR_3D
#define VECTOR_3D

```

^{*19} 他のプログラムでも使い回しが効くという意味です。

^{*20} 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。

```
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 (includ guard)」と呼びます^{*21}。

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^{*22} 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こと、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^{*23}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。

^{*21} 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。

^{*22} 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。

^{*23} インスタンス (instance) とも呼びます。

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*24。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、一般的な関数は戻り値を持ちます。しかしコンストラクタは特殊な関数であり、戻り値を持ちません。戻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、コンストラクタは戻り値を持たないので分かっているため、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、

```
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入れ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。

*24 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。

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、実際に使うときは

```
Vector3D v3 = v1 + v2;
```

のように使います。

```
Vector3D v3 = v1.operator+(v2);
```

のように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わ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^{*25}。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;
```

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

^{*25} 3.7 を読んでから、再度この段落を読んでみてください。


```

1 #ifndef LORENTZ_VECTOR
2 #define LORENTZ_VECTOR
3
4 #include "Vector3D.h"
5
6 class LorentzVector : public Vector3D {
7 private:
8     double fT;
9
10 public:
11     LorentzVector();
12     LorentzVector(double x, double y, double z, double t);
13     LorentzVector(const LorentzVector& other);
14     virtual ~LorentzVector();
15
16     virtual double T() const { return fT; }
17     virtual void Print() const;
18
19     LorentzVector& operator=(const LorentzVector& other);
20     LorentzVector operator+(const LorentzVector& other);
21     LorentzVector operator-(const LorentzVector& other);
22     double operator*(const LorentzVector& other);
23 };
24
25 #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(), fT(0) {}
6
7 LorentzVector::LorentzVector(const LorentzVector& other)
8     : Vector3D(other), fT(other.fT) {}
9
10 LorentzVector::LorentzVector(double x, double y, double z, double t)
11     : Vector3D(x, y, z), fT(t) {}
12
13 LorentzVector::~LorentzVector() {}
14
15 void LorentzVector::Print() const {
16     printf("(x, y, z, t) = (%lf, %lf, %lf, %lf)\n", X(), Y(), Z(), fT);
17 }
18
19 LorentzVector& LorentzVector::operator=(const LorentzVector& other) {
20     if (this != &other) {
21         Vector3D::operator=(other);
22         fT = other.fT;
23     }
24 }

```



```

25     return *this;
26 }
27
28 LorentzVector LorentzVector::operator+(const LorentzVector& vec) {
29     return LorentzVector(X() + vec.X(), Y() + vec.Y(), Z() + vec.Z(),
30                          fT + vec.fT);
31 }
32
33 LorentzVector LorentzVector::operator-(const LorentzVector& vec) {
34     return LorentzVector(X() - vec.X(), Y() - vec.Y(), Z() - vec.Z(),
35                          fT - vec.fT);
36 }
37
38 double LorentzVector::operator*(const LorentzVector& vec) {
39     return X() * vec.X() + Y() * vec.Y() + Z() * vec.Z() - fT * vec.fT;
40 }

```

コード 3.11 LorentzVector_main.cxx

```

1  #include <cstdio>
2  #include "LorentzVector.h"
3
4  int main() {
5      LorentzVector v0; // default constructor
6      LorentzVector v1(1.5, 2.3, -0.4, 4.2); // constructor with arguments
7      LorentzVector v2 =
8          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 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、測定値の分布かどのようになっているかを表すときに頻りに使われます。物理学実験で目にする例では、不安定粒子の崩壊時間の分布 (指数分布)、結晶シンチレータの発光量の分布 (ガウス分布)、光検出器に微弱光を入射したときの検出光電子数の分布 (ポアソン分布) などがあります。身近な日常生活の例では、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

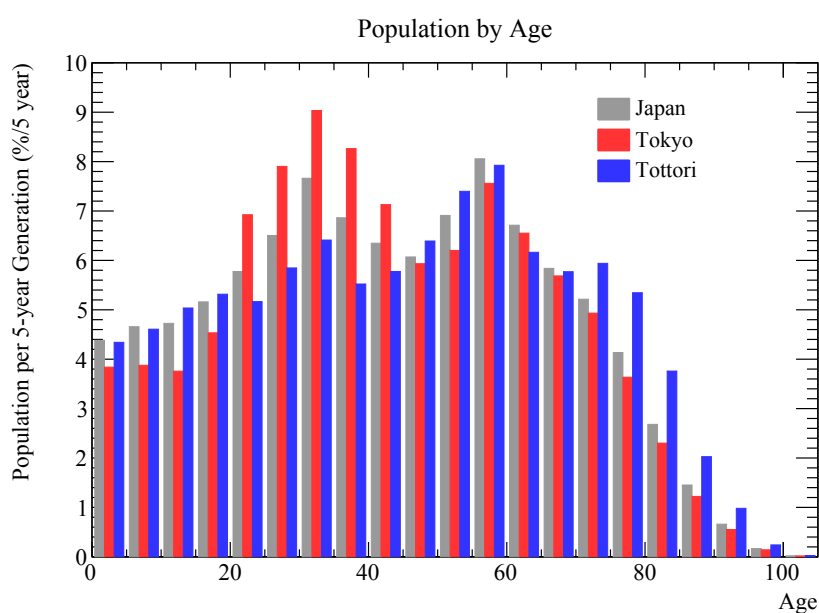


図 5.1 2005 年国勢調査を元にした、全国合計、東京都、鳥根県の年齢別人口分布。コード 9.3 で同じ結果を得られる。データは <http://www.e-stat.go.jp/SG1/estat/List.do?bid=000001007609&cycode=0> から入手可能。

コード 5.1 population.dat

```

1 Japan Tokyo Tottori
2 5578087 476692 26333
3 5928495 481382 27945
4 6014652 466593 30545
5 6568380 562968 32239
6 7350598 859742 31331
7 8280049 981230 35464
8 9754857 1121689 38890
9 8735781 1026016 33490
10 8080596 885146 35032
11 7725861 736656 38768
12 8796499 770054 44873
13 10255164 938669 48068
14 8544629 813422 37384
15 7432610 705944 35001
16 6637497 612400 36028
17 5262801 451357 32420
18 3412393 285738 22804
19 1849260 151770 12294
20 840870 68497 5951
21 211221 17606 1459
22 25353 2215 156

```

が出てくるわけです*1*2。

コード 5.2 population.C

```

1 void population() {
2     std::ifstream fin("population.dat");
3
4     const Int_t kHistN = 3;
5     const Int_t kBinsN = 21;
6     TH1D* hist[kHistN];
7
8     for (Int_t i = 0; i < kHistN; ++i) {
9         std::string str;
10        fin >> str;
11        hist[i] = new TH1D(str.c_str(), str.c_str(), kBinsN, 0, 105);
12    }
13
14    for (Int_t j = 0; j < kBinsN; ++j) {
15        for (Int_t i = 0; i < kHistN; ++i) {
16            Int_t pop;
17            fin >> pop;
18            hist[i]->SetBinContent(j + 1, pop);

```

*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。

```
19     }
20 }
21
22 TCanvas* can1 = new TCanvas("can1", "histogram");
23
24 TH1D* frame = new TH1D(
25     "frame",
26     "Population by Age;Age;Population per 5-year Generation (%/5 year)",
27     kBinsN, 0, 105);
28 frame->SetMaximum(10);
29 frame->Draw();
30
31 TLegend* leg1 = new TLegend(0.65, 0.7, 0.85, 0.85);
32 leg1->SetFillStyle(0);
33
34 Int_t kColor[kHistN] = {kGray + 1, kRed - 4, kBlue - 4};
35
36 for (Int_t i = 0; i < kHistN; ++i) {
37     hist[i]->Scale(100. / hist[i]->GetEffectiveEntries()); // normalize
38     hist[i]->SetLineColor(kColor[i]);
39     hist[i]->SetFillColor(kColor[i]);
40     hist[i]->SetBarWidth(0.28);
41     hist[i]->SetBarOffset(0.08 + 0.28 * i);
42     hist[i]->Draw("bar same");
43     leg1->AddEntry(hist[i], hist[i]->GetTitle(), "f");
44 }
45 leg1->Draw();
46
47 TCanvas* can2 = new TCanvas("can2", "graph");
48 frame->Draw();
49 TGraph* graph[kHistN];
50 TLegend* leg2 = new TLegend(0.65, 0.7, 0.85, 0.85);
51 leg2->SetFillStyle(0);
52
53 for (Int_t i = 0; i < kHistN; ++i) {
54     graph[i] = new TGraph();
55     for (Int_t j = 0; j < kBinsN; ++j) {
56         graph[i]->SetPoint(j, hist[i]->GetBinCenter(j + 1),
57                             hist[i]->GetBinContent(j + 1));
58     }
59     graph[i]->SetLineColor(kColor[i]);
60     graph[i]->SetMarkerColor(kColor[i]);
61     graph[i]->Draw("same");
62     leg2->AddEntry(graph[i], hist[i]->GetTitle(), "l");
63 }
64 leg2->Draw();
65 }
```

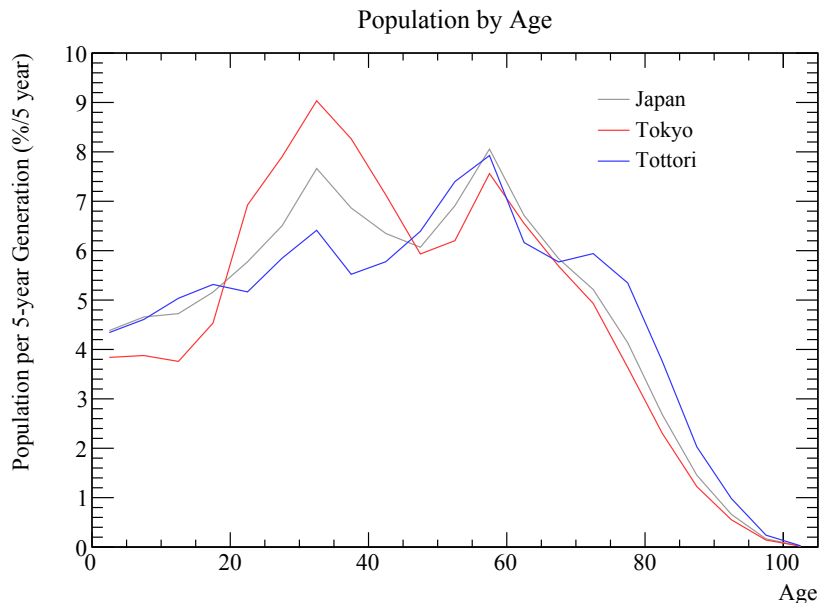


図 5.2 図 5.1 の間違った表示方法の例

5.1.1 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。

5.1.2 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なりません。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です。

図 5.2 では意図的に不適切な表示をしましたが、実際に、研究者といえどもヒストグラムの間違った使い方を行っている場合があります。例えば福島第一原発の事故後、東京大学柏キャンパスの柏地区環境安全管理室ではキャンパス内の空間線量率の測定を行いました^{*3}。多数の測定点での空間線量率の分布として、図 5.3 を掲載しています。

既に説明した通り、図 5.3 の表示方法は不適切です。ヒストグラムは棒グラフで描くべきであり、折れ線グラフにしてはいけません。点と点を線で結んで良いのは、その傾きに意味があるときです。図 5.3 の縦軸の単位は、正確に書くと「ポイント数/(0.02 μ Sv/h)」になります^{*4}。しかしヒストグラムが折れ線で結ばれてしまっているため、このグラフを積分しても、実際の測定点数と同じにはならないでしょう。繰り返しますが、ヒストグラムはどのようなビン幅で作

^{*3} <http://www.kashiwa.u-tokyo.ac.jp/kankyo/ks201111/>

^{*4} 「ポイント数」は本当は「単位」ではありません。

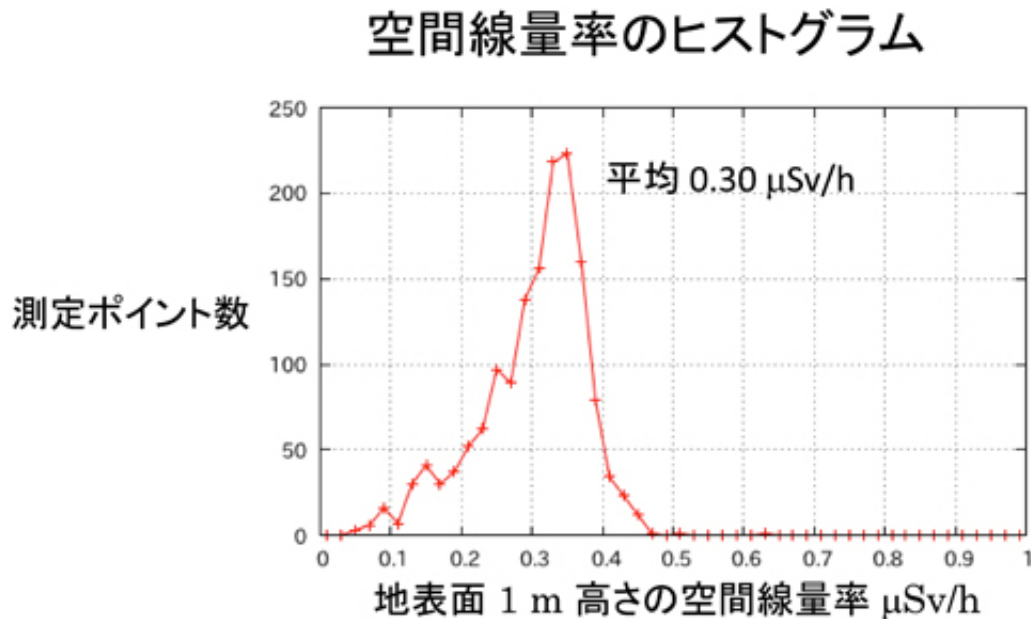


図 5.3 東京大学柏キャンパス内の空間線量率の分布 (柏地区環境安全管理室より引用)

図しても面積は一定です。

5.2 1次元ヒストグラム

それでは、ROOT でどのようにヒストグラムを扱うのか順番に説明します。まずは1次元のヒストグラムからです。ROOTには1次元のヒストグラムを扱うためのクラスが複数存在します。純粹仮想クラスである TH1、それを継承した TH1C、TH1S、TH1I、TH1F、TH1D です。最初は TH1D だけ覚えておけば十分です^{*5}。

それでは TH1D の簡単な使い方の説明です。まず ROOT を起動し、次の操作を行って下さい。図 5.4 のような図が表示されるはずですが。

```
root [0] TH1D* hist = new TH1D("hist", "Gaussian Distribution", 100, -10, 10)
(TH1D *) 0x7fc56c639860
root [1] hist->GetXaxis()->SetTitle("Physics Quantity #it{X}")
root [2] hist->GetYaxis()->SetTitle("Entries")
root [3] const Double_t kMean = 3.
(const Double_t) 3.00000
root [4] const Double_t kSigma = 2.
(const Double_t) 2.00000
root [5] for(Int_t i = 0; i < 10000; i++){
root (cont'ed, cancel with .@) [6] Double_t x = gRandom->Gaus(kMean, kSigma);
root (cont'ed, cancel with .@) [7] hist->Fill(x);
root (cont'ed, cancel with .@) [8] }
root [9] hist->Draw()
Info in <TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [10] hist->GetMean()
```

^{*5} TH1D は、それぞれの bin に詰まった値が Double_t 型で保存されます。TH1C、TH1S、TH1I、TH1F は bin の値が Char_t、Short_t、Int_t、Float_t 型で保存されます。消費されるメモリの量、整数で値を保持したいかそれとも浮動小数でも良いか、などを考慮して使うヒストグラムのクラスを選択します。

```
(Double_t) 3.00946
root [11] hist->GetMeanError()
(Double_t) 0.0198828
root [12] hist->GetStdDev()
(Double_t) 1.98778
root [13] hist->GetStdDevError()
(Double_t) 0.0140593
```

5.3 2次元ヒストグラム

```
root [0] TH2D* h2 = new TH2D("h2", "2D Gaussian Distribution;#it{x};#it{y};Entries",
    100, -10, 10, 100, -10, 10)
(TH2D *) 0x7fe8c3615eb0
root [1] const Double_t kSigma = 2.
(const Double_t) 2.00000
root [2] for(Int_t i = 0; i < 100000; i++){
root (cont'ed, cancel with .@) [3] Double_t x = gRandom->Gaus(0, kSigma);
root (cont'ed, cancel with .@) [4] Double_t y = gRandom->Gaus(0, kSigma);
root (cont'ed, cancel with .@) [5] h2->Fill(x, y);
root (cont'ed, cancel with .@) [6] }
root [7] TCanvas* can = new TCanvas("can", "can", 600, 600)
(TCanvas *) 0x7fe8c359a9a0
root [8] h2->Draw("colz")
```

5.4 3次元ヒストグラム

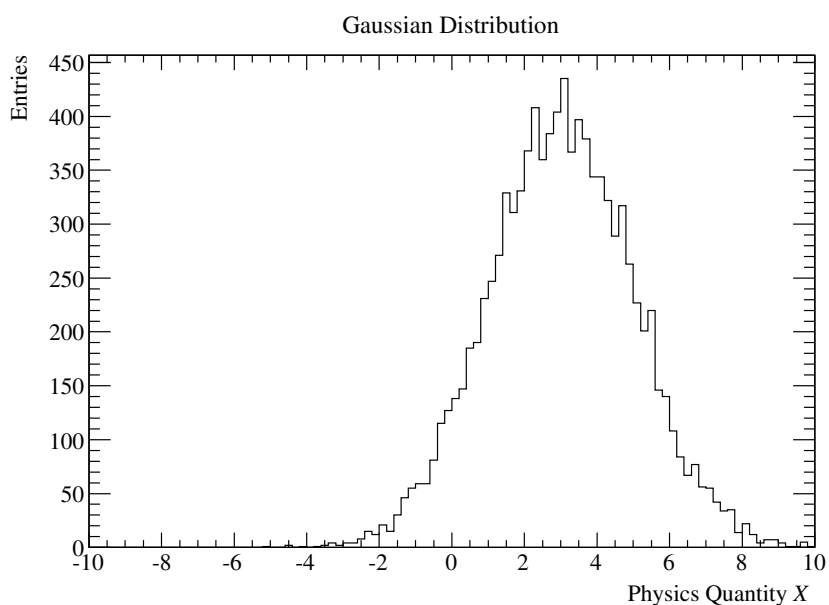


図 5.4 物理量 X (平均 $\mu = 3$ 、標準偏差 $\sigma = 2$) のガウス分布の例

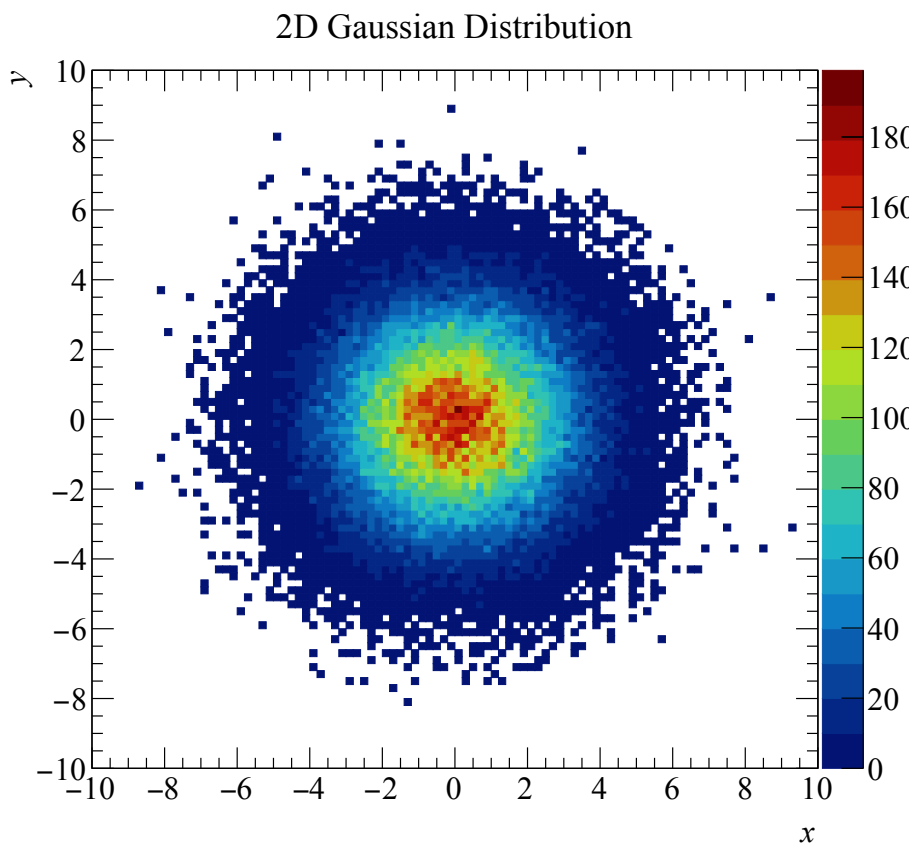


図 5.5 物理量 x 、 y (平均 $\mu_x = \mu_y = 0$ 、標準偏差 $\sigma = 2$) の 2 次元ガウス分布の例

6 グラフ

7 Tree

8 Python

8.1 注意

Python に関する技術の発展は ROOT のそれより目覚ましく、この文書も含め、ウェブ上に転がる情報が簡単に陳腐化します。これはオープンソースコミュニティの活発さや、Python という言語の人気のためです。ここに書かれている情報は 2019 年 4 月現在、筆者の知る限りの範囲で、なるべく新しいものになるように務めています、数年で古くなるということを知っておいてください。

8.2 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 Cling に似ています。
- 様々なパッケージ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、天文学界で 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パッケージ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 に挑戦してみしましょう。

```
In [1]: import ROOT
In [2]: from astropy.io import fits
In [3]: hist = ROOT.TH1D("hist", "Energy distribution;Energy (MeV)", 100, 0, 1e3
...: )
In [4]: energies = fits.open("photon/lat_photon_weekly_w009_p302_v001.fits")[1]
...: .data.field("ENERGY")
In [5]: for i in range(energies.size):
...:     hist.Fill(energies[i])
```

```
In [6]: hist.Draw()
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

8.3 簡単に試してみる

Python は ROOT と同様にコマンドラインからインタプリタを呼び出して 1 行ずつコマンドを実行することもできれば、スクリプトにまとめて実行することもできます。

8.3.1 起動と終了

まず Python (ここでは IPython) のインタプリタに 1 行ずつ命令を渡していく操作を試してみましょう。面白い例ではありませんが、3.1 節で解説した C++ の `hello_world.cxx` を Python に直してみましょう。

まず IPython を起動します。

```
$ ipython
Python 3.7.3 (default, Apr 18 2019, 10:17:38)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

次に、C の `printf` に似た関数である `print` 関数を使って、「Hello World!」という文字列を出力してみます。

```
In [1]: print('Hello World!')
Hello World!
```

Python 2 までは `print` 関数の使い方として、次のような例がよく用いられていました。しかし Python 3 では多くの仕様変更になり、このような `print` 関数の使い方は Python 3 で動作しなくなりました。書籍やインターネット上にある情報は Python 2 と Python 3 のものが混在していますので、注意してください。

```
In [1]: print 'Hello World!'
Hello World!
```

最後に、`Ctrl-D` を入力すると IPython を終了できます^{*1}。本当に終了して良いのか聞かれるので、良い場合はそのまま `Enter` キーを、IPython を続ける場合は `N` キーを押してから `Enter` キーを押します。

```
In [2]:
Do you really want to exit ([y]/n)?
$
```

8.3.2 スクリプトの書き方

Python スクリプトの使い方は、大別して三通りあります。

*1 Control キーを押しながら、D キーを同時に押す

8.4 Python 3 のインストール

macOS や CentOS 7 では、標準で Python がインストールされているはずですが、ただし、この Python のバージョンは Python 2 のため、特にこだわりが無く Python をこれから新たに習得する人は、Python 3 を導入して使い始めることを推奨します。

8.4.1 macOS Mojave の場合

macOS で新たにソフトウェアを追加する場合、節 A.1 で説明する通り、Homebrew というパッケージ管理システムを使うのが一般的です。ここでは Homebrew がすでに入っているという前提で説明します。

次のコマンドを実行することで、python3 と pip3 が、

```
$ brew install python3
```

また次のコマンドで Python 3 用の ipython がインストールされます。

```
$ pip3 install ipython
```

問題なく Python 3 が入ったことを確かめるため、python3 コマンドを実行してみます。次のように表示されれば成功です。

```
$ python3
Python 3.7.3 (default, Apr 18 2019, 10:17:38)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Homebrew では Python 3 のコマンドを python3 や python3.7 としてインストールし、python コマンドはデフォルトの /usr/bin/python として Python 2 のまま残ります。ただし、他の環境では python2 が Python 2 で python が Python 3 だったりするので、自分が Python 2 を使っているのか Python 3 を使っているのか、よく注意してください。

```
$ python
Python 2.7.10 (default, Feb 22 2019, 21:17:52)
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.37.14)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

8.4.2 CentOS 7 の場合

CentOS 7 のような Red Hat Enterprise Linux (RHEL) クローンと呼ばれる Linux ディストリビューションでは、yum (ヤム) と呼ばれるパッケージ管理システムが標準的に使われます。多くのソフトウェアを yum から入手可能ですが、Python 3 は標準では入れられないため、まずレポジトリを追加します。

```
$ sudo yum install -y https://centos7.iuscommunity.org/ius-release.rpm
```

これで yum がソフトウェアを検索する際の新たな場所が追加されます。次に python36u-devel をインストールし、python3.6 と pip3.6 をインストールします。


```
$ sudo yum install python36u-devel
$ sudo yum install python36u-pip
```

これで、python3.6 を実行すると Python 3.6 を起動することができるようになりました。

```
$ which python3.6
/usr/bin/python3.6
$ python3.6
Python 3.6.7 (default, Dec 5 2018, 15:02:05)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

デフォルトで入っている Python は 2.7.5 です。これは /usr/bin/python として残ります。

```
$ which python
/usr/bin/python
$ python
Python 2.7.5 (default, Aug 4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

次に pip3.6 を使って Python 3 用の IPython をインストールします。

```
$ which pip3.6
/usr/bin/pip3.6
$ sudo pip3.6 install ipython
$ sudo pip3.6 install --upgrade pip
```

Python 2 用の IPython を入れていない場合、/usr/bin/ipython と /usr/bin/ipython3 が Python 3 用の IPython になります。

```
$ which ipython
/usr/bin/ipython
$ which ipython3
/usr/bin/ipython3
$ ipython3
Python 3.6.7 (default, Dec 5 2018, 15:02:05)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

8.5 pip

すでに前節までで見たように、pip もしくは pip3 というコマンドを使って Python の追加パッケージをインストールする方法を説明しました。この pip (ピップ) というコマンドは Python のコミュニティで広く使われているパッケージ管理システムのひとつです。Python Package Index (PyPI、パイピーアイ)^{*2}に登録されている様々な追加

^{*2} <https://pypi.org>

パッケージを自動でインストールします。通常は `/usr/local/lib/python/site-package` という場所に新しいパッケージが追加されます。

`pip` の基本は次のコマンドです。 `search` でその名前のパッケージの情報やすでにインストールされているかを調べ、 `install` でインストールします。

```
$ pip search PACKAGE_NAME
$ pip install PACKAGE_NAME
```

8.6 IPython

IPython

8.7 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/>

8.8 Python の基本

8.9 PyROOT

8.9.1 C++ から Python へ

8.9.2 メモリ管理

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

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

```

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

```

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 ($\equiv 0$) から最大値 ($\equiv 1$) 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");
```

という1行を加えて、起動時に読み込ませておくのも大丈夫です。

*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、(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。

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

```
31 hist->Draw("same colz");  
32 gPad->Update();  
33 }  
34 }
```

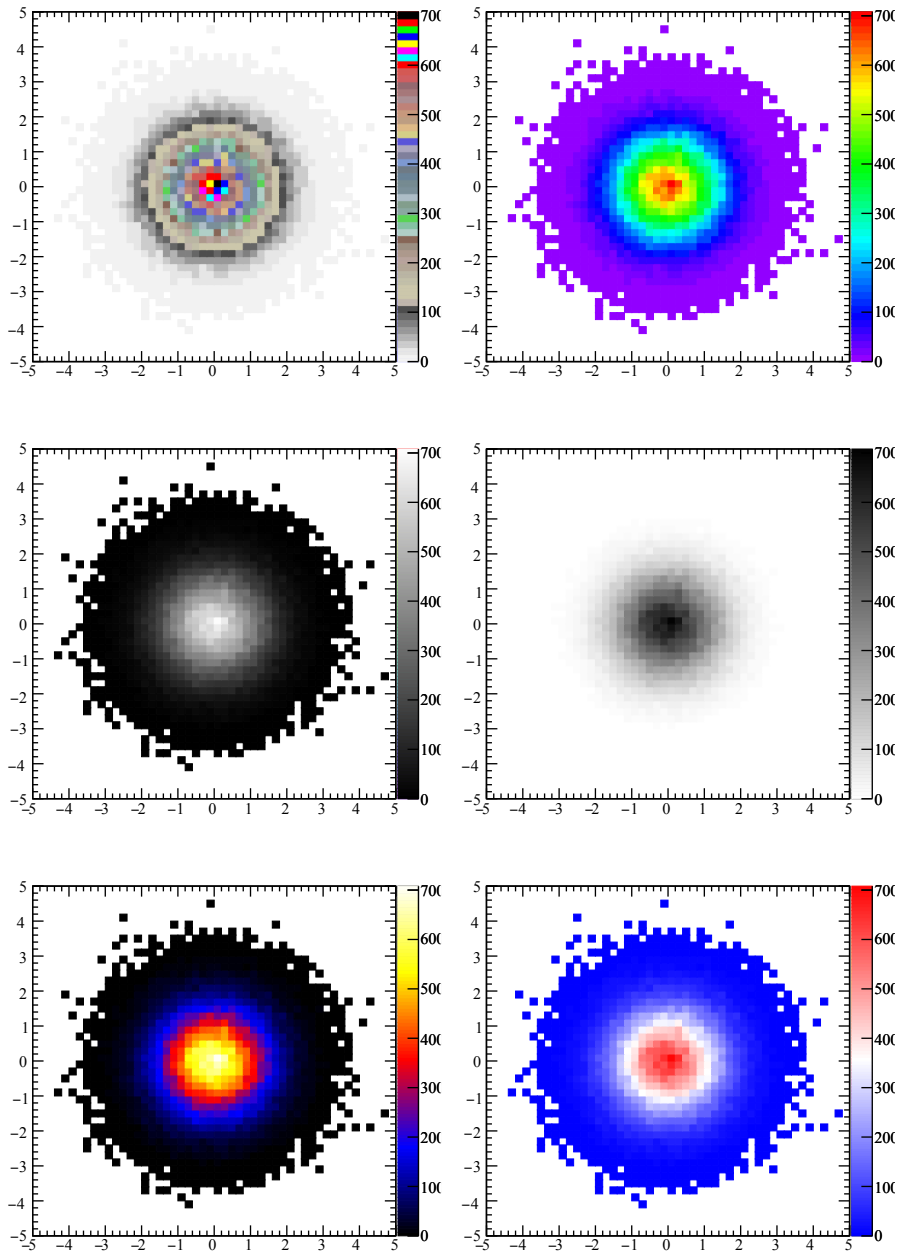


図 9.1 コード 9.2 の出力結果

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     gStyle->SetOptStat(0);
3
4     TCanvas* can = new TCanvas("can", "can", 400, 400);
5
6     TH2D* hist = new TH2D("hist", "", 50, -5, 5, 50, -5, 5);
7     hist->SetContour(100);
8     hist->GetXaxis()->SetAxisColor(0);
9     hist->GetYaxis()->SetAxisColor(0);
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    }
16
17    gROOT->ProcessLine(".L color_def.C");
18    BPalette();
19    hist->Draw("colz");
20    gPad->Update();
21    TPaletteAxis* palette =
22        (TPaletteAxis*)hist->GetListOfFunctions()->FindObject("palette");
23    Int_t col = palette->GetValueColor(hist->GetMinimum());
24    hist->Draw("colz");
25    gPad->SetFrameFillColor(col);
26    gPad->Update();
27 }

```

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 ExactSizeCanvas.C

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

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

```

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

```

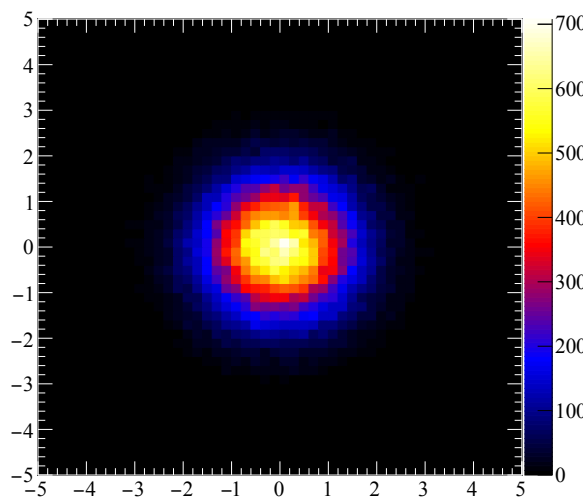


図9.2 コード9.3の出力結果

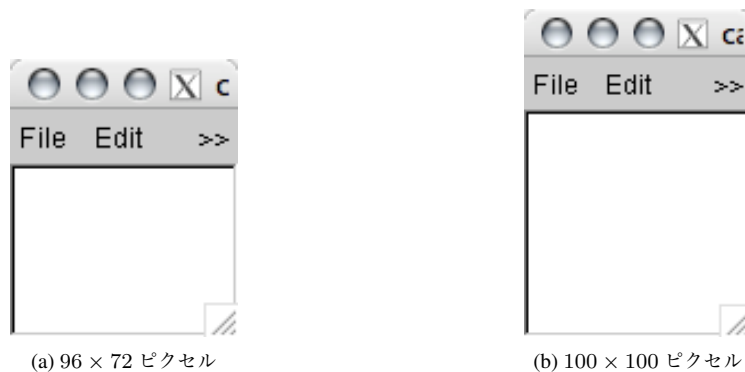


図9.3 TCanvasの描画領域の違い

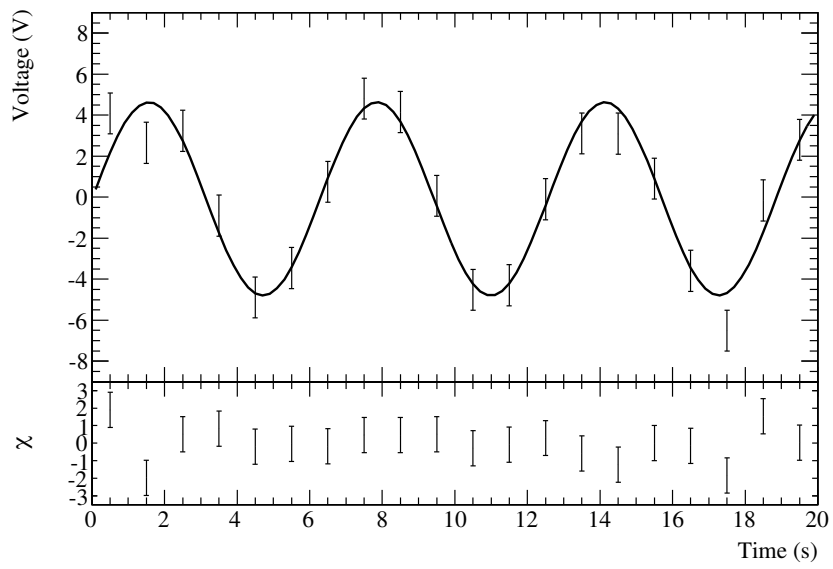


図 9.4 コード 9.5 の出力結果

10 統計学の基礎

実験データの解析作業の目的のひとつは、何らかの物理量をそこから抽出し、その値が統計学的にどれだけ正しいかを評価することです。例えば電子の質量が約 511 keV であることを私たちは知っていますが、その真の値がいくつかは誰も知りません^{*1}。物理実験から統計学的な表現で言えるのは、電子質量は

$$m_e = 510.998\,946\,1 \pm 0.000\,003\,1 \text{ keV} \quad (10.1)$$

ということだけです [2]。これは 510.998 946 1 keV がこれまでの実験でわかった最も良い電子質量の推定値であり、その誤差は 0.000 003 1 keV である、もしくは言い方を変えると、約 68% の確率^{*2}で 0.000 003 1 keV の範囲に収まっているという意味です。

ROOT ではヒストグラムが自動で計算する平均値や標準偏差といった数値を始めとして、カイ二乗フィットや検定など、様々な統計処理が現れます。これらを理解せずに ROOT を使ったりデータ解析を行うと、間違った物理量やその誤差を導出する恐れがありますし、また論文を読んでも実験データの解釈を正しくできないでしょう。

実験系高エネルギー宇宙物理学の大学院生が必要とする統計学の基本的な情報、またそれを ROOT でどのように扱えば良いかをこの章では説明します。

10.1 参考書

この章では非常に基本的なことしか触れませんし、その目的は統計学を学ぶことではなく、ROOT でどのように統計学的な考え方をするかを指針を与えることです。統計学、特に素粒子物理学や宇宙物理学で重要となる統計処理に関しては、いくつかの参考書を挙げるにとどめます。

1. C. Patrignani and Particle Data Group, “Review of particle physics”, *Chinese Physics C* **40**, 100001 (2016) [☞](#)
素粒子物理学の研究者が 2 年ごとに編纂している、無料で入手可能な書籍です。この本の確率と統計の章は短いですがよくまとまっています。
2. Olaf Behnke et al. “Data Analysis in High Energy Physics: A Practical Guide to Statistical Methods”, Wiley-VCH (2013) [☞](#)
比較的最近の本で、unfolding や系統誤差の取り扱い、また天文学での事例など多岐にわたり解説しています。
3. Louis Lyons, “Statistics for Nuclear and Particle Physicists”, Cambridge University Press, (1989) [☞](#)
古くからある素粒子・原子核物理向けの統計の教科書です。指導教員の多くが薦めると思います。
4. Glenn F. Knoll, “Radiation Detection and Measurement”, Wiley, 4th edition, (2010) [☞](#)
放射線計測に関する様々な情報が載っている教科書ですが、簡単な統計の話も取り扱っていて簡単に読めます。

^{*1} 光速 c のように 299 792 458 m/s と正確に「定義」されている物理量も存在します。

^{*2} より正確には約 68.27% であり、これは正規分布の $\pm 1\sigma$ の範囲に入る確率です。この意味については本章で後述します。

10.2 基本的な用語

10.2.1 統計誤差と系統誤差

一般的に測定値には誤差がつきます。例えばある放射性同位体の放射能を測定することを考えてみましょう。10 秒間に 100 回の崩壊が計数されたとすると、この放射性同位体は 10 Bq (ベクレル^{*3}) だと言えます。しかしこれは 10 秒間で崩壊した数がたまたま 100 回だけであり、繰り返し測定を行うと 110 回だったり 90 回だったりする可能性もあります。したがって我々が決定できる 10 Bq というのはその測定における最良の推定値であり、確率に基づいた誤差が伴います。これを**統計誤差 (statistical error)** と呼びます。

この例の場合、原理的には測定時間を無限に長くすることで、統計誤差を限りなく小さくすることができます。しかし無限の測定時間を使うことは不可能であり、またその間に放射能自体が変化してしまいます。我々はあらゆる実験において、限られた時間、限られた装置、限られた金銭的・人的資源を使うことしかできませんので、必ず統計誤差と付き合うこととなります。

一方、10 秒間の測定と簡単に言っても、10 秒間を精確に決めるのは困難です。使っている時計が原子時計でもない限り、多少のずれは必ず発生します。10 秒間だと思って計測していても、実際には 9.9 秒だったり 10.1 秒だったり、測定系自体に狂いが存在する可能性があります。この場合、10 Bq という推定値は、本当は 10.1 Bq だったり 9.9 Bq だったりするかもしれません。このような測定系自体の持つ誤差を**系統誤差 (systematic error)** と呼びます。

例えば上述の時計の場合、時計がどの程度ずれているかというのを、何かしらの外部装置を使って較正し、系統誤差を評価する必要があります。数学の計算さえ間違えなければ、一般的にはあらゆる実験で統計誤差を計算することが可能ですが、系統誤差の評価方法には王道は存在せず、それを解説した書籍も多くないでしょう。本書でも系統誤差については解説しませんので、各実験ごとの系統誤差の評価は、指導教員や先輩に相談しましょう。

10.2.2 母集団と標本

カミオカンデ実験の検出した超新星 SN 1987A からの 11 個のニュートリノは、SN 1987A から放出された全てのニュートリノではありません。大マゼラン雲から地球の方向へ飛んできたニュートリノのうち、たまたまカミオカンデで検出された 11 個に過ぎません。我々は地球上に住んでいるため、SN 1987A で発生した全方向のニュートリノを観測することはできませんし、また検出器の体積にも限りがあるため、地球に飛来した全てのニュートリノを検出することはできません。

このように物理実験では、発生した全ての事象を観測することができず、その一部だけを観測することによって物理量 (例えばこの場合は超新星爆発のエネルギーがニュートリノにどれだけ渡されるかなど) を測定することが多々あります。この一部分のことを**標本 (sample)** と呼び、それに含まれる事象の数を**標本の大きさ (sample size)** と呼びます^{*4}。一方、SN 1987A で放出されたニュートリノの全てを**母集団 (population)** と呼びます。すなわち、得られた標本から母集団の性質を推定するわけです^{*5}。このような母集団の性質を特徴付ける変数のことを**母数 (parameter)** と呼びます。

たかだか数百の電話回答を使って 1 億人の世論を調査できるのは、このような標本から母集団を推定するという統計処理を行うためです。また選挙の出口調査の結果から開票 1 分で当選確定が出るのも同じ理由です。さらに料理の味見

^{*3} 1 秒間に平均 N 回崩壊する放射性同位体の放射能を N Bq と表します。

^{*4} 「標本数」(the number of samples) と誤用されることがありますが、これは用語として正しくありません。カミオカンデの他に IMB 実験と Baksan 実験でもニュートリノが同時に検出されており、標本は合計で 3 つあることとなります。このとき、標本数は 3 であると言えます。

^{*5} 検出されたニュートリノは検出されやすいニュートリノでもあります。水タンクの有効体積はニュートリノの種類やエネルギーに依存するため、母集団を代表するものではありません。エネルギー依存性などを考慮し、実際には母集団がどのようなものかを推定します。

をするときも、全て食べずにひと口舐めれば問題ありません。これらは全て、標本は母集団の性質を反映しているという前提に立っています。

10.2.3 平均値と分散と標準偏差

標本を特徴付ける指標として、もっともよく現れるのが**平均値 (mean)**^{*6*7}です。標本の平均値 \bar{x} は

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{x_1 + x_2 + \cdots + x_n}{n} \quad (10.2)$$

と表されます。ここで n は標本の大きさ、 x_i は何らかの測定値です。これはあくまで標本の平均であり、母集団の平均値 (母平均) μ とは異なります。

平均値はその標本が典型的にどのような値を持っているかを示す一つの指標です。例えばガンマ線や X 線がシンチレータに入射した時に得られる光子数や、1 光子を検出した時に得られる光電子増倍管の出力電荷には確率的なばらつきがあります。662 keV のガンマ線が繰り返しシンチレータに入射したとしても、得られる出力電荷は入射のたびに変化します。しかし平均値を計算することで、平均的にはどのような応答が得られるのかを示すことができます。

測定値 x_i のばらつきは、統計学では**分散 (variance)** という散らばりの尺度で表されます。母集団の持つ分散 (母分散) を σ^2 で表し、標本の持つ分散 (標本分散) を s^2 と表します。ここで標本分散には 2 種類あり、母平均 μ が既知の場合は

$$s^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2 \quad (10.3)$$

であり、母平均を知らず標本平均のみを知っている場合は

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (10.4)$$

と表すことができます。通常は母平均 μ を知りませんので、 $n-1$ で割った式 (10.4) を使うことになります。 n ではなく $n-1$ で割る理由は、 $n-1$ で割ることで標本分散の期待値 $E(s^2)$ が母分散 σ^2 に等しくなるからです。

値のばらつきの尺度として、例えば半値全幅 (はんちぜんはば、full width at half maximum、FWHM) や最大値と最小値の差を使うなど、いくつかの考え方はありますが、標準偏差を用いるのはそれが数学的に取り扱いやすく、また後で述べるように色々な場所で姿を表すからです。

分散はそのままでは測定値と次元が違うため、多くの場合その平方根である**標準偏差 (standard deviation)** σ もしくは s を用います。ROOT のヒストグラムでは標本の標準偏差を `TH1::GetStdDev` で取り出すことができますが、 $n-1$ ではなく n で割った分散の平方根を返すので注意が必要です。特に n が小さいとき、その値は母集団の標準偏差より小さくなる傾向が出ます。

母平均 μ を知らないとき、その値を推定するには標本平均 \bar{x} を使うしかありません。しかし標本平均はあくまで母平均の近似値であり、真の値からは必ずずれています。このずれは標本平均の分散 $V(\bar{x})$ を求めることで得られます。独立な試行 (それぞれの測定が互いに影響を及ぼさないもの) の和の分散は、それぞれの試行の分散の和で表されることが知られています。例えば 2 回の測定 X_1 と X_2 があつたとき

$$V(X_1 + X_2) = V(X_1) + V(X_2) \quad (10.5)$$

となります。また係数 a に対して

$$V(aX) = a^2 V(X) \quad (10.6)$$

*6 より意味を明確にするため (相乗平均と区別するため)、算術平均 (arithmetic mean)、相加平均とも呼ばれます。

*7 平均の英訳として average も存在しますが、これは算術平均以外にも中央値 (median) や最頻値 (mode) など、分布を特徴付ける広い意味での「平均的な」値も指す言葉であり、「代表値」とする方が適切です。

が成り立ちます。したがって、標本平均の分散は

$$V(\bar{x}) = V\left(\frac{1}{n} \sum_{i=1}^n x_i\right) \quad (10.7)$$

$$= \frac{1}{n^2} V\left(\sum_{i=1}^n x_i\right) \quad (10.8)$$

$$= \frac{1}{n^2} \cdot n\sigma^2 \quad (10.9)$$

$$= \frac{\sigma^2}{n} \quad (10.10)$$

となります。このことから、測定回数（標本の大きさ） n を増やすと標本平均の分散は n に反比例して小さくなることが分かります。また標本平均の誤差はこの平方根を取り σ/\sqrt{n} で与えられます。ただし σ は標本分散から推定するのが通常のため、母平均の推定値として

$$\mu = \bar{x} \pm \frac{s}{\sqrt{n}} \quad (10.11)$$

が得られます。

n が大きいき、 x_i がどのような確率分布にしたがって発生しようとも、 \bar{x} の値は後述する中心極限定理によって近似的に正規分布に従います。したがって、 \bar{x} の誤差として $\frac{s}{\sqrt{n}}$ を与えた場合、これは68%の確率でこの範囲に真の母平均が存在するというを意味します。

10.2.4 大数の法則

繰り返し行うことが可能で、かつ各試行が互いに影響を及ぼさない測定があるとき、その測定を多数回繰り返した際に得られる測定値の平均は、その測定の期待値に近づきます。これを**大数の法則 (law of large numbers)**と言います。例えばサイコロの出目の期待値は常に $\frac{7}{2}$ であるので、サイコロを繰り返し投げたときに出る目の平均はこの値に近づくという、直感的に分かりやすい現象を数学的に証明したものです。

実際にROOTを使ってこの法則を確かめてみます。dice.Cを実行すると、図10.1のように、サイコロの出目の標本平均が試行回数を増やすにつれて $\frac{7}{2}$ に近づいていくのが分かります。

10.2.5 中心極限定理

図10.1では100通りの標本平均の変化を示しましたが、そのばらつきは試行回数を増やすにつれて小さくなっていくのが分かります。またこのばらつきは試行回数 n が大きくなると、正規分布に近づきます。このように、試行回数 n が大きいきに標本平均が正規分布で近似できることを**中心極限定理 (central limit of theorem)**と言います。中心極限定理は測定値の分布がどのような確率分布であっても成り立つことが知られています*8。

図10.2も同じくdice.Cで行ったシミュレーションです。試行回数が1回、10回、...10⁵回の場合の標本平均の分布がどのようになるかを、10000通り試してその分布を示しています。試行回数が1回のときは当然離散的ですが、 n を大きくするにつれて分布が滑らかになり、正規分布へと近づきます。

式10.11では標本平均が母平均の周辺に $\frac{\sigma}{\sqrt{n}}$ の標準偏差を持つ正規分布になることを説明しました。実際、図10.2を見ると $n = 10^5$ のときに標準偏差 $s = 0.005416$ となっており、これはサイコロの出目の標準偏差 $\sigma = 1.70783$ を $\sqrt{100000}$ で割ったものと近くなっています。

*8 ただし、もとの確率分布で分散が定義できない場合、正規分布にならない場合があります。

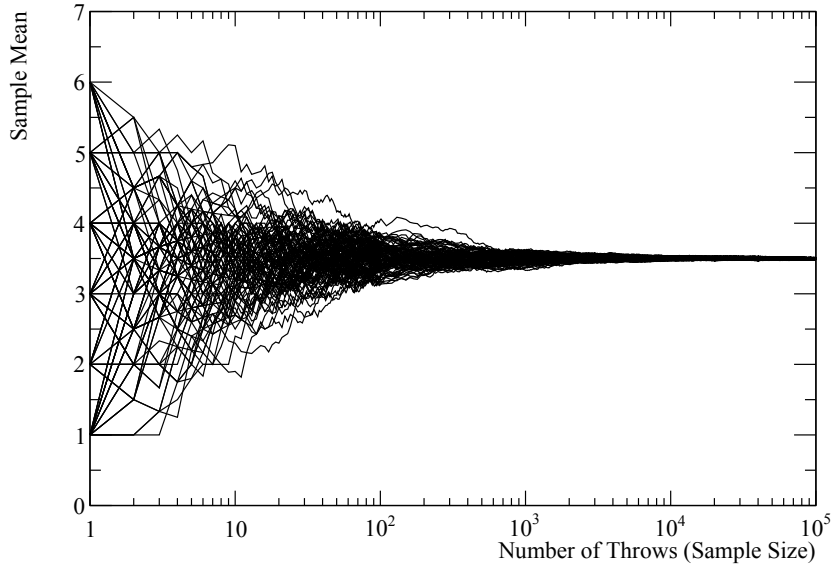


図 10.1 サイコロを振るシミュレーションによる大数の法則の実証例。10⁵ 回の試行を繰り返した場合の標本平均の変化を、100 通り表示したものを。

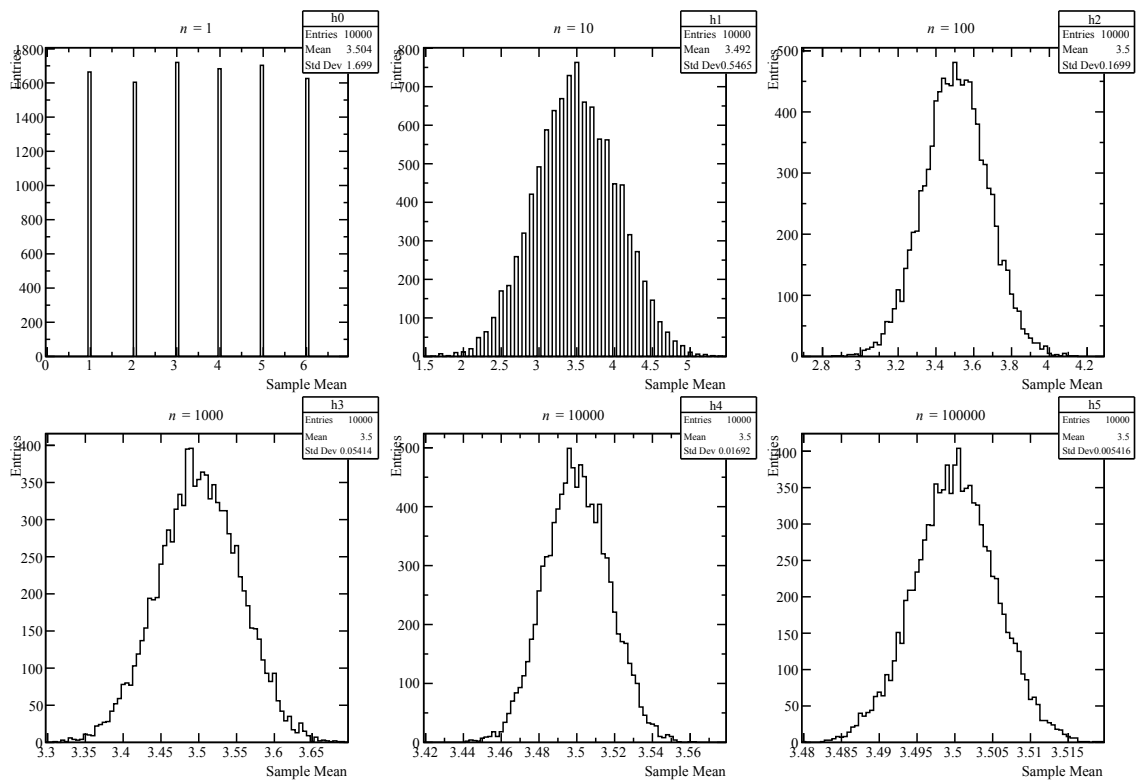


図 10.2 サイコロを振るシミュレーションによる中心極限定理の実証。試行回数 n を大きくするにつれて、標本平均の分布が正規分布に近づく。

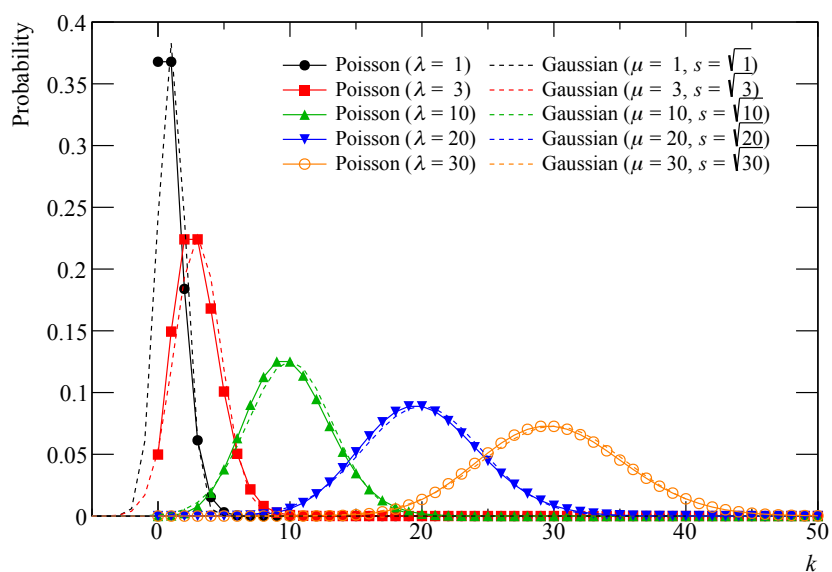


図 10.3 様々な平均値を持つポアソン分布と正規分布の比較

10.3 様々な確率分布

10.3.1 正規分布

10.3.2 ポアソン分布

10.3.3 指数分布

10.3.4 二項分布

10.3.5 カイ二乗分布

コード 10.1 Poisson.C

```
1 void Poisson() {
2     const Int_t kN = 5;
3     const Double_t kLambda[kN] = {1, 3, 10, 20, 30};
4
5     TCanvas* can = new TCanvas;
6     can->DrawFrame(-5, 0, 50, 0.4, "#it{k};Probability");
7
8     TGraph* graphP[kN];
9     TGraph* graphG[kN];
10
11    TLegend* leg = new TLegend(0.3, 0.65, 0.85, 0.85);
12
13    for (Int_t i = 0; i < kN; ++i) {
14        graphP[i] = new TGraph;
15        for (Int_t j = 0; j <= 50; ++j) {
16            graphP[i]->SetPoint(j, j, TMath::PoissonI(j, kLambda[i]));
17        }
18        graphP[i]->Draw("pl same");
19        graphP[i]->SetMarkerColor(i + 1);
20        graphP[i]->SetMarkerStyle(20 + i);
21        graphP[i]->SetLineColor(i + 1);
22    }
23
24    for (Int_t i = 0; i < kN; ++i) {
25        graphG[i] = new TGraph;
26        for (Int_t j = -5; j <= 40; ++j) {
27            Double_t p = (TMath::Erf((j + 0.5 - kLambda[i])/TMath::Sqrt(2*kLambda[i])) +
28                1)/2.
29            - (TMath::Erf((j - 0.5 - kLambda[i])/TMath::Sqrt(2*kLambda[i])) + 1)/2.;
30            graphG[i]->SetPoint(j + 5, j, p);
31        }
32        graphG[i]->Draw("l same");
33        graphG[i]->SetLineStyle(2);
34        graphG[i]->SetLineColor(i + 1);
35    }
36
37    for (Int_t i = 0; i < kN; ++i) {
38        leg->AddEntry(graphP[i], Form("Poisson (#it{#lambda} = %2.0f)", kLambda[i]), "lp
39        ");
40        leg->AddEntry(graphG[i], Form("Gaussian (#it{#mu} = %2.0f, #it{s} = #sqrt{%2.0f
41        })", kLambda[i], kLambda[i]), "l");
42    }
43
44    leg->SetNColumns(2);
45    leg->Draw();
46 }
```

付録 A パッケージ管理システム

macOS や Linux にソフトウェアやライブラリを追加インストールする場合、主に 3 つの方法があります。1 つ目はインストーラの付属したソフトウェアをダウンロードしたり、Apple の App Store を経由してインストールする方法です。例えば LINE のパソコン用アプリケーションなどがこれに該当します。また Python もインストーラを配布していますが、この使用はあまり一般的ではないように思います^{*1}。

2 つ目はパッケージ管理システム (package management system、package manager) を使用する方法です。これは主に、Python などコマンドラインで使用するソフトウェアのインストールや管理に用いられます。パッケージ管理システムは単に必要なソフトウェアやライブラリをインストールするだけでなく、そのソフトウェアに必要な他のライブラリを自動的に選択してインストールしたり、バージョンの依存関係を解決してくれたりします。またソフトウェアごとに異なるウェブサイトに情報が散逸することなく、同一のレポジトリに情報が集約され、インストール方法も共通化されているという利点があります。

Mac では後述する Homebrew が 2019 年現在では主流であり^{*2}、CentOS 7 では Yum が標準です。また Python に特化したパッケージ管理システムとして pip や conda があるのは第 8 章で説明した通りです。本章では、この Homebrew と Yum について簡単に解説します。

3 つ目の方法は CMake や configure を使ってソフトウェアのビルドとインストールをする方法です。この解説は付録 D を参照してください。

A.1 Homebrew

Homebrew は 2019 年現在、Mac でもっとも人気のあるパッケージ管理システムです^{*3*4}。homebrew

まず <https://brew.sh> に記述のある通り、Homebrew をインストールしてみましょう^{*5}。次のコマンドを管理者権限をもつユーザで実行すると

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

色々出てきた後にパスワードを聞かれますので、パスワードを入れて進めてください。

```
Press RETURN to continue or any other key to abort
==> /usr/bin/sudo /bin/mkdir -p /usr/local/Cellar /usr/local/Homebrew /usr/local/Frameworks /usr/local/bin /usr/local/etc /usr/local/include /usr/local/lib /usr/
```

^{*1} <https://www.python.org/downloads/mac-osx/>

^{*2} Mac OS X がリリースされた当初は Fink (<http://www.finkproject.org>) がもっとも人気があり、その後 MacPort (<https://www.macports.org>) に人気に移り、今では Homebrew が主流となりました。

^{*3} <https://brew.sh>

^{*4} 「home brew」は日本語で自家醸造という意味であり、Homebrew に出てくる用語は cask (大樽) や cellar (貯蔵庫) など、酒関係の英語が多いです。

^{*5} Homebrew の更新に従いこのコマンドは変更になる可能性があるため、必ず本家の記述を参照してください。

```
local/opt /usr/local/sbin /usr/local/share /usr/local/share/zsh /usr/local/share
/zsh/site-functions /usr/local/var
Password:
```

そうすると `/usr/local` が作成され、Homebrew 関係のファイルが（ほぼ）全てそこにインストールされるようになります*6。例えば Python 3 を Homebrew でインストールすると `/usr/local/bin/python3` という場所に置かれます。

Homebrew によって `/usr/local` 以下に作られるディレクトリは、管理者権限を持つユーザが読み書きできる設定に変更されます。そのため、最初の Homebrew にインストール以降は原則としてパスワードの入力は求められません。

第 8 章で説明したように、Python 3 のインストールなどを試してみましょう。

```
$ brew install python3
```

他にまず必要なものは CMake と MacTeX*7 だと思います。

```
$ brew install cmake
$ brew cask install mactex
```

A.2 yum

CentOS 7 などの Red Hat Enterprise Linux (RHEL) 派生の Linux ディストリビューション（いわゆる「RHEL クローン」）では、Yum (Yellowdog Updater Modified、ヤム) というパッケージ管理システムが多く使われています。macOS の Homebrew とは違いシステム標準で組み込まれています。そのため CentOS の OS インストール時点で追加パッケージを選択すると、OS インストール後に yum コマンドで追加インストールのは基本的に同じことだと思ってください。

CentOS 7 や RHEL は比較的「枯れた」ソフトウェアを提供する傾向にあります。つまり、あまり新しいソフトウェアのバージョンを取り込まず、実績と安定性を重視して古いバージョンを使用します。そのため Python 3 など新しいソフトウェアは、標準の Yum のレポジトリからは取得できません。そこで、第 8 章で説明したように、場合によってレポジトリを追加する必要があります*8。

```
$ sudo yum install -y https://centos7.iuscommunity.org/ius-release.rpm
```

例えば Python 3 をインストールするには、次のようにします*9。/usr 以下に新たにファイルを追加するため、管理者権限で実行する必要があります。

```
$ sudo yum install python36u-devel
$ which python3.6
/usr/local/python3.6
```

また ROOT のビルドで必要となる CMake は 3.6 以上ですので、cmake3 を入れてください。

```
$ sudo yum install cmake3
```

*6 例えば MacTeX を入れた場合などは、/Library 以下にファイルが置かれる場合があります

*7 Mac に L^AT_EX をインストールする方法はいくつかありますが、Homebrew で MacTeX をインストールするのが一番簡単だと思います。

*8 追加レポジトリは多数ありますので、ここに例として示す <https://centos7.iuscommunity.org/ius-release.rpm> に限りません。

*9 `yum search python3` を実行すると、`python36u-devel` と `python36u` が見つかるはずです。この例のように `-devel` がついているものは、ソフトウェア開発で必要となるヘッダーファイルなども一緒にインストールします。例えば ROOT のビルド時に Python のヘッダーファイルも必要となりますので、`-devel` のほうをインストールするようにしてください。

```
$ which cmake3
/usr/bin/cmake3
$ cmake3 --version
cmake3 version 3.13.4

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

付録 B L^AT_EX 環境の構築

L^AT_EX とは何かということについては、ここでは詳しく説明しません。大雑把には、テキストファイルで作られた文書をそこそこ綺麗な出力で PDF などに自動で組版（くみはん）してくれるソフトウェアです。この文書も L^AT_EX で作られています。

L^AT_EX を知らなくても修士論文は書けますが、宇宙や素粒子関連の業界では、ほとんどの人が L^AT_EX を使って修士論文を書きます（実際には日本語対応の pL^AT_EX）。また同様に投稿論文も L^AT_EX で書くことがほとんどです（実際には PDF の生成が簡単な pdfL^AT_EX）、多くの物理学徒が避けては通れない道具です。実際に修士論文を pL^AT_EX で書く場合の例としては、手前味噌ですが「修士論文 LaTeX テンプレート | 名古屋大学宇宙地球環境研究所の理学系修士学生用」^{*1}が参考になると思います。

L^AT_EX のインストール方法は様々であり、インターネット上の情報もバラバラです。また古い情報も混在しているため、筆者も何が正しいのか、何がお勧めなのかよく分かりません。しかし、ここで示すやり方が「正しそうに思える」ので、ひとつの例として挙げます。この情報は 2019 年 4 月現在のものです。TeXLive などの 2019 年版が出たら、適宜読み替えてください^{*2}。

macOS でも CentOS 7 でも、合計で 5 GB くらいのファイルをインストールします。ダウンロードに 1 時間以上かかったり、パソコンの空き領域を食いつぶしたりするので注意してください。

B.1 macOS での MacTeX のインストール

macOS で LaTeX 環境を構築するには、Homebrew で MacTeX というパッケージを追加するのが簡単です。MacTeX は TeXLive という L^AT_EX デイストリビューションに Mac 固有のアプリケーションなどを追加したものです。例えば Keynote などに数式を入れたいときに便利な LaTeXiT、文献管理ソフトの BibDesk、TeX 統合環境の TeXShop などが /Applications にインストールされます^{*3}。

いつも通り単純に次のコマンド一発でいけます。ただし、brew install ではなく cask というのが間に挟まっているのを注意してください。

```
$ brew cask install mactex
```

次のコマンドを実行した際に

```
$ sudo tlmgr paper
```

もしもページの大きさの設定が次のように letter (21.59 cm × 27.94 cm)^{*4}と表示される場合、

```
Current context paper size (from /usr/local/texlive/2018/texmf-config/tex/context/
user/cont-sys.tex): letter
```

^{*1} <https://github.com/akira-okumura/MasterThesisTemplate>

^{*2} 2019 年 4 月現在、2018 年版が出ています。

^{*3} 筆者はこのうち LaTeXiT と BibDesk のみ常用しています。

^{*4} アメリカで一般的に使われる印刷用紙の大きさです。


```
Current dvipdfmx paper size (from /usr/local/texlive/2018/texmf-config/dvipdfmx/
dvipdfmx.cfg): letter
Current dvips paper size (from /usr/local/texlive/2018/texmf-config/dvips/config/
config.ps): letter
Current pdftex paper size (from /usr/local/texlive/2018/texmf-config/tex/generic/
config/pdftexconfig.tex): letter
Current psutils paper size (from /usr/local/texlive/2018/texmf-config/psutils/paper.
cfg): letter
Current xdvi paper size (from /usr/local/texlive/2018/texmf-config/xdvi/XDvi):
letter
```

次のコマンドで A4 サイズ (21.0 cm × 29.7 cm) に設定を変更してください。

```
$ sudo tlmgr paper a4
```

macOS で L^AT_EX を使う場合、初期設定ではおそらく IPA 明朝と IPA ゴシックというフォントが使用されます。しかし macOS に同梱されているヒラギノフォント (ヒラギノ明朝およびヒラギノ角ゴシック) を使うと PDF の仕上がりが良くなります (可読性の高い綺麗なフォントが埋め込まれます) ので、仕上がりにこだわりたい人はヒラギノフォントを埋め込む設定を行なってください。この作業は macOS のバージョンや TeXLive のバージョンによって異なりますので、<http://doratex.hatenablog.jp/entry/20190327/1553617143>などを参考にしてください。

B.2 CentOS 7 での TeXLive 2018 のインストール

<https://www.tug.org/texlive/quickinstall.html> を参考に、まずはネットワーク越しにインストール^{*5}するためのコマンドをダウンロードし展開します。

```
$ cd
$ curl -O -L http://mirror.ctan.org/systems/texlive/tlnet/install-tl-unx.tar.gz
$ tar zxvf install-tl-unx.tar.gz
$ cd install-tl-20190227/
```

この展開したディレクトリの中にインストーラがあるので、これを管理者権限で実行します。そうすると次のように色々が表示されるので、i を入力してインストールを進めましょう。もしカスタマイズしたい人は、英語の説明に従ってください。

```
$ sudo ./install-tl
Loading ftp://ftp.u-aizu.ac.jp/pub/tex/CTAN/systems/texlive/tlnet/tlpkg/texlive.
tlpdb
Installing TeX Live 2018 from: ftp://ftp.u-aizu.ac.jp/pub/tex/CTAN/systems/texlive/
tlnet (verified)
Platform: x86_64-linux => 'GNU/Linux on x86_64'
Distribution: net (downloading)
Using URL: ftp://ftp.u-aizu.ac.jp/pub/tex/CTAN/systems/texlive/tlnet
Directory for temporary files: /tmp/bDF8hSU8Tp

=====> TeX Live installation procedure <=====
=====> Letters/digits in <angle brackets> indicate <=====
```

^{*5} 必要なファイルをインストーラとともに最初に落としてくるのではなく、インストーラを実行した後に必要ファイルをダウンロードしてくる方法。

```

=====>  menu items for actions or customizations  <=====

Detected platform: GNU/Linux on x86_64

<B> set binary platforms: 1 out of 17

<S> set installation scheme: scheme-full

<C> set installation collections:
    40 collections out of 41, disk space required: 5806 MB

<D> set directories:
    TEXDIR (the main TeX directory):
        /usr/local/texlive/2018
    TEXMFLOCAL (directory for site-wide local files):
        /usr/local/texlive/texmf-local
    TEXMFSYSVAR (directory for variable and automatically generated data):
        /usr/local/texlive/2018/texmf-var
    TEXMFSYSCONFIG (directory for local config):
        /usr/local/texlive/2018/texmf-config
    TEXMFVAR (personal directory for variable and automatically generated data):
        ~/.texlive2018/texmf-var
    TEXMFCONFIG (personal directory for local config):
        ~/.texlive2018/texmf-config
    TEXMFHOME (directory for user-specific files):
        ~/texmf

<O> options:
    [ ] use letter size instead of A4 by default
    [X] allow execution of restricted list of programs via \write18
    [X] create all format files
    [X] install macro/font doc tree
    [X] install macro/font source tree
    [ ] create symlinks to standard directories

<V> set up for portable installation

Actions:
<I> start installation to hard disk
<P> save installation profile to 'texlive.profile' and exit
<H> help
<Q> quit

Enter command: i

```

かなり時間がかかりますが、必要なファイルを全てダウンロードし終わると、最後に環境変数を設定するように指示が出ます。

```

Installing to: /usr/local/texlive/2018
Installing [0001/3752, time/total: ??:??/?:?]: 12many [376k]
Installing [0002/3752, time/total: 00:04/08:25:18]: 2up [66k]
Installing [0003/3752, time/total: 00:05/08:57:54]: Asana-Math [482k]

```

```
Installing [0004/3752, time/total: 00:07/05:59:39]: ESIEEcvt [137k] (略)

Add /usr/local/texlive/2018/texmf-dist/doc/man to MANPATH.
Add /usr/local/texlive/2018/texmf-dist/doc/info to INFOPATH.
Most importantly, add /usr/local/texlive/2018/bin/x86_64-linux
to your PATH for current and future sessions.
```

これに従い、`/.bashrc` や `/.zshrc` に次の 3 行を追加します^{*6}。

```
export MANPATH=/usr/local/texlive/2018/texmf-dist/doc/man:$MANPATH
export INFOPATH=/usr/local/texlive/2018/texmf-dist/doc/info:$INFOPATH
export PATH=/usr/local/texlive/2018/bin/x86_64-linux:$PATH
```

B.3 動作確認

新しいターミナルのウィンドウを開き、次のコマンドで `RHEA.pdf` (この文書) が生成できるかを確認しましょう。

```
$ git clone https://github.com/akira-okumura/RHEA.git
$ cd RHEA
$ ls
Makefile  RHEA.bib  RHEA.tex  misc      tex
README.md RHEA.bst  fig       src
$ make
$ ls
Makefile  RHEA.bbl  RHEA.bst  RHEA.out  RHEA.toc  src
README.md RHEA.bib  RHEA.dvi  RHEA.pdf  fig       tex
RHEA.aux  RHEA.blg  RHEA.log  RHEA.tex  misc
$ open RHEA.pdf # macOS の場合
$ xdg-open RHEA.pdf # Linux の場合
```

^{*6} インストールの年によって 2018 が 2019 に変わっていたりする可能性があるので、実際の出力に従ってください。

付録 C Mac での研究環境の構築

※この説明はかなり古い情報になってしまっています。そのうち書き直すので、あまり参考にしないでください。

ここでは、Mac で研究環境を構築する際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。この章での説明は、OS X Lion を前提に書かれており、また使用するスクリーンショットも Lion 環境のもので、Snow Leopard では多少異なる可能性があるので注意してください。

C.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。日本語環境の場合だとこうはいきません。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 Adobe 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

*1 L^AT_EX 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 C.1 「システム環境設定」から「言語とテキスト」を開く

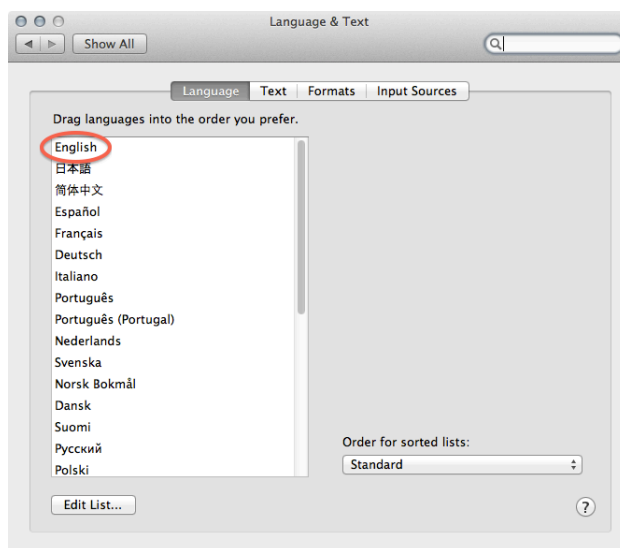


図 C.2 「日本語」ではなく「English」を先頭に持ってくる（この画面は既に英語環境になっている場合のもの）

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。

図 C.1 から図 C.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が、図 C.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。

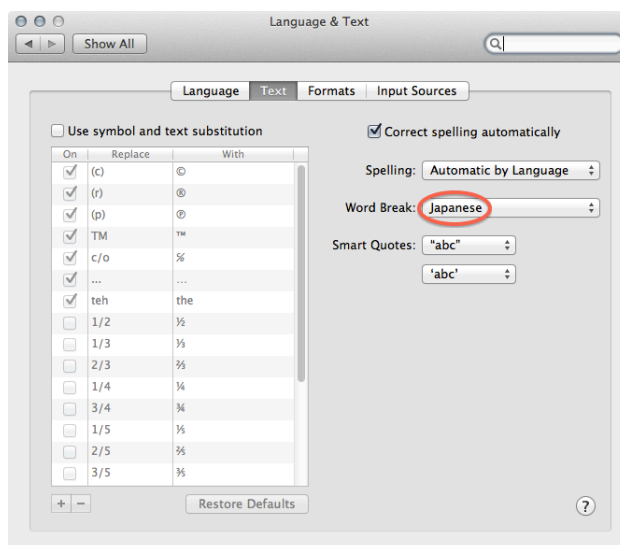


図 C.3 「単語区切り (Word Break)」を「Japanese」にする (この画面は既に英語環境になっている場合のもの)



図 C.4 「Show all filename extensions」にチェックを入れると、全てのファイルで拡張子が表示されるようになる

C.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 C.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。

C.3 キーボードの設定

C.3.1 Caps Lock キーを Control キーに変更する

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 C.5 のように、System Preferences から「Keyboard」を開いて「Keyboard」タブの「Modifier keys...」を押し、Caps Lock を Control にします。後述するように、macOS 環境では Emacs の操作体系に倣ったキーボードショートカットが使われています。そのため、Control キーの使用率が他の OS に比べて非常に高くなるのが特徴です。US 配列のキーボードを使用している場合には Control キーが押しにくい位置にあるため、滅多に使わない、かつ最も押しやすい位置にある Caps Lock を Control キーにしてしまうことがよく行われています。

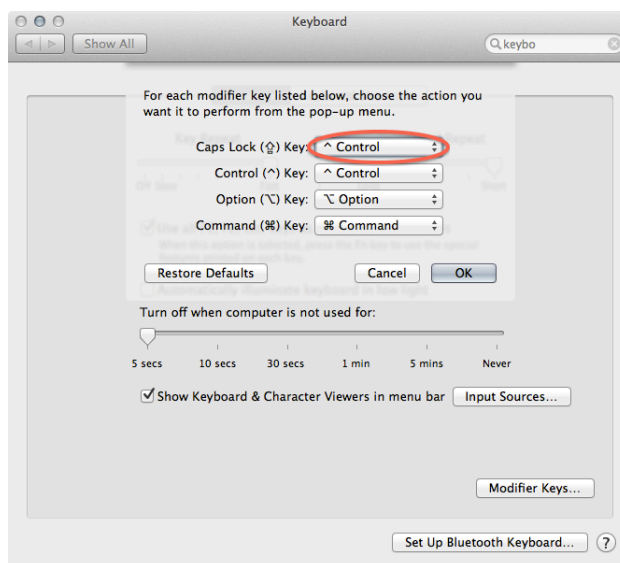


図 C.5 Caps Lock を Control キーに変更する

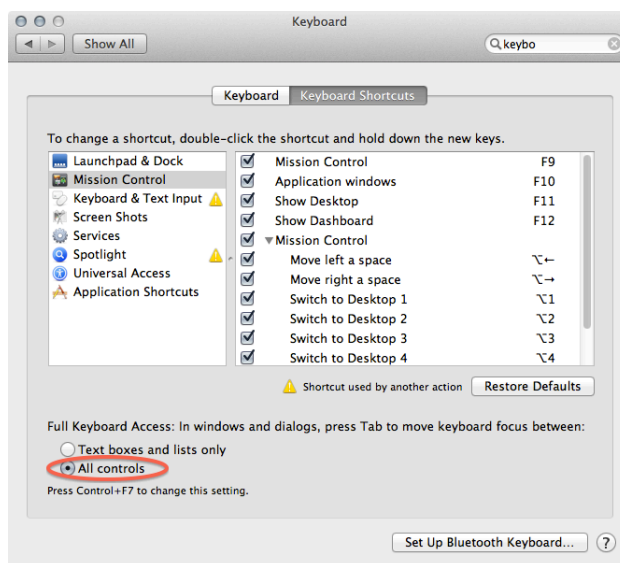


図 C.6 全てのボタンなどを tab キーで移動できるようにする

C.3.2 Tab キーの動作の変更

次に、図 C.6 のように「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。

例えば、Mac の電源ボタンを軽く一度だけ押すと、図 C.7 のような確認ダイアログが出てくるはずですが、ここで数回 Tab キーを押すと、「Cancel」ボタンが強調表示されるようになります。また「Shut Down」ボタンが明滅しているでしょう。この状態で Space キーを押すと、マウスやトラックパッドの操作をしなくても強調表示された「Cancel」ボタンが押されます。また Return キーを押すと明滅している「Shut Down」ボタンが押されて Mac が再起動します。このように、Mac ではダイアログのボタンをクリックする代わりに Space キーと Return キーで操作することができます。

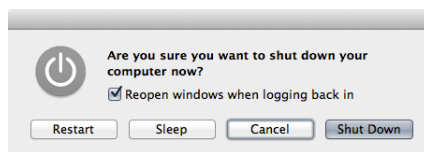


図 C.7 再起動の確認ダイアログ。Tab を数回押すと、「Cancel」が強調表示される。「Shut Down」は明滅している。

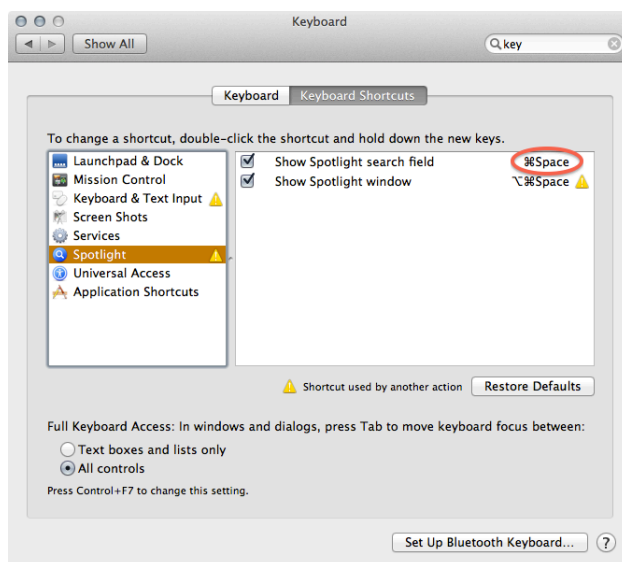


図 C.8 Spotlight のショートカットを「Control-Space」から「Command-Space」に変更した状態

C.3.3 Spotlight のショートカットの変更

最後に、Spotlight のショートカットを図 C.8 のように Control-Space（画面上の表示は「^Space」）の組み合わせから Command-Space（画面上の表示は「⌘ Space」）などの組み合わせに変更しましょう。Control-Space は後述するように Emacs で頻繁に使用するショートカットであるため、Spotlight のショートカットとして使用されてしまうと不便だからです。

Command-Space を Spotlight にあてがってしまうと、文字入力の日本語と英語の切り替えはどうするのかと、Mac に少し慣れた人なら疑問に思うかもしれません。しかし最近の Mac には JIS 配列の場合「かな」と「英数」キーが存在するため、これらのキーを使ったほうが文字入力の切り替えは簡単に行えます。また US 配列の Mac を使っている場合には、C.4 にて後述する KeyRemap4MacBook を使うことで「かな」と「英数」を Command キーに割り当てるのが可能になるため、やはり Command-Space のショートカットは不要です。

C.4 KeyRemap4MacBook

C.7 で述べるように、macOS のキーボード操作の多くは Emacs の操作体系に基づいています。しかし、この操作に対応していないアプリケーションも存在するため（Microsoft 製品、Adobe 製品、Eclipse など）、KeyRemap4MacBook^{*3} をインストールすることで macOS の操作を快適にすることができます。

KeyRemap4MacBook をインストール後、「System Preferences」から「KeyRemap4MacBook」の「Change Key」タ

^{*3} <http://pqrs.org/macosx/keyremap4macbook/index.html.ja>

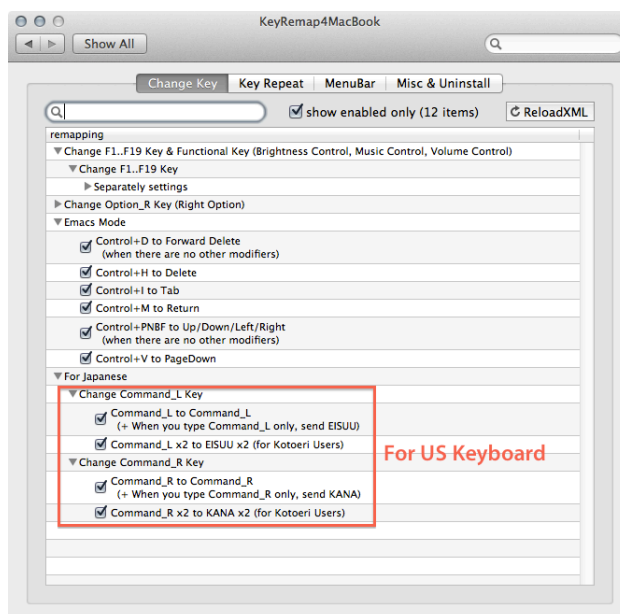


図 C.9 macOS のほとんどの操作で、Emacs に似た操作体系を実現するための設定の例。US キーボードの使用者は、赤枠内の設定も追加することで日本語入力が楽になる。

ブを開くと、図 C.9 のような画面が現われます。大量の設定項目が存在しますが、そのうち、「Control+D to Forward Delete」、「Control+H to Delete」、「Control-I to Tab」、「Control+M to Return」、「Control+PNBF to Up/Down/Left/Right」、「Control+V to PageDown」などを動作させると良いでしょう。

また、JIS 配列のキーボードではなく、US 配列のものを使用している人の場合、「Command_L to Command_L」、「Command_L x2 to EISUU x2」、「Command_R to Command_R」、「Command_R x2 to KANA x2」を動作させておけば、左右の Command キーを英数キー、かなキーとして使用することが可能になります。もちろん、他のキーと組み合わせた場合には Command キーとしても動作します。

次に、「Key Repeat」タブを開くと「[Key Repeat] Wait」という設定項目があります。ここの時間を短くすることで、キーを押下し続けたときに繰り返し入力される時間間隔を短くすることが可能です。あなたが Mac や Emacs の操作に熟達する頃には、デフォルトの 80 ミリ秒では遅過ぎると感じるようになってはいるはずです。

C.5 zsh

macOS の標準の shell 環境は、Snow Leopard や Lion では bash になっています。bash は Linux や macOS で広く使われており、ソフトウェアの一部や OS の設定スクリプトなどで見ることができます。csh や tcsh を使う人も少なくない数が存在しますが、2012 年現在で Mac を研究に使うには bash を選択するのが標準的です。

一方で、bash よりも更に高機能な shell として最近では zsh の人気が高まっています。zsh は bash の文法に非常に似ているものの、独自の拡張機能をたくさん持っているため、bash に慣れている人には「便利な bash」として使うことが可能です。また、初学者の場合は bash と zsh の違いは全く気にする必要はないでしょうから、bash ではなく zsh を使い始めるのが良いでしょう。

図 C.11 と図 C.12 に、shell を zsh に変更する方法を説明します。まず図 C.11 のように、「System Preferences」の「Users & Groups」を表示し、自分のアカウントを右クリック（トラックパッドの場合は二本指クリック）します。「Advanced Options...」が現われるので、これを選択します。ウィンドウ左下の鍵マークがロックされている状態の場合は右クリックが効きませんので、パスワードを入力してこれを解除して下さい。

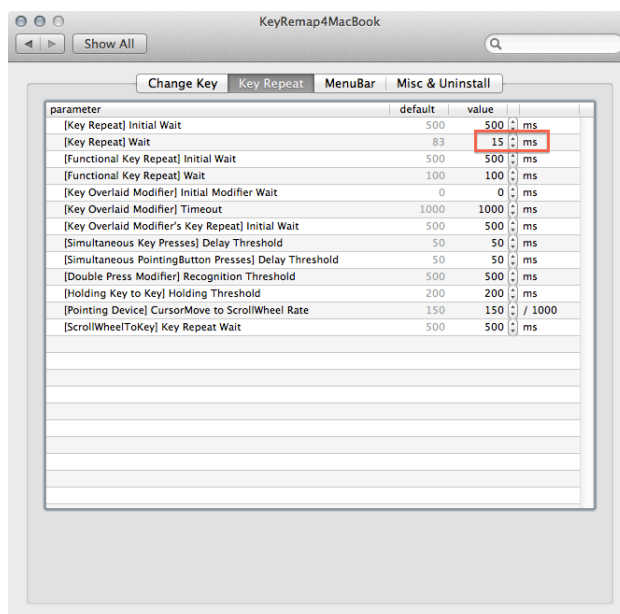


図 C.10 キーボードを押下し続けた場合の繰り返し入力の設定

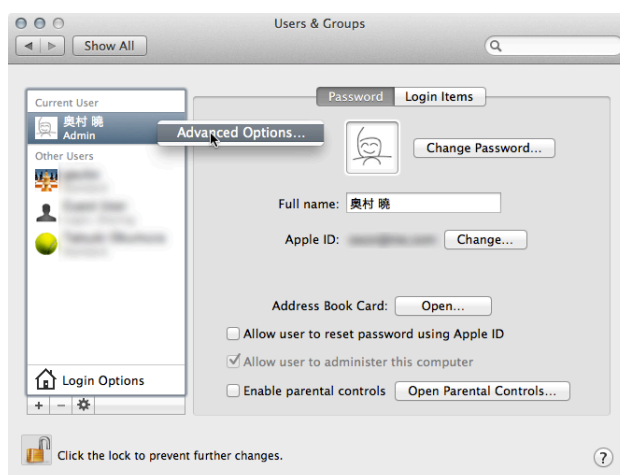


図 C.11 自分のアカウントを右クリックして「Advanced Options...」を表示する

図 C.12 が表示されるので、「Login shell」を「/bin/bash」から「/bin/zsh」に書き換えて下さい。ここで、絶対に打ち間違いをしないように注意してください。これで、次に Terminal を開くときから、ログイン shell が zsh に変更されます。

bash から zsh に変更すると、各種の設定は \$HOME/.bashrc ではなく \$HOME/.zshrc に書くようになります。例えば次の設定を \$HOME/.bashrc に書くと指示があった場合、\$HOME/.zshrc にこれを書いて下さい。

```
export ROOTSYS=/usr/local/root
```

zsh の設定や \$HOME/.zshrc の書きかたについては、「漢の zsh^{*4}」というコラムの解説が非常に詳しく分かりやすいので、参考にして下さい。

*4 <http://news.mynavi.jp/column/zsh/index.html>



図 C.12 「/bin/bash」を「/bin/zsh」に書き換える

C.6 MacPorts

MacPorts^{*5}とは、UNIX や Linux で広く使われ、また開発の行われてきたソフトウェア資産を、Mac でも使えるようにするためのプロジェクトのひとつです。GUI ソフトウェアでもそうですが、他の環境で動作するソフトウェアだからと言って、Mac でも簡単に動くわけではありません。Mac 以外の UNIX に書かれたソフトウェアであっても、Mac 用にコンパイルしたり、各種設定を変更してやる必要があります。しかし MacPorts のように Mac に移植するプロジェクトの成果を利用することで、難しい作業を一般ユーザが行う必要がなくなります。

同様のプロジェクトに、Fink^{*6}や Homebrew^{*7}があります。macOS のリリース初期では Fink が最も活発でしたが、ROOT と組み合わせた場合に問題が何度か発生したため、筆者は MacPorts に移行しました。最近では Homebrew が流行っていますが、MacPorts に比べるとまだユーザ数は多くありません。新しもの好きの人は Homebrew を検討してみてください。

C.6.1 MacPorts の導入

MacPorts をダウンロードし、Mac の他のソフトウェアと同様にインストーラを使ってインストールします。MacPorts 本体と、関連するコマンド、ライブラリ、ヘッダーファイルなどは全て /opt にインストールされます。以下の設定を \$HOME/.zshrc に追加して下さい。

```
export PATH=/opt/local/bin:$PATH
export LD_LIBRARY_PATH=/opt/local/lib:$LD_LIBRARY_PATH
export C_INCLUDE_PATH=/opt/local/inc:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=/opt/local/inc:$C_INCLUDE_PATH
```

Terminal から、次のコマンドを実行して \$HOME/.zshrc の設定を反映しましょう。新しい Terminal を開いた場合は、いちいち実行する必要はありません。

*5 <http://www.macports.org/>

*6 <http://www.finkproject.org/>

*7 <http://mxcl.github.com/homebrew/>

```
$ source ~/.zshrc
```

MacPorts で好きなソフトウェアをインストールするには、いくつかコマンドを実行する必要があります。Emacs の導入事例を C.7 で説明します。

C.6.2 Emacs

まず、MacPorts で Emacs をインストールできるかどうか調べてみましょう。次のコマンドで説明文中に「emacs」の含まれるソフトウェア (port) があるか確認します。

```
$ port search emacs
```

そうすると、たくさんの出力が次のように表示されるはずです。これを細かく見ていくと、「GNU Emacs」という文字列を含んだ port がいくつか存在するはずです。

```
auctex @11.86 (editors, print)
  A major emacs mode for editing TeX files.
(snip)
emacs @23.4 (editors)
  The GNU Emacs text editor

emacs-app @23.4 (aqua, editors)
  The GNU Emacs text editor (Cocoa version)

emacs-app-devel @20091101 (aqua, editors)
  The GNU Emacs text editor, recent CVS development version

emacs-snapshot @20120423 (editors)
  The GNU Emacs text editor

emacs-w3m @1.4.4 (www)
  Use the w3m web browser inside emacs.

emacs22 @22.3 (editors)
  The GNU Emacs text editor
(snip)
zile @2.3.23 (editors)
  Zile Is Lossy Emacs

Found 38 ports.
```

「emacs @23.4 (editors)」というのが、バージョン 23.4 の Emacs だと検討がつくので、さらに次のコマンドで詳細を見てみましょう。

```
$ port info emacs
```

次のように、説明が何行も表示されます。

```
emacs @23.4, Revision 1 (editors)
Variants:          dbus, gtk, motif, universal, x11

Description:      GNU Emacs is a self-documenting, customizable, extensible
```

```

real-time display editor. Users new to Emacs will be able
to use basic features fairly rapidly by studying the
tutorial and using the self-documentation features. Emacs
also has an extensive interactive manual browser. It is
easily extensible since its editing commands are written
in Lisp.
Homepage:      http://www.gnu.org/software/emacs/emacs.html

Build Dependencies:  pkgconfig, texinfo
Library Dependencies: ncurses
Conflicts with:     xemacs
Platforms:         darwin, freebsd
License:          GPL-3+
Maintainers:      dports@macports.org, openmaintainer@macports.org

```

実は、macOS には最初から Emacs がインストールしてあるため、必ずしも Emacs を自分でインストールする必要はありません。しかし、macOS に含まれる Emacs は X11 に対応していないため、Terminal の中でしか Emacs を立ち上げることはできません。一方、MacPorts の場合には X11 に対応した Emacs をインストールすることが可能です。研究をする上では、こちらのほうが便利なので MacPorts で Emacs を導入しましょう。

ここで、「Variants:」と書かれた説明を見ると、「x11」とあるのが分かります。Emacs を MacPorts でインストールするときに、どのようなオプションが選択可能かを示しており、この場合は「x11」というオプションを指定すれば X11 に対応した Emacs がインストールされることを示しています。

次のコマンドを実行し、X11 に対応した Emacs をインストールしましょう。管理者パスワードを要求されます。

```
$ sudo port install emacs +x11
```

インストールが開始されると、Emacs を使うために必要な他の port もインストールされます。以下のような出力が表示されれば、Emacs を使えるようになっているはずです。

```

---> Fetching archive for zlib
---> Attempting to fetch zlib-1.2.7_0.darwin_11.x86_64.tbz2 from http://packages.
      macports.org/zlib
---> Attempting to fetch zlib-1.2.7_0.darwin_11.x86_64.tbz2.rmd160 from http://
      packages.macports.org/zlib
---> Installing zlib @1.2.7_0
---> Cleaning zlib
(snip)
---> Computing dependencies for emacs
---> Fetching archive for emacs
---> Attempting to fetch emacs-23.4_1.darwin_11.x86_64.tbz2 from http://packages.
      macports.org/emacs
---> Attempting to fetch emacs-23.4_1.darwin_11.x86_64.tbz2.rmd160 from http://
      packages.macports.org/emacs
---> Installing emacs @23.4_1
---> Activating emacs @23.4_1

D-Bus support is no longer included in the default Emacs installation. To build
Emacs with D-Bus support, please install with the +dbus variant.

---> Cleaning emacs

```

Lion で Xcode 4.3 を使っている場合に次のようなエラーが表示されるかもしれません。

```
Error:
Error: No valid Xcode installation is properly selected.
Error: Please use xcode-select to select an Xcode installation:
Error:     sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer #
        version 4.3.2
Error:
Warning: xcodebuild exists but failed to execute
Warning: Xcode does not appear to be installed; most ports will likely fail to build
.
```

そのときは、エラーの指示通りに従い、次のコマンドを実行しましょう。これで MacPorts が動作するようになるはずです。

```
$ sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer
```

C.6.3 L^AT_EX

研究で誰しも必要になるのが L^AT_EX の環境です。様々なインストール方法があり、インターネット上にも色々な情報が散らばっています。MacPorts から L^AT_EX を入れる場合でも、ptex、tetex、texlive という三つの port が存在します。

```
$ port info ptex
pTeX @20110314, Revision 1 (tex, print, textproc, japanese)
Variants:          euc, [+]motif, nextaw, no_hiragino, no_otf, no_x11, sjis,
                  [+]utf8, xaw, xaw3d

Description:      Japanese TeX (pTeX) processing environment
Homepage:         http://www.nn.iij4u.or.jp/~tutimura/tex/ptetex.html

Build Dependencies:  ghostscript-fonts-hiragino, nkf
Library Dependencies: gd2, jpeg, libiconv, libpaper, libpng, ncurses, perl5,
                  t1lib, zlib, openmotif
Runtime Dependencies: ghostscript-fonts-hiragino, tlutils, texi2html, texinfo
Conflicts with:    texlive-common
Platforms:        darwin
License:          Restrictive/Distributable
Maintainers:      takanori@macports.org, openmaintainer@macports.org
```

Mac で L^AT_EX を使う場合、文字コードを UTF-8 にするのが標準的です。ptex では特に指定しない限りは UTF-8 に対応したものがインストールされるため、「Variants:」の説明に「[+]utf8」と書いてあります。

次のコマンドで ptex を入れましょう。platex、pdflatex、dvipdfmx などのコマンドと一緒にインストールされます。好みに応じて、代わりに texlive や tetex を導入するのも良いでしょう*8。

```
$ sudo port install ptex
```

*8 違いが何なのかは筆者は理解していません。

表 C.1 Emacs の基本的なキーボードショートカット

ショートカット	英単語	機能
C-f	F orward	一文字進む (→キーと同等)
C-b	B ackward	一文字戻る (←キーと同等)
C-p	P revious line	次の行へ進む (↓キーと同等)
C-n	N ext line	前の行へ戻る (↑キーと同等)
C-a	A head	行頭へ移動
C-e	E nd	行末へ移動
C-k	K ill	カーソル位置から行末までを全て kill-ring と呼ばれるメモリ領域に移動 (カットに類似)
C-y	Y ank	kill-ring にあるものをカーソル位置にペースト
C-t	T ranspose	カーソルの前後の文字を入れ替える
C-h	不明	カーソル前方を一文字削除 (Mac の Delete、Windows の Backspace と同等)
C-d	forward D elete	カーソル後方を一文字削除 (Windows の Delete と同等)
C-m	不明	改行 (カーソルも次の行へ移動する)
C-o	不明	改行 (カーソルは元の位置にとどまる)
C-Space C-w	不明	C-Space 入力時のカーソル位置から、C-w 入力時のカーソル位置までの範囲を kill-ring に移動する
C-x C-f	F ind file	既存ファイルを開く、なければ作成する
C-x C-s	S ave	ファイルを保存する
C-x C-c	不明	終了する
C-x C-w	不明	ファイルを別名で保存する
C-x C-2	不明	ウインドウを二分割にする
C-x C-1	不明	ウインドウを一分割にする
C-s	S earch	検索する
C-r	R verse search	後方に検索する
M gg	不明	特定の行に移動する

C.7 Emacs の操作体系に慣れる

ここでは、macOS での標準的なエディタ (editor) のひとつである Emacs の操作について、その初歩の初歩について説明します。エディタは文章を書くためのソフトウェアです。ただし、ワープロのように様々な装飾をするための機能はありません。プログラムを書いたり、L^AT_EX のソースを書いたり、装飾ではなく文そのものに意味があるときに使います。そのような作業に特化したワープロにはない様々な機能が搭載されています。

Emacs の特徴のひとつに、文字入力に特殊なキーボードショートカット (Emacs key binding) を使うことが挙げられます。慣れないうちは全く便利に感じないと思いますが、一度慣れてしまうと、もう他のエディタには戻れなくなるのでぜひ覚えて下さい。macOS を使う場合、このショートカットをほとんどのソフトウェアでも利用可能です。例えば、macOS に標準で付属しているメールクライアント「Mail」でメッセージを編集する時にも、このショートカットが使用できます。基本的なキーボードショートカットの一覧を表 C.1 にまとめました。

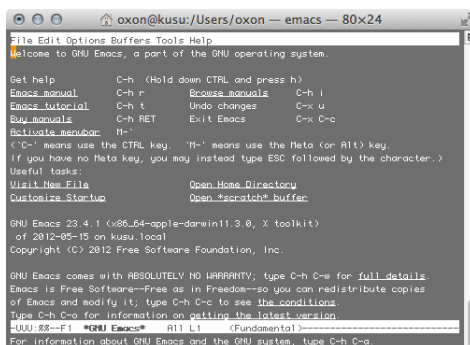


図 C.13 X11 対応でない Emacs の起動例

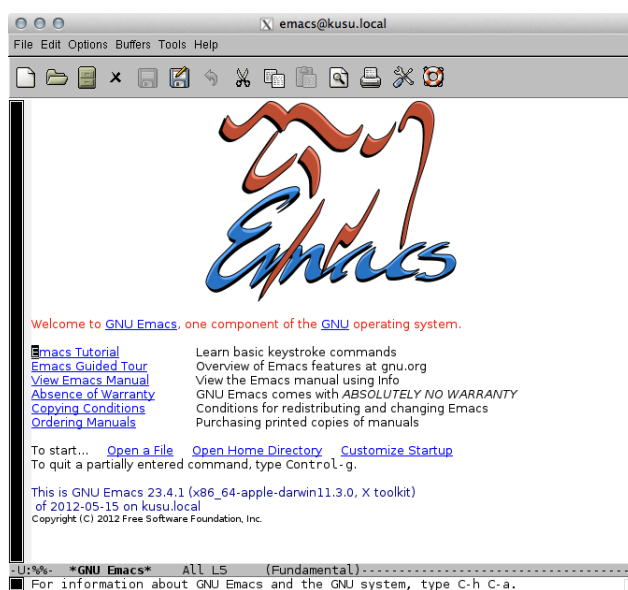


図 C.14 X11 対応の Emacs の起動例

まずは Emacs を次のコマンドで起動してみましょう。MacPorts から Emacs を入れていなかったり、MacPorts で「+x11」をつけずに Emacs をインストールした場合には図 C.13 のような表示が現われるはず*9。

```
$ emacs
```

X11 に対応した Emacs を起動した場合は、図 C.14 のようなウィンドウが表示されます。まずは、新しいファイルを作成してみましょう。Control キーを左手小指で押下し、左手中指で X キーを同時に押します。その後、Control キーを押した状態のまま左手人さし指で F キーを同時に押します。これを、「C-x C-f」のように表記します*10。表 C.1 にあるように、これは新規ファイルを作成するためのショートカットです。図 C.15 のように「Find file:」の後に続けて作成したいファイル名を入力しましょう。図 C.15 では /foo.txt を作成しています。

あらたなファイルが作成されたので（保存はされていません）、図 C.16 のように文字を入力しましょう。この後、C-x C-s を入力すると実際にディスク上に保存されます。図 C.16 の下部に「Wrote /Users/oxon/foo.txt」と表示されます。最後に、C-x C-c と入力して Emacs を終了しましょう。

*9 X11 に対応した Emacs でも、「-nw」という引数をつけて Emacs を起動すれば、同様の表示にすることが可能です。接続速度の遅い（つまり画像を飛ばすのが遅い）SSH の接続先などで作業する場合に使うこともあります。

*10 Control-x、Ctl-x、Cn-x など、いくつかの表記方法があります。好みの問題です。

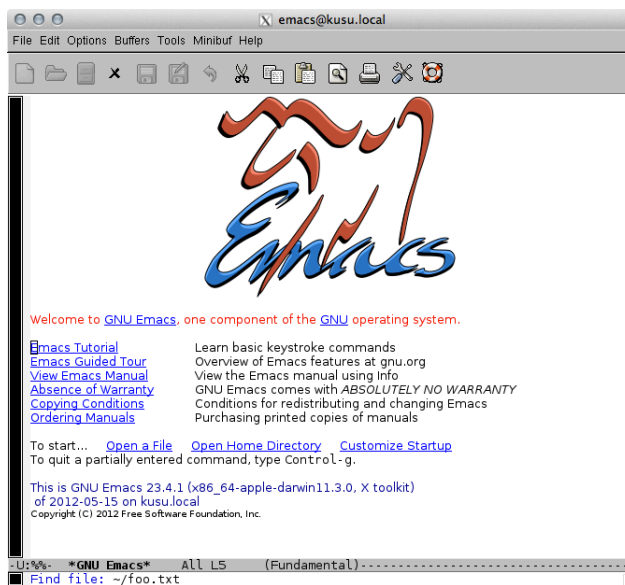


図 C.15 Emacs で新規ファイルを作成する

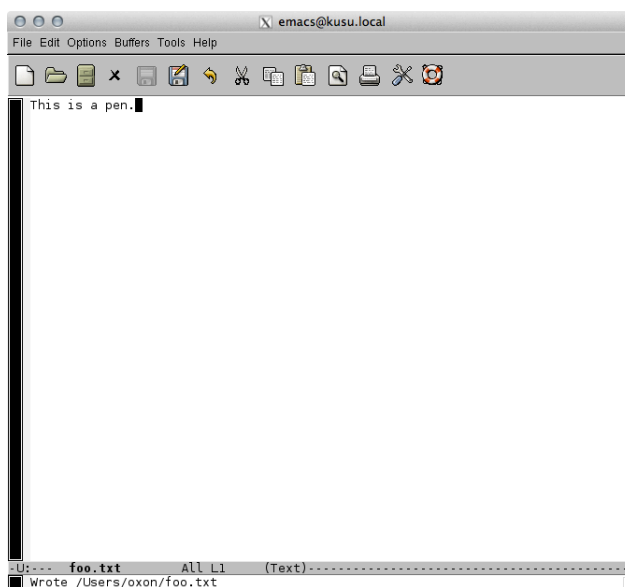


図 C.16 Emacs で文字入力をし、保存する

Emacs の操作に慣れるには、まずこの Control キーを多用した操作に慣れることが重要です。表 C.1 に挙げたショートカットを中心にまずは覚え、どのような操作や機能、設定があるのかを調べていきましょう。多機能過ぎて、うんざりするはずです。

C.8 C-Space C-w を macOS で使う

表 C.1 に示したショートカットの多くは macOS のアプリケーションでも使うことができます。また使えないアプリケーションでも、C.4 に書いたように KeyRemap4MacBook の設定で使用可能になります。

しかし、デフォルトの状態では C-Space C-w は macOS のアプリケーションでは使うことができません。Emacs に慣

コード C.1 DefaultKeyBinding.dict

```

1 {
2     "^x" = {
3         "^s" = "save:";           /* C-x C-s */
4         "^w" = "saveAs:";        /* C-x C-w */
5     };
6
7     "^ " = "setMark:";           /* C-space */
8     "^w" = "deleteToMark:";     /* C-w */
9 }

```

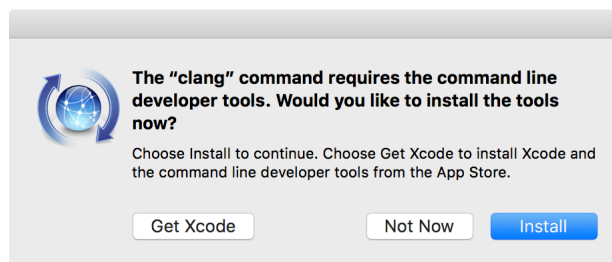


図 C.17 Command Line Developer Tools のインストールダイアログ

れてくると、C-Space C-w が Emacs 以外で使えないのが不便に感じるでしょう。macOS にはこの操作を実現するための機能が隠されています。

コード C.1 を、~/Library/KeyBindings/DefaultKeyBinding.dict として保存して下さい^{*11}。「Mail」などを起動して文字入力する際に、C-Space C-w や C-x C-s が有効になっているはず^{*12}。

C.9 Xcode

```

$ clang
xcodeselect: note: no developer tools were found at '/Applications/Xcode.app',
  requesting install. Choose an option in the dialog to download the command line
  developer tools.

```

```

$ clang
clang: error: no input files

```

C.10 知っている便利な macOS 特有のコマンド

^{*11} <http://akenox.dip.jp/index.php?Translations%2FMac%20OS%20X%20Key%20Bindings> に詳しい解説がある。

^{*12} Lion からは「Save As...」が「Duplicate」などに置き換わってしまったため、コード C.1 の設定をしても C-x C-w は動作しません。

付録 D configure と Make

D.1 configure を使った ROOT のコンパイル (非推奨)

最近の ROOT では非推奨もしくは全くサポートされていません^{*1}が、configure スクリプトと make コマンドを使った ROOT のビルドは、かつて標準的な手法でした。また同様の手法で多くのオープンソースソフトウェアをビルドすることもできますので、ここでは参考として configure と make を使ったビルド方法を紹介します。

まずは CMake の時と同様に ROOT のソースコードを /usr/local に展開します。

```
$ cd /usr/local
$ sudo tar zxvf ~/root_v6.12.06.source.tar.gz
```

次に configure スクリプトを実行します。このスクリプトは CMake や CmakeLists.txt と同様に、ソフトウェアのビルドに必要な情報を集約します。

```
$ cd root-6.12.06
$ sudo ./configure
```

色々とターミナルに出力されますが、最後に次のような表示が出れば configure は成功です。もし何かしらのエラーが表示される場合、ROOT のビルドに必要な他のコマンドやライブラリがインストールされていないことが考えられます。表示されたエラーをよく読み、原因が何かを考えて対応しましょう。

```
Enabled support for asimage, astiff, bonjour, builtin_afterimage, builtin_ftgl,
  builtin_freetype, builtin_gl2ps, builtin_glew, builtin_unuran, builtin_lz4,
  builtin_llvm, libcxx, cocoa, explicitlink, fink, fitsio, gviz, genvector, krb5,
  ldap, memstat, opengl, python, rpath, search_usrlocal, shared, sqlite, tmva, xml
.
```

To build ROOT type:

```
make
```

```
*****
* The classic configure/make method of building ROOT is now DEPRECATED in *
* favor of the CMake one. *
* Refer to README/INSTALL file for updated instructions or visit the page *
* https://root.cern.ch/building-root *
*****
```

6.12/06 では、configure を使ったビルドは「DEPRECATED」(旧式で非推奨) であると表示されます。新しい ROOT を使う場合は、節 2.1.4 の CMake を使ったビルドを推奨します。

^{*1} 少なくとも 6.16/00 以降ではサポートされていません。

あとは、CMake のときと同様に make を実行します。

```
$ sudo make -j8
```

うまくコンパイルが全て完了すると、次のような表示が最後に出るはずですが。

```
root [0]
Processing hsimple.C...
hsimple : Real Time = 0.07 seconds Cpu Time = 0.05 seconds
(TFile *) 0x7fb0fae560c0

=====
===          ROOT BUILD SUCCESSFUL.          ===
=== Run 'source bin/thisroot.[c]sh' before starting ROOT ===
=====
```

make 実行時に何もオプションを渡さない場合、発展的な機能やマイナーな機能は「Enabled support」の一覧に入りません。make 実行時に例えば次のようにオプションをつけることで、オプションもビルドすることができるようになります。

```
$ sudo ./configure --enable-minuit2 --enable-gdml
```

configure を使った場合は展開したソースファイルのディレクトリ内でコンパイルを行いますので、次のような `/.bashrc` の設定になります。

```
cd /usr/local/root-6.12.06
source bin/thisroot.sh
cd - > /dev/null
```

D.2 configure と Make の一般的な解説

macOS や Linux 環境でソフトウェアを導入する時、インストールの方法は主に3つあります。1つ目は、C.6 で説明したようなパッケージ管理ソフト^{*2}を使う方法です。2つ目は、既にコンパイル済みでバイナリ配布されているソフトを導入する方法です。例えば、KeyRemap4MacBook をインストーラを使ってインストールしたのがこれに当たります。また、ROOT を入れる場合でもバイナリ配布されているものを使うことが可能です。3つ目は、ソースコードの配布されているソフトウェアを自分でコンパイル (compile) する^{*3}方法です。これは 2.1 で ROOT の導入として説明しました。

自分でソフトウェアをソースコードからコンパイルする場合、標準的なやり方が存在します。また、研究で通常使うソフトウェアの多くがこの標準的な方法に則して作成されています。そのためここで説明する方法を覚えておけば、多くのソフトウェアを自分で簡単にインストールできるようになります。

このやり方を覚えるために、「GNU Hello^{*4}」と呼ばれるソフトウェアを試しにインストールしてみましょう。この「GNU Hello」は標準的な方法を学ぶ上で最適なもので、インストールしても実用性はありません。

まず、次のコマンドで <http://ftp.gnu.org/gnu/hello/> から最新版の `hello-2.8.tar.gz` を落としてきて展開します。展開されたディレクトリに移動して `ls` で中身を見てみましょう。

^{*2} macOS では MacPorts、Fink、Homebrew など、Linux では Yum や APT など。

^{*3} ビルド (build) するとも言います。

^{*4} <http://www.gnu.org/software/hello/>

```

$ curl -O http://ftp.gnu.org/gnu/hello/hello-2.8.tar.gz
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 681k  100 681k    0     0  150k      0  0:00:04  0:00:04  --:--:-- 166k
$ tar zxf hello-2.8.tar.gz
$ cd hello-2.8
$ ls
ABOUT-NLS      INSTALL        THANKS         configure.ac   man
AUTHORS         Makefile.am   TODO          contrib       po
COPYING        Makefile.in   aclocal.m4    doc           src
ChangeLog      NEWS          build-aux     lib           tests
ChangeLog.O    README        config.in     m4
GNUMakefile    README-release configure      maint.mk

```

ここで重要なのは、README、INSTALL、configure の 3 つです。README は、そのソフトウェアに関する全般的な説明が書かれています。INSTALL には、そのソフトウェアのインストール方法が書かれています。less コマンドなどで、試しに中身を読んでみて下さい。

```

$ less README
$ less INSTALL

```

configure はシェルスクリプトで、そのソフトウェアをコンパイルするときに必要なライブラリやコマンドが存在するかどうか、存在するパスはどこか、OS の環境は何かを自動的に調べて必要な設定をしてくれます。また、必要に応じてユーザが様々なオプションを指定することができます。どのようなオプションがあるかは、次のようにヘルプを表示することで読むことができます。

```

$ ./configure --help

```

様々なオプションが表示されますが、このうち最も頻繁に使われるのがインストール先の指定のオプション `-prefix` です。次のように configure を実行することで、hello コマンドを `/usr/local/bin` にインストールし、またマニュアルを `/usr/local/share/man/man1` にインストールします。もし `/usr/local` の代わりに他の場所をしていすれば、これらのファイルは指定した場所の下にインストールされます。

```

$ ./configure --prefix=/usr/local

```

configure が問題なく実行されれば、Makefile というファイルが新たに生成されているはずです。このファイルの中には、そのソフトウェアをどのような手順でコンパイルすれば良いかが全て書かれています。ユーザはどのようなソースコードがあるのかなどを気にする必要はありません。この Makefile の手順の通りに自動的にソフトウェアをコンパイルするには、以下のコマンドを打つだけです。

```

$ make -j 4

```

make コマンドが Makefile の中身を自動的に調べ、そこに書かれている手順を追ってコンパイルを行います。-j 4 のオプションは、使っている CPU 能力を最大限に引き出すためのものです。この場合、4 コアの CPU を全て使ってくれます。

これで必要なファイルが全て生成されたので、最後にインストール作業を行います。通常は `/usr/local` にインストールする場合に管理者権限が必要なので、次のコマンドを実行します。

```

$ sudo make install
Password:

```

これで、`/usr/local/bin/hello` などが配置されたはずですが、では、確認のため `hello` を実行しましょう。

```
$ hello
Hello, world!
```

どのようなオプションを渡せるかは、`-help` や `-h` をつけて確認します。

```
$ hello -h
Usage: hello [OPTION]...
Print a friendly, customizable greeting.

-h, --help          display this help and exit
-v, --version       display version information and exit

-t, --traditional  use traditional greeting format
-n, --next-generation use next-generation greeting format
-g, --greeting=TEXT use TEXT as the greeting message

Report bugs to: bug-hello@gnu.org
GNU Hello home page: <http://www.gnu.org/software/hello/>
General help using GNU software: <http://www.gnu.org/gethelp/>
```

また `hello` コマンドのマニュアルを見るには `man` コマンドを使います。

```
$ man hello
```

付録 E 本文中で登場した ROOT スクリプトの PyROOT 版

ここでは、本文中で登場した ROOT スクリプトを PyROOT で書き換えた例を掲載します。foo.C という ROOT スクリプトであれば、foo.py という PyROOT スクリプトにしてあります。

コード E.1 population.py

```
1 import ROOT
2
3 def population():
4     global hist, can, leg
5     fin = open('population.dat')
6
7     kBinsN = 21
8     hist = []
9
10    names = fin.readline().split('\t')
11
12    for i in range(len(names)):
13        hist.append(ROOT.TH1D(names[i], names[i], kBinsN, 0, 105))
14
15    for i, line in enumerate(fin.readlines()):
16        data = line.split('\t')
17        for j in range(len(hist)):
18            hist[j].SetBinContent(i + 1, float(data[j]))
19
20    can = ROOT.TCanvas('can', 'histogram')
21    can.DrawFrame(0, 0, 105, 10, 'Population by Age;Age;Population per 5-year
22        Generation (%/5 year)')
23
24    leg = ROOT.TLegend(0.65, 0.7, 0.85, 0.85)
25    leg.SetFillStyle(0)
26
27    kColor = (ROOT.kGray + 1, ROOT.kRed - 4, ROOT.kBlue - 4)
28
29    for i in range(len(names)):
30        hist[i].Scale(100./hist[i].GetEffectiveEntries())
31        hist[i].SetLineColor(kColor[i])
32        hist[i].SetFillColor(kColor[i])
33        hist[i].SetBarWidth(0.28)
34        hist[i].SetBarOffset(0.08 + 0.28*i)
35        hist[i].Draw('bar same')
```

```
35     leg.AddEntry(hist[i], hist[i].GetTitle(), 'f')
36
37     leg.Draw()
```


参考文献

- [1] R. Brun and F. Rademakers, “ROOT – an object oriented data analysis framework”, **389**, 81 (1997), *New Computing Techniques in Physics Research V*.
- [2] C. Patrignani and Particle Data Group, “Review of particle physics”, *Chinese Physics C* **40**, 100001 (2016).