KumaROOT

しょーた

2015 年 4 月 4 日

目次

第 1 章

はじめに

これまで古巣の研究室に設置した DokuWiki[*1] に ROOT などの情報を書きためていましたが、そろそろ引っ越しをしなくてはいけないかもしれません。せっかく書きためたのでどうにかして残しておきたいのですが、いまの職場のサーバでは PHP などが一切動かないため、GitHub にて公開することにしました。ついでに、動作するサンプルコードも作成していこうと思っています。

1.1 「くま ROOT」の由来

「くま ROOT」と名づけたのは、私が ROOT を使い始めたときに、最初に目を通した「猿にも使える ROOT」(通称：さる ROOT) の知名度にあやかりたかったためです。「さる」の次は「くま」を読んでもらえるように頑張りたいと思います。

ということで、読者は、ちょっとだけ ROOT を使ったことがある学生／研究者が対象になると思います。できるだけ「逆引き辞典」として使えるようにしたいと思います。

1.2 ROOT について

「ROOT」とは高エネルギー物理学業界で広く使われている解析フレームワークです。CERN が中心となって開発を行っています。もっとちゃんとした説明はそこら辺のウェブに落ちているので、ここでは僕の所感を述べることにします。

「解析フレームワーク」と聞くとたいそうに聞こえますが、「お絵かきソフト」と捉え

[*1] http://www-he.scphys.kyoto-u.ac.jp/member/shotakaha/dokuwiki/doku.php

て気軽に使えば良いと思います。[*2]。例えば、これまでエクセルで作成していたグラフを、ROOT で作ってみるとか。ROOT マクロをひとつ作っておけば、再利用できる範囲はエクセル以上だと思います。

あと、ROOT を使うにあたり注意事項は一つです。「決して深入りしないこと」。どうにもならん部分は「仕様」と思って諦めましょう。

[*2] もちろん、お絵かきといっても、ヒストグラムやグラフが中心になります

第 2 章

参考サイト

この KumaROOT に載っている情報は、下記のサイトから集め、自分で試してみたものです。本当に詳しいことは下記の情報を自分で読んでみるのが一番かもしれません。

ROOT 公式ユーザーズガイド　困ったらとりあえず読みましょう！「ROOT User's Guide」の項目から自分の読みたいドキュメントを選べばよいです。とてーも長いので全部読もうとしてはいけません。必要な時に、必要は箇所を、必要なだけ読むようにしましょう。あと、これを書いているときに気づいたのですが、初心者は「ROOT Primer」に目を通してみると良いかもしれません。

ROOT 公式リファレンスガイド　クラス名やそのメソッド名、内容、使い方を知りたい場合に使いましょう！といっても、このページから辿らなくても、「cern root ttree」[*1]などと Google 検索すればたいていヒットします。「クラスの説明＋メソッド一覧」という構成になっています。最初は読むのに苦労するかもしれませんが、使いこなせるように頑張りましょう。ROOT 中級者はみんな使っています。

ROOT 公式チュートリアル　付属のサンプルコードの説明です。ドキュメントやレファレンスを読むより、実際にコードを動かし、ソースを読むほうが早く身につきます。初めて使うクラスなどはとりあえずサンプルを動かしてみましょう。

ROOT 公式コーディングガイド　ROOT のソースコードを読むときに役に立つかも。なんとなく知っておくと良いです。覚える必要は全くありません。

ROOT 解体新書 by 楠本さん[*2]　gROOT や、gStyle の設定など、他のページ／マニュアルでは見られない項目がすごく詳しい。

[*1] 検索の際「cern」と付けるのが重要です。でないと、管理者の意味の「root」がたくさんヒットします。

[*4] ページのリンク切れを確認（*[2015-01-27 Tue]*）

宇宙線実験の覚え書き（大学院生版） by **奥村さん**　PyROOT **のことをググっていてこのページに辿り着きました**[*3]**。いまは** **宇宙線実験の覚え書き（社会人版）** **にお引っ越ししたみたいです。**

[*3] **青い色のページにお世話になってた気がするんだけど、気のせいかな**

第 3 章

ROOT を使いたい

3.1 オススメのインストール方法（for Mac ユーザ）

```
$ sudo port install root5     ## for ROOT5 user
$ sudo port install root6     ## for ROOT6 user
```

さて、まずはインストールしないとはじまりません。Mac の場合は MacPorts からインストールできるようになっています[*1]。すごく楽ちんなのでオススメです。環境変数（$ROOTSYS、$LD_LIBRARY_PATH、$DYLD_LIBRARY_PATH など）の設定も不要です。

ROOT6 は 2014 年にリリースされました。MacPorts の場合「root6」というパッケージ名[*2]でインストールすることができます。ROOT5 と一緒にインストールしておいても大丈夫ですが、一緒には使えません。後述する方法で簡単に切り替えることが出来ます。

現在は ROOT5 から ROOT6 へ移行する過渡期です。そのため、進行中の実験は ROOT5 系を使っていることが多いです[*3]。むやみに最新版を使うと、実験固有のソフトウェア群が動作しないこともあります。その場合は、素直に実験で推奨されているバージョンには従ってください。

[*1] Homebrew や fink など、その他のパッケージ管理ツールは使ったことがないので分かりません。誰か情報をくださいな

[*2] 正確には「ポート名」かも。あまり気にしないでください。

[*3] LHC 実験の要求に応えるため（？）、３年位前から ROOT の開発が非常に活発に行われています。しかし、その他の実験では、安定性を求め古いバージョン使い続けていることがあります。あと OS が古いと最新バージョンはうまくインストールできないこともあります

3.1.1 Git を使ったインストール方法

Linux を使っている場合は、Git を使うとよいでしょう。方法は「installing-root-source | ROOT 公式ページ」に載っている通りです。

```
## STEP1 : make a directory for cloning
[any]  $ mkdir ~/repos/git
[any]  $ cd ~/repos/git

## STEP2 : clone ROOT repository from CERN
[git]  $ git clone http://root.cern.ch/git/root.git
[git]  $ cd root

## STEP3 : look for a version
[root] $ git tag -l

## STEP4 : checkout the tag version you want
[root] $ git checkout -b v5-34-08 v5-34-08

## STEP5 : set PREFIX and configure, then make, then make install
##       : use tee command for logging output file when building and
    installing
[root(v5-34-08)] $ ./configure --prefix=/usr/local/heplib/ROOT/v5-34-08
[root(v5-34-08)] $ make 2>&1 | tee make.log                    ## buiding ROOT
[root(v5-34-08)] $ make install 2>&1 | tee makeinstall.log     ## installed
    under $PREFIX
```

1. 本体をどこかに clone する（今回は、~/repos/git/ 以下）
2. どんなタグがあるのかを調べる
3. 目的のバージョンのタグ名が分かったら、そのタグをチェックアウトします。ブランチ名は好きにしてかまいません（今回は、タグ名と同じ名前）。
4. あとは、従来通り PREFIX を指定して configure します。configure の内容は たしか config.status に書きだされます。make、make install の際、PREFIX で指定したディレクトリによっては sudo が必要です。また、失敗した場合に備えてログを残しておくとよいです。今回は、tee コマンドを使うことで、端末に表示しながらログファイルに保存しています。

あとは、~/.bashrc に環境変数の設定を書いておきます。

3.1.2 従来のインストール方法

ググればたくさん出てきますが、一応紹介しておきます。

```
## STEP1 : make directory for tar.gz file and wget it
$ cd /usr/local/heplib/tarballs
$ wget ftp://root.cern.ch/root/root_v5.30.06.source.tar.gz     ## check URL at
    ROOT website
```

```
## STEP2 : expand tar.gz
[ROOT] $ cd /usr/loca/heplib/ROOT
[ROOT] $ tar zxvf ../tarballs/root_v5.30.06.source.tar.gz

## STEP3 : set PREFIX then configure -> make -> make install
[ROOT] $ cd root
[root] $ ./configure --prefix=/usr/local/heplib/ROOT/v5-30-06
[root] $ make 2>&1 | tee make.log
[root] $ make install 2>&1 | tee makeinstall.log
```

3.1.3 インストール方法 for Windows ユーザ

Windows はよく分かりません。ごめんなさい。たぶん「downloading-root | ROOT 公式ページ」から目的のバージョンを選び、そこからバイナリを落としてくるのが一番簡単だと思います。

3.2 ROOT5 と ROOT6 を試してみたい

```
$ sudo port select root root6    ## use ROOT6
$ sudo port select root root5    ## use ROOT5
```

MacPorts で ROOT をインストールする利点のひとつは、ROOT5 と ROOT6 が簡単に切り替えられることです。

実はこの「port select」は ROOT だけでなく、Python のバージョン切り替えなどもできます。どのパッケージが使えるかは以下のコマンドで確認できます

```
$ port select --summary
```

3.2.1 ROOT5 と ROOT6 の違いについて

ROOT マクロなどを実行する際に使うインタプリタが変更されたみたいです[*4]。細かい違いは全く分かりませんが、文法のチェックが厳密になったみたいです。

実は ROOT5 では C 言語／ C++ 言語の文法的には間違っているマクロでも動いてくれました[*5]。そのため、テストで作ったマクロで動作確認した後、より多くのデータを解析するためにコンパイルするとエラーが多出。そのデバッグに追われるということは日常茶飯事でした。

[*4] CINT → CINT++ に変更

[*5] よく知られていると思われるのは、a.b でも a->b でも動いちゃうことでしょうか

ROOT6 では、このマクロの文法チェックも厳しくなったみたいです。ひぇぇ。でも心配しなくて大丈夫。エラーの内容を詳しく教えてくれるようになりました。よくある行末のセミコロンのつけ忘れなども指摘してくれます。これで場所の分からない segmentation fault に悩まされることも減るでしょう。

試しに、ROOT5 のチュートリアルを ROOT6 で実行してみてください。「warning」や「error」がたくさん表示されます。

```
## STEP1 : set to ROOT6
$ sudo port select root root6

## STEP2 : move to tutorials/ directory of ROOT5
$ cd /opt/local/libexec/root5/share/doc/root/tutorials/

## STEP3 : start (ROOT=) ROOT6
$ root

## ===  an example of warning ===
/opt/local/libexec/root5/share/doc/root/tutorials/rootalias.C:7:13:
 warning: using the result of an assignment as a condition without
 parentheses [-Wparentheses]
      if (e = getenv("EDITOR"))
          ~~^~~~~~~~~~~~~~~~~~

## === an example of error ===

/opt/local/libexec/root5/share/doc/root/tutorials/rootalias.C:39:12:
 error: cannot initialize return object of type 'char *' with an rvalue of
 type 'const char *'
    return gSystem->WorkingDirectory();
           ^~~~~~~~~~~~~~~~~~~~~~~~~~~


$ .q
```

3.3 PyROOT を使いたい

```
$ sudo port install root5 +python27   ## when ROOT5, you need to specify +
    pythonXX variants
$ sudo port install root6             ## when ROOT6, no need to specify
    variants
```

CERN には「へびつかい」が多いらしく、「PyROOT[*6]」というモジュールを使えば、Python 上で ROOT が使えるようになっています。

その場合は、MacPorts でインストールする際に variants で指定する必要があります。しかも、この variants は自分の使っている Python のバージョンに合わせる必要があります。ミスマッチな場合は、動作しません（＝クラッシュします）。

ROOT6 の場合は python27 がデフォルトで ON になっています。

[*6] 実は rootpy というのもあります。こっちのほうが Python native な感じです。前に試そうとしてたのですがインストールでコケてしまいました。動かせたら項目を作るかも

3.4 Emacs で ROOT を編集したい

```
$ locate root-help.el     # check path
```

これもあまり知られていないと思うのですが、Emacs 上で ROOT のソースを編集するのを簡単にする Elisp パッケージが一緒にインストールされます。locate コマンドでどこにあるか調べておきましょう。

ちなみに、僕の場合（= MacPorts の場合)、以下にありました。

```
/opt/local/libexec/root5/share/emacs/site-lisp/root-help.el
/opt/local/libexec/root6/share/emacs/site-lisp/root-help.el
```

これの使い方に関しては、あとできちんと調べて書くことにします。

3.5 ROOT の tutorial を使いたい

```
/opt/local/libexec/root5/share/doc/root/tutorials/      ## ROOT5
/opt/local/libexec/root6/share/doc/root/tutorials/      ## ROOT6
```

実は ROOT をインストールすると、たくさんのサンプルコードもついてきます。使い方をウェブで検索してもよく分からない場合は、このサンプルコードを動かしながら中身をいじくってみるのが一番です。

とりあえず、いつでも使えるようにテスト用ディレクトリを作成しコピーしておきましょう。以下に一例を示しましたが、自分の環境に合わせて適宜変更してください。

```
$ cp -r /opt/local/libexec/root5/share/doc/root/tutorials ~/TEST/root5/
$ cp -r /opt/local/libexec/root6/share/doc/root/tutorials ~/TEST/root6/
```

cp コマンドを使う際には、-r オプションを付けることでサブディレクトリもコピーできます。その際、コピー元（＝第１引数）の最後に「/」付けてはいけません。コピー先（＝第２引数）の最後には「/」を付けてもよいです（もしかしたらなくてもよいのかも)[*7]。

[*7] この辺はよく忘れます。失敗したらコピー先を削除すればいいだけなので、失敗もしてみてください

第 4 章

ROOT tutorial 編

この章では、主に ROOT に付属している tutorial を使用して、使い方を簡単に紹介します。前節の最後にも書きましたが、手元にコピーを作っておきましょう。

```
$ cp -r /opt/local/libexec/root6/share/doc/root/tutorials ~/TEST/root6/
```

とりあえず ROOT6 の tutorial を使います。気が向いたら ROOT5 との比較もしようかと思います。

4.1 とりあえず起動

```
$ cd ~/TEST/root6/tutorials/
$ root

root[0]
```

コマンドラインで「root」と入力すると、ROOT が起動します。この状態だと、対話的に ROOT を操作することができます。

4.2 とりあえず終了

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

ROOT セッション内で「.q」を入力すると、ROOT が終了します。それで終了しない場合は、「.qqq・・・」の様に q をたくさんにします。

4.2.1 rootlogon.C と rootlogoff.C

さて、tutorials **をコピーしたディレクトリで** ROOT **を起動／終了すると、以下の様なメッセージが表示されたはずです。**

```
Welcome to the ROOT tutorials

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

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

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

root [0]
```

```
Taking a break from ROOT? Hope to see you back!
```

これは、同じディレクトリに、「rooglogon.C**」と「**rootlogoff.C**」があるからです。この２つのファイルを用意しておくことで、**ROOT **起動時および終了時の動作を設定することができます。**

個人的には、数ヶ月ぶりに触るプログラムなんてほとんど忘れてしまっているので、rootlogon.C **に手順を書いて残したりしています。**

4.2.2 ROOT 起動時に読み込まれるファイルの順番

ROOT **起動時に以下の順番でファイルが読み込まれます。**

1. system.rootrc
2. ~/.rootrc
3. ./rootlogon.C

個人的な全体設定は「~/.rootrc**」へ、そのプログラムだけの設定は「**./rootlogon.C**」に書いておけばよいです。**

4.3 demos.C を実行してみる

さて、ROOT **を起動して表示されたメッセージにしたがって、**demos.C **を実行してみましょう。**ROOT **内で実行する場合は、「**.x **ファイル名」と入力します。ファイル名の部分は** TAB **補完ができます。これを** bash **で実行する場合は以下のようにします。**

```
$ root demos.C
```

さてさて、実行すると図 4.1 のようなツールバーが出てきます。

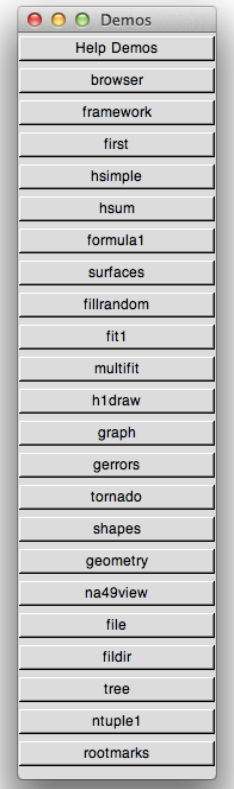

図 4.1　demos.C を実行した時に出てくるツールバー的なもの

一番上にある、「Help Demos」をクリックすると、図 4.2 のようなキャンバスが表示されます。

とりあえずこの通りにボタンを押してみましょう。

4.4　hsimple.C を実行してみる

```
$ root hsimple.C
```

前節のようにボタンを押して実行するか、上の行の様にコマンドラインから「hsimple.C」を走らせると、キャンバスが表示され、ヒストグラムが成長していきます。それと同時

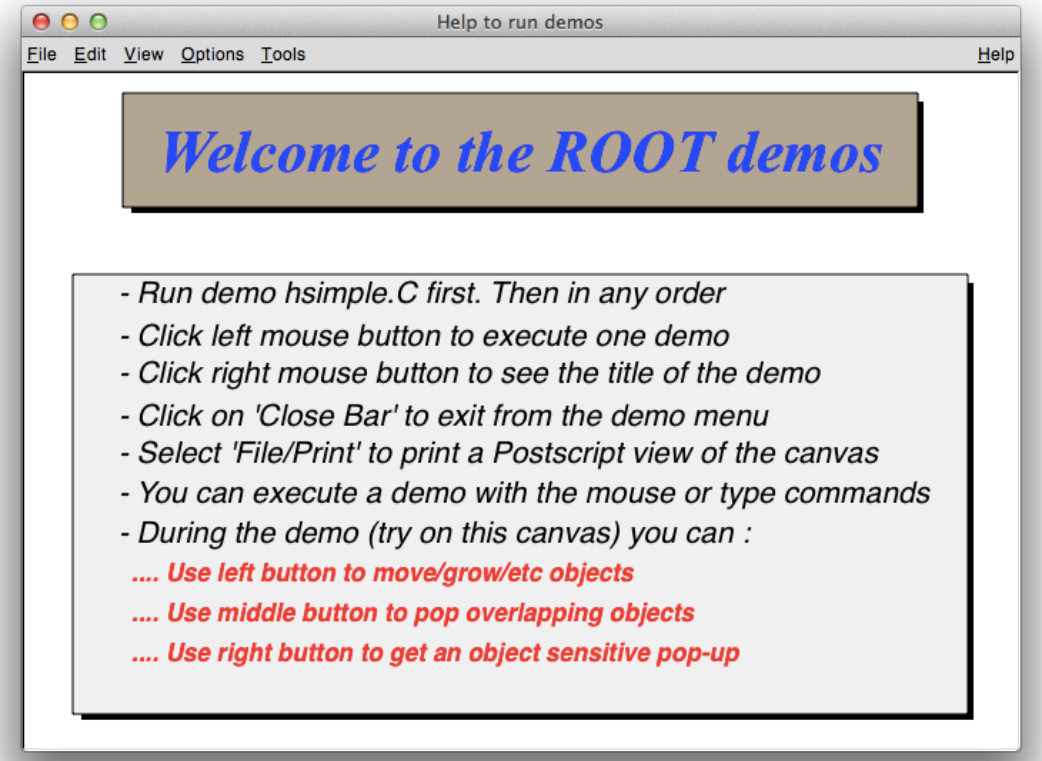

図 4.2　Help Demos を実行すると出てくるキャンバス

に、「hsimple.root」という ROOT ファイルが作成されます。

「hsimple.C」を開いて、上から順番に何をしているのかを確認してみましょう。

4.4.1　インクルードファイル

とりあえず無視して OK です。コンパイルする場合は必要ですが、マクロで動かす場合は書かなくてもよいです。

```
#include <TFile.h>
#include <TNtuple.h>
#include <TH2.h>
#include <TProfile.h>
#include <TCanvas.h>
#include <TFrame.h>
#include <TROOT.h>
#include <TSystem.h>
#include <TRandom3.h>
#include <TBenchmark.h>
#include <TInterpreter.h>
```

4.4.2 関数の定義

マクロの場合ファイル名と関数名は一緒にします。戻り型はなんでも OK です。引数を指定することもできます。

```
TFile *hsimple(Int_t get=0)
```

4.4.3 コメントの挿入

コメントは C++ の作法で挿入できます

```
{
//  This program creates :
//    - a one dimensional histogram
//    - a two dimensional histogram
//    - a profile histogram
//    - a memory-resident ntuple
//
//  These objects are filled with some random numbers and saved on a file.
//  If get=1 the macro returns a pointer to the TFile of "hsimple.root"
//          if this file exists, otherwise it is created.
//  The file "hsimple.root" is created in $ROOTSYS/tutorials if the caller
    has
//  write access to this directory, otherwise the file is created in $PWD
```

4.4.4 ファイル名の宣言

TString クラスという文字列クラスを使っています。普通の C/C++ の関数を使うよりはるかに楽なので、積極的に使うと良いと思います。

```
TString filename = "hsimple.root";
TString dir = gSystem->UnixPathName(__FILE__);
dir.ReplaceAll("hsimple.C","");
dir.ReplaceAll("/./","/");
TFile *hfile = 0;
if (get) {
   // if the argument get =1 return the file "hsimple.root"
   // if the file does not exist, it is created
   TString fullPath = dir+"hsimple.root";
   if (!gSystem->AccessPathName(fullPath,kFileExists)) {
      hfile = TFile::Open(fullPath); //in $ROOTSYS/tutorials
      if (hfile) return hfile;
   }
   //otherwise try $PWD/hsimple.root
   if (!gSystem->AccessPathName("hsimple.root",kFileExists)) {
      hfile = TFile::Open("hsimple.root"); //in current dir
      if (hfile) return hfile;
   }
}
```

```
//no hsimple.root file found. Must generate it !
//generate hsimple.root in current directory if we have write access
if (gSystem->AccessPathName(".",kWritePermission)) {
   printf("you must run the script in a directory with write access\n");
   return 0;
}
```

4.4.5 ROOT ファイルを開く

TFile クラスを使います。直前の if 文の中ではファイルの存在を確認しています。ファイルがある場合は、TFile::Open メソッドでファイルを開いています。ない場合は、TFile::TFile コンストラクタで新しい TFile オブジェクトを作成しています。

```
TFile *hfile = 0;

hfile = TFile::Open(fullPath); //in $ROOTSYS/tutorials
hfile = TFile::Open("hsimple.root"); //in current dir

hfile = (TFile*)gROOT->FindObject(filename); if (hfile) hfile->Close();
hfile = new TFile(filename,"RECREATE","Demo ROOT file with histograms");
```

4.4.6 ヒストグラムを作成する

TH1 クラス、TH2 クラスなどを使います。ここでは、TProfile クラスや TNtuple クラスも使われています。

```
// Create some histograms, a profile histogram and an ntuple
TH1F *hpx = new TH1F("hpx","This is the px distribution",100,-4,4);
hpx->SetFillColor(48);
TH2F *hpxpy = new TH2F("hpxpy","py vs px",40,-4,4,40,-4,4);
TProfile *hprof = new TProfile("hprof","Profile of pz versus px"
    ,100,-4,4,0,20);
TNtuple *ntuple = new TNtuple("ntuple","Demo ntuple","px:py:pz:random:i");
```

4.4.7 プロセス時間の測定開始

このマクロを実行すると、ターミナル上にプロセス時間が表示されます。この部分から測定を開始しています。

```
gBenchmark->Start("hsimple");
```

4.4.8 キャンバスの作成

グラフを描く領域をキャンバスと呼びます。TCanvas クラスを使います。

```
// Create a new canvas.
TCanvas *c1 = new TCanvas("c1","Dynamic Filling Example",200,10,700,500);
c1->SetFillColor(42);
c1->GetFrame()->SetFillColor(21);
c1->GetFrame()->SetBorderSize(6);
c1->GetFrame()->SetBorderMode(-1);
```

4.4.9 ヒストグラムに値を詰める

このマクロでは、ヒストグラムにランダムな値を詰め込んでいます。

```
// Fill histograms randomly
TRandom3 random;
Float_t px, py, pz;
const Int_t kUPDATE = 1000;
for (Int_t i = 0; i < 25000; i++) {
   random.Rannor(px,py);
   pz = px*px + py*py;
   Float_t rnd = random.Rndm(1);
   hpx->Fill(px);
   hpxpy->Fill(px,py);
   hprof->Fill(px,pz);
   ntuple->Fill(px,py,pz,rnd,i);
```

4.4.10 キャンバスに描画する

TH1::Draw() **メソッドで描画します。**

```
   if (i && (i%kUPDATE) == 0) {
      if (i == kUPDATE) hpx->Draw();
      c1->Modified();
      c1->Update();
      if (gSystem->ProcessEvents())
         break;
   }
}
```

4.4.11 プロセス時間の表示

```
gBenchmark->Show("hsimple");
```

4.4.12 ROOT ファイルに保存する

```
   // Save all objects in this file
   hpx->SetFillColor(0);
   hfile->Write();
   hpx->SetFillColor(48);
   c1->Modified();
```

```
    return hfile;

// Note that the file is automatically close when application terminates
// or when the file destructor is called.
}
```

第 5 章

全体設定編

5.1 初期設定したい（rootrc, rootlogon.C）

5.1.1 rootrc

bash の設定を ~/.bashrc に書くように、ROOT の設定は ~/.rootrc に書きます。デフォルト値は、{ROOT をインストールしたパス}/etc/system.rootrc に書かれているので、とりあえずこれをホームディレクトリにコピーして編集したら OK です。

```
$ locate system.root
# $ROOTSYS/etc/system.rootrc
$ cp $ROOTSYS/etc/system.rootrc ~/.rootrc
```

5.1.2 rootlogon.C

5.2 キャンバスを無地にしたい（gROOT → SetStyle）

```
gROOT->SetStyle("Plain");
```

ROOT v5.30 からキャンバスの色がデフォルトで無地になりました。なので、それ以降のバージョンを使っている場合、特に設定は必要ありません。

（いないと思いますが）昔のキャンバス（＝灰色っぽいやつ）を使いたい場合は、"Classic" を指定します。

5.3 統計情報を表示したい（gStyle->SetOptStat）

ヒストグラムを描画すると、右上にそのヒストグラムの情報が表示されます。デフォルトだと３つしか表示されないので、少し増やしておきます。

```
gStyle->SetOptStat(112211)
```

引数はビットのようなものを表しています。このビットは右から読まれます。最大で９くらいまでいける気がする。0 or 書かなければ「非表示」、1 は「表示」、2 は「エラー表示」です。

5.4 フィットの結果を表示したい（gStyle->SetOptFit）

```
gStyle->SetOptFit(1111111)
```

ビットの使い方は、ひとつ前の「統計情報を表示したい」と同じです。

5.5 ヒストグラムの線の太さを一括で変更したい（gStyle->SetHistLineWidth）

ヒストグラムの外枠線の太さは、一括で設定しておくことができます。デフォルトだと少し細い気がするので、太くしておくとよいと思います。ただし、たくさんのヒストグラムを描く際は、見えにくくなってしまうので細くします。その辺りは臨機応変にお願いします。

```
gStyle->SetHistLineWidth(2)
```

5.6 デフォルトの色を変更したい（gROOT->GetColor->SetRGB）

```
gROOT->GetColor(3)->SetRGB(0., 0.7, 0.); // Green  (0, 1, 0)->(0, 0.7, 0)
gROOT->GetColor(5)->SetRGB(1., 0.5, 0.); // Yellow (1, 1, 0)->(1, 0.5, 0)
gROOT->GetColor(7)->SetRGB(0.6, 0.3, 0.6); // Cyan (0, 1, 1)->(0.6, 0.3, 0.6)
```

デフォルトは（1:黒, 2:赤, 3:黄, 4:青, 5:黄緑, 6:マゼンダ, 7:シアン）なのですが、この中で、（3:黄, 5:黄緑, 7:シアン）は明るすぎてとても見えづらいので、もう少し見やすい色に変更します。

上２つは奥村さんのページのコピペ、最後のはシアンを紫っぽい色に変更しました。

RGB の度合いは自分の好みで選んでください。手順としては、RGB の値を検索（Wikipedia 使用すると良い）→ その値を 256（ほんとは 255 かも？）で割るだけです。

おまけとして、ROOT 公式ブログの「虹色カラーマップを使うこと」 の記事もリンクしておきます。

5.7 横軸に時間を使いたい（SetTimeFormat, SetTimeDisplay）

```
gStyle->SetTimeOffset(-788918400);     // set diff. btw Unix and ROOT epoch
graph->GetXaxis()->SetTimeDisplay(1);
graph->GetXaxis()->SetTimeFormat("%Y\/%m\/%d");
graph->GetXaxis()->SetTimeOffset(0, "gmt");     // set GMT+0
```

Unix の epoch time は 1970 年 01 月 01 日 00 時 00 分 00 秒から始まるのに対し、ROOT の epoch time は 1995 年 01 月 01 日 00 時 00 分 00 秒から始まるので、その差をオフセットとして設定する必要がある。

5.7.1 Unix epoch と ROOTepoch の差を計算する

簡単な計算なので確かめてみる

```
25[years] * 365[days/year * 24[hours/day] * 60[minutes/hour] * 60[seconds/
    minute]
+ 6[days] * 24[hours/day] * 60[minutes/hour] * 60[seconds/minutes]     // 6
    leap year in 25 years
= 788918400[seconds]
```

5.7.2 GMT+0 に設定する

```
graph->GetXaxis()->SetTimeOffset(0, "gmt");
```

理由は忘れてしまったが、上の設定をしないと軸の時間がずれてしまってたはず。

epoch の時間ではなく、作成したグラフ／ヒストグラムの軸に対して設定する

5.7.3 月日と時刻を 2 段にして表示したい

```
graph->GetXaxis()->SetTimeFormat("#splitline{/%m\/%d}{%H:%M}");
```

時間に対する安定性を示したい場合などに使える。

5.8 キャンバスに補助線を描きたい（gStyle->SetPadGridX）

```
gStyle->SetPadGridX(1)     // X-axis grid
gStyle->SetPadGridY(1)     // Y-axis grid
```

5.9 グラフの軸を一括してログ表示にする（gStyle->SetOptLogx）

```
gStyle->SetOptLogx(1)     // X-axis
gStyle->SetOptLogy(1)     // Y-axis
```

5.10 軸の目盛り間隔を変更したい（gStyle->SetNdivisions）

```
gStyle->SetNdivisions(TTSSPP)
```

PP　軸全体の分割数

SS　PP 分割された目盛り 1 つ分の分割数

TT　SS 分割された目盛り 1 つ分の分割数

デフォルトは 510 になっている。PP=10、SS=05、TT=00 なので、軸を 10 分割してその 1 目盛りを 5 分割、ということで全体で 50 目盛りになる。

全体を 100 目盛りにするには、20510 にすればよい。（10 分割、その 1 目盛りを 5 分割、さらにその 1 目盛りを 2 分割 = 100 目盛り）

第 6 章

ヒストグラム編

6.1 1 次元ヒストグラムを作成したい : TH1D

ヒストグラムは以下の手順で作成・描画します。

1. **ヒストグラムのオブジェクトの作成**
2. **ヒストグラムに値を詰める**
3. **ヒストグラムをキャンバスに描画する**

それらに必要な ROOT のクラス・メソッドは以下通りです。

```
TH1D TH1D(const char* name, const char* title, Int_t nbinsx, Double_t xlow,
    Double_t xup)
Int_t Fill(Double_t x)
void Draw(Option_t* option = "")
```

TH1D　**1 次元ヒストグラム（Double 型）のコンストラクタ。**

- name　**オブジェクト名。他のオブジェクト（ヒストグラムだったり、キャンバスだったり）と重複しないようにする。**
- title　**ヒストグラム全体のタイトル。X 軸、Y 軸のタイトルも同時に設定することができる。**
- nbinsx　**X 軸のビンの数。X 軸を何分割するか決める。**
- xlow　**X 軸の最小値**
- xup　**X 軸の最大値**

Fill　**ヒストグラムに値を詰めるメソッド**

Draw　**ヒストグラムを描くメソッド。描画のオプションを設定できる。**

```
TString hname, htitle;
hname.Form("hname");     // <------------------------ object name of histogram
htitle.Form("title;xtitle;ytitle;");     // <-------- title and axis name
Double_t xmin = 0, xmax = 10;     // <--------------- left edge and right edge
Int_t xbin = (Int_t)xmax - (Int_t)xmin;     // <----- number of bins

TH1D *h1 = new TH1D(hname.Data(), htitle.Data(), xbin, xmin, xmax);
```

- **ヒストグラムに限らず** ROOT **オブジェクトには「名前」をセットする必要がある**
- **タイトル部分を「;」で区切ることで、軸名を設定することができる（"タイトル**;X **軸名**;Y **軸名前"）**
- TString::Form **は** printf **の書式が使えるのでとても便利**

6.2 ２次元ヒストグラムを作成したい : TH2D

```
TH2D TH2D(const char* name, const char* title, Int_t nbinsx, Double_t xlow,
    Double_t xup, Int_t nbinsy, Double_t ylow, Double_t yup)
Int_t Fill(Double_t x, Double_t y)
void Draw(Option_t* option = "")
```

6.3 タイトルを変更したい : SetTitle

```
void SetTitle(const char* title = "")   // *MENU*
```

6.4 統計ボックスを表示したい : SetStats

```
void SetStats(Bool_t stats = kTRUE)      // *MENU*
```

6.5 X 軸名を設定したい : SetXTitle

```
void SetXTitle(const char* title)
```

6.6 タイトルを中心にしたい

6.7 平均値、RMS を知りたい : GetMean, GetRMS

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

6.8 値を詰めたい : Fill

6.9 面積でノーマライズしたい

6.10 ２軸グラフを作成したい

「tutorials/hist/twoscales.C」を参考にする

第 7 章

TTree 編

ROOT を使うにあたって、TTree（もしくは次の章の TChain）は基礎中の基礎です[*1]。とりあえず、取得したデータはさっさと TTree に変換してしまいましょう。

7.1 テキストファイルを TTree に変換したい

取得したデータはとりあえずテキストデータとして保存するのが一番簡単な方法です。そして、ROOT で解析するときは TTree になっていると楽ちんです。なので、手間を掛けずにさっさと変換してしまいましょう。

7.1.1 TTree::ReadFile を使う方法

データがテキストファイルで保存されている場合、それを TTree に変換する最も簡単な方法です。

```
tree->ReadFile(ifn.Data(), "row1/I:row2/I:row3/I:row4/D:row5/I");
```

第 1 引数　入力ファイル名

第 2 引数　branch descriptor。TTree のブランチ変数になります。複数のブランチ変数を指定する場合は、コロン（:）で区切って記述します。Int_t 型の場合は「ブランチ名/I」、Double_t 型の場合は「ブランチ名/D」といった感じで、その変数名（=

[*1] 「さる ROOT」や、他のウェブサイトでは「TNtuple」をサンプルとして取り上げていますが、これだけを使っている研究者はみたことがありません。TTree のベースには TNtuple があるのかもしれませんが、なんでこんな使われていないものをサンプルにするのか疑問です

ブランチ名）とその型を指定できます。型を省略した場合は Float_t 型の「ブランチ名/F」になるみたいです。

サンプルコード

仮に、100 行４列のテキストファイルがあるとします。このファイルの「行数」はイベント数に相当し、「列数」は取得したデータの項目に相当します。

```
100     105     104     103
101     106     103     100
...
```

```
{
  // STEP1: Set input filename
  TString ifn = "inputfilename";

  // STEP2: Create TTree
  TTree *tree = new TTree("tree", "tree using ReadFile()");

  // STEP3: Read data using TTree::ReadFile(...) method
  tree->ReadFile(ifn.Data(), "row1/I:row2/I:row3/I:row4/D:row5/I");

  // STEP4: Create TFile to save TTree
  TString ofn = "out.root";
  TFile *fout = new TFile(ofn, "recreate");

  // STEP5: Write TTree to TFile
  tree->Write();

  // STEP6: Close TFile
  fout->Close();

  return;
}
```

7.1.2 TTree::Branch() を使う方法

```
tree->Branch("run", &run, "run/I")
```

第１引数　ブランチ名；なんでも良い。用意した変数名と違っていても構わない

第２引数　変数のアドレス；変数が実体の場合は、&を先頭につけてアドレスを指定する。配列の場合はそのまま（array） or 最初の配列のアドレス（&array[*2]）を指定する。事前に変数を用意しておかないと怒られる

[*2] DEFINITION NOT FOUND.

第３引数　変数の型；“変数／型”の形で記述する。int **型は** I, float **型は** F, double **型は** F **など**

サンプルコード

よくある方法です。ググればいっぱい見つかります。

```
{
    // STEP1: データファイルを読み込む
    TString ifn = "inputfilename"
    ifstream fin;
    fin.open(ifn);

    // STEP2: データを格納するための変数を定義する
    int val1, val2, val3, val4;

    // STEP3: を作成するTTree
    TTree *tree = new TTree("name", "title);

    // STEP4: TTree::Branch(...)を使って、各変数のブランチを作成する
    tree->Branch("val1", &val1, "val1/I");
    tree->Branch("val2", &val2, "val2/I");
    tree->Branch("val3", &val3, "val3/I");
    tree->Branch("val4", &val4, "val4/I");

    // STEP5: Cでファイルを読み込むときの常套手段++
    while (fin >> val1 >> val2 >> val3 >> val4) {
        // STEP6: データのエントリの区切りで必ずTTree::Fill()する
        tree->Fill();
    }

    // STEP7: 作成したを保存するためのを作成するTTreeTFile
    TString ofn = "outputfilename";
    TFile *fout = new TFile(ofn, "recreate");

    // STEP8: にを書き込むTFileTTree
    tree->Write();  //

    // STEP9: を閉じるTFile
    // プログラム（やマクロ）終了時に勝手に閉じてくれるらしいが一応
    fout->Close();

    return;
}
```

前述した ReadFile **を使った方法と比べると、コードの行数がぐーんと多いことが分かります。(**ReadFile **の場合、肝となる部分はたったの一行です)。**

行数が増えた分、汎用性が高くなっています。こちらの方法だと、ブランチに「配列」を設定することも可能です。

7.2　ブランチに配列を使いたい

TTree::Branch **を理解していれば簡単です。**

```
int val1[100];
TTree *tree = new TTree("tree", "tree using array");
tree->Branch("val1", val1, "val1[100]/I");
```

第２引数には「変数のアドレス」を指定します。val1 **は 配列の先頭アドレスを指しているので、＆をつける必要はありません。第３引数には、配列の長さをベタ書きします。**

7.2.1 ブランチに文字列を使いたい

配列を使うことができるので、文字列のブランチを作ることもできます。

```
char hoge[32];
tree->Branch("moji", hoge", "moji[32]/C")
sprintf(hoge, "hogehogefugafuga")
tree->Fill();
```

7.3 ブランチに可変長配列を使いたい

少し手間を加えると可変長配列も扱えます。

1. **配列の大きさ** fN **を定義する**
2. **配列** val **を定義する**
3. fN **のブランチを作る**
4. val **のブランチを作る**

```
Int_t fN;                                          // (1) 設定したい配列の大きさ
Int_t val[max];                                    // (2) val[max]: はよりも大きな数
    maxfN
tree->Branch("nch", &fN, "nch/I");                 // (3) まずをブランチにセットする；だと
    何の変数か分かりづらいので、（全チャンネル数の意）に変更した点に注意fNfNnch
tree->Branch("val", val, "val[nch]/I");            // (4) 次にval[fNをセットする；]でも、
    でもなくなく、にする点に注意maxfNnch

// (4)を以下のようにすると、"Illegal leaf ..." と怒られる
tree->Branch("val", val, "val[fN]/I");             // には、ブランチ名を入れる必要があるらし
    い（fN
```

7.3.1 ブランチに可変長文字列を使いたい

```
#include <string.h>     // strlen()を使うために必要

const Int_t NMAX_MOJI = 100;
char hoge[NMAX_MOJI];
Int_t nmoji;
tree->Branch("nmoji", &nmoji, "nmoji/I");
tree->Branch("moji", hoge, "hoge[nmoji]/C");
```

```
sprintf(hoge, "hoge-hoge-fuga-ga");
nmoji = strlen(hoge)
tree->Fill()
```

7.3.2 ブランチに std::vector を使いたい

```
#include <vector>

std::vector<Double_t> vec;
TTree *tree = new TTree("tree", "tree using vector");
tree->Branch("vec", &vec);
```

<vector>を include する :: namespace を定義しない場合は、”std::vector<型> 変数名”と宣言すること。当たり前のことだけど、結構忘れてしまう。ROOT(CINT) を起動させると、“vector<型> 変数名”で使えてしまうため、よく忘れる…orz vector 型の変数は実体であるため、第２引数は先頭に“&“が必要 array と同じようにすると怒られる ROOT が空気を読んでくれるため、第３引数はなくてよいみたいまぁでも一番最後のブランチにするのが無難かも

第 8 章

TChain 編

TChain を使うと同じ構造の TTree を複数連結（= chain）して、ひとつの TTree として扱うことができます。TTree を継承したクラスなので、連結した後は TTree と同じように使えば OK です。

8.1 複数の TTree を読み込みたい

```
TChain *chain=new TChain("tree", "tree title");
```

第 1 引数　読み込む TTree の名前; 読み込む TTree の名前と一致してないと怒られる
第 2 引数　タイトル; 説明みたいなもの。なくても大丈夫

```
chain->Add("../anadata/CALIB_RUN10.root");
chain->Add("../anadata/CALIB_RUN11.root");
chain->Add("../anadata/CALIB_RUN12.root");
```

第 1 引数　ファイル名; ワイルドカード指定もできる

8.1.1 サンプルコード : ループで読み込む

```
TChain *chain=new TChain("chain", "chainname");
const Int_t fNFile=11;
Int_t iFile;
for (iFile=0; iFile<fNFile; ++iFile) {
    chain->Add(Form("../anadata/CALIB_RUN%d.root", iFile+10));
}
```

8.1.2 サンプルコード : ワイルドカード指定

```
TChain *ch = new TChain("upk");
ch->Add("upk_run*.root")
```

8.2 読み込んだ TTree の数を知りたい

```
chain->GetNtrees()
```

8.3 読み込んだ TTree のリストを取得したい

```
TObjArray *fileElements = fBsd->GetListOfFiles();
TIter next(fileElements);
TChainElement *chEl = 0;
while (( chEl=(TChainElement*)next() )) {
    fprintf(stdout, "[%s]\tListOfFiles\t'%s'\n", __FUNCTION__, chEl->GetTitle
        () );
}
```

ROOT マニュアルに載ってた

第 9 章

TFile 編

第 10 章

TCanvas 編

10.1 色見本を見たい

ROOT のプロンプト内で下のように入力すると、簡単に確認できる。

```
root> TCanvas c
root> c.DrawColorTable()
```

10.2 グラフの軸をログ表示にしたい

```
TCanvas *c1;
c1->SetLogy();
```

10.2.1 キャンバスを分割している場合

まず、分割したいキャンバスに移動する

```
c1->cd(2)->SetLogy();
h1->Draw();
```

gPad は current canvas へのポインタなので、下のようにも書くことができる。

```
c1->cd(2);
gPad->SetLogy();
```

10.3 複数のキャンバスを PDF に保存したい

PDF 形式で保存する場合のみ、複数のキャンバスを 1 つの PDF に書き出すことができる。やったことないけれど PostScript でもできるらしい。PNG はできない。

ROOT **公式ユーザーズガイド** “9. Graphics and the Graphical Userinterface : The Postscript Interface” (p139) **参照**

```
TString name;
name.Form("canv.pdf");
TCanvas *c1 = new TCanvas(name.Data(), name.Data(), 1000, 500);

c1->Print(name + "[", "pdf");     // ここで"canv.pdfを開く感じ"

for (Int_t ihist = 0; ihist < Nhists; ihist++) {
    hist[ihist]->Draw();
    c1->Print(name, "pdf")        // ここで、キャンバスを保存する
}
c1->Print(name + "]", "pdf");     // ここで"canv.pdfを閉じる感じ"
```

最後の一文を以下のように変更すれば、別の TCanvas **オブジェクトを追加して保存することができる。**

```
c2->Print(name, "pdf")
c2->Print(name + "]", "pdf")
```

10.3.1 “[“と”(“の違いについて

[　　この時点ではページを出力しない

(　　この時点でページを出力する（空白のページができる？）

第 11 章

TLegend ／ TText 編

第 12 章

TString 編

C/C++ では文字とか文字列の扱いは面倒くさいのですが、ROOT には TString という便利なクラスがあります。使わない手はないでしょう、ということで紹介しておきます。

12.1 フォーマット文字列を作りたい

```
TString str;
str.Form("Hist%d", i);
```

12.2 文字列を取り出す

```
str.Data();
```

12.3 使い方の一例

複数のヒストグラムをループで生成したいときなどによく使います。

```
const Int_t nhist = 10;
TString hname, htitle;
for (Int_t i = 0; i < nhist; i++) {
    hname.Form("h%02d", i);
    htitle.Form("%s;%s;%s", hname.Data(), "x", "y");
    h[i] = new TH1D(hname.Data(), htitle.Data(), xbin, xmin, xmax);
}
```

第 13 章

その他

ここでは、ROOT 以外の研究で役に立ちそうなことについてまとめます。

13.1 Emacs

最初に断っておくと、特に「Emacs 信者」というわけではないです。Emacs しか使えない、ただのポンコツ[*1]です。ただし、プログラミングを快適に行うにあたって「自分に合ったエディタ」を選択するというのは重要なことだと思います。思った以上に長いお付き合いになるので、愛着を持てるエディタを選びましょう。

僕の場合は修士でプログラミングを始めた時に、最初に触ったのが Emacs だったというのが主な理由[*2]ですけどね。最初はこれまで使っていたテキストエディタ[*3]と同じように矢印キーなどで操作していました。ウェブなどでキーバインドを確認しながら、だんだんと操作方法を覚えていった気がします。

最近では guru-mode なるパッケージもあるので、これを入れて練習したらいいと思います。マニュアルにある通りにインストールして、(setq guru-warn-only t) にして使うとよいです。矢印キーなどを押すとミニバッファに適切なキーバインドが表示されるので、使いながら覚えるのにピッタリです。後述する Prelude にはデフォルトで入っています（というか同じ作者）。

[*1] Vim も基本操作はできるようにしといた方がよいと思っています
[*2] Mac の Cocoa アプリも Emacs キーバインドで使えるのも大きな理由だったかも
[*3] Word やメモ帳

13.1.1 Emacs の学習コスト

快適に操作をするためにある程度キーバインドを覚えないといけなかったり、カスタマイズしようと思うと Elisp の知識が必要だったり、と学習コストは高め[*4]です。

プログラミングの統合開発環境には「Eclipse」や「Xcode」、文書作成には「Word」だったり、TEX の統合環境には「EasyTeX」や「TeXShop」などがあります。これらは用途に特化していおり、便利になるように設計されているのですが、結局、各ソフト毎の操作を覚えなくてはいけないため、実は学習コストは同じくらいなんじゃないかなとも思います。

13.1.2 基本操作

操作方法を羅列するのは無意味なので、よく使う操作について Emacs、vim、そして less を比較してみました。

ページ移動

Emacs	vim	less	操作内容
C-n	j, RET	j, RET	次の行
C-p	k	k	前の行
C-v	C-f	SPC	1 ページ進む
M-v	C-b	S-SPC	1 ページ戻る
M-<	gg	g	ファイルの先頭
M->	G	G	ファイルの最後
M-g g 数値	数値 G	:数値	指定した数値の行へジャンプ
		d	半ページ進む
		u	半ページ戻る
C-x C-c	:q, :q!	q	ファイルを閉じる

カーソル移動

Emacs	vim	操作内容

[*4] 決して安くはない

C-f	l, SPC	**次の文字**
C-b	h	**前の文字**
M-f	w, e	**次の単語**[*5]
M-b	b	**前の単語**
C-a	0	**行頭**
	^	**文頭（行頭にある文字）**
C-e	$	**行末**
C-i		**タブ（インデント？）**
C-l		**画面の移動（上-中-下）**

切り貼り

Emacs	vim	**操作内容**
C-k	d$	**カーソルの位置から行末までを切り取り**
C-w	d$, dd, dw	**選択範囲を切り取り**
	dd	**一行削除（切り取り）**
	dw	**１単語を切り取り**
	d$, d^, d0	**それぞれ切り取り**
M-w	y	**選択範囲をコピー（**yank**）**
	yy	**一行コピー（**yank**）**
	yw	**１単語をコピー（**yank**）**
	y$, y^, y0	**それぞれコピー（**yank**）**
C-y	p	**貼り付け**

検索

Emacs	vim	**操作内容**
C-s	/**文字**, n, C-i	**前方検索** [*6]
C-r	?**文字**, N, C-o	**後方検索**
C-@	v	**マーカーのセット**
M-%	:s/old/new	**現在行の最初の文字を置換（**old -> new**）**
	:s/old/new/g	**現在行のすべての文字を置換（**old -> new**）**

[*19] Emacs **の場合、**jaword **パッケージ を導入すると日本語の単語移動が賢くなります**

	:%s/old/new/gc	ファイル全体のすべての文字を、確認しながら置換

ファイル操作

Emacs	vim	操作内容
C-x C-s	:w	ファイルを保存
C-x C-w	:w ファイル名	ファイル名を指定して保存
C-x C-i	:r ファイル名	ファイル名の中身を挿入
C-d	x	カーソルの下の文字を削除（Delete）
C-h[*7]	Backspace	カーソルの左の文字を削除（Backspace）

エディタ特有

Emacs	vim	操作内容
	ESC	ノーマルモードへ切替
	i	カーソルの位置に追加
	a	カーソルの次の位置に追加
	A	行末に追加
	I	行頭に追加
	o	カーソルの下の行に追加
	O	カーソルの上の行に追加
C-j		改行
C-o		改行
C-m		改行
RET		改行
C-x u	u	直前の動作の取り消し
	U	行全体の変更の取り消し
	C-r	取り消しの取り消し
	r	カーソル下の１文字の置換
	R	カーソル下の複数文字の置換
	cw	カーソル位置の単語の変更（削除＋挿入）

[*20] Emacs の場合、cmigemo と migemo パッケージ を導入するとローマ字で日本語検索が可能になります。インストールと設定の詳細は るびきち「日刊 Emacs」 を参考にするとよいと思います

[*21] 元々は Help ですが、置き換えています

	c$	カーソル位置から行末までの変更（削除＋挿入）
	c0	カーソル位置から行頭までの変更（削除＋挿入）
	c^	カーソル位置から文頭までの変更（削除＋挿入）
	C-g	ファイル内の位置の表示
	%	対応するカッコへ移動
	!コマンド	外部コマンドを実行

13.1.3 キーボード設定

Emacs を快適に操作するためには、少しだけキーボード設定をした方が良いと思います。あんまり変えすぎちゃうと、他の PC を使うときや PC を買い替えた時の再設定がめんどくさくなるので、ほどほどに。

私の場合は以下の３点を変更しています。

	変更前	変更後
Control キーの追加	CapsLock	Control
Spotlight 検索	Control + Space	Option + Command + Space
Select next input source	Option + Command + Space	OFF

13.1.4 Emacs + Org

Emacs 標準の「アウトラインモード」ですが、それ以上のことがたくさんできます。この KumaROOT も Org-mode で作成しています[*8]。とても多機能なので、詳しくは「M-x org-info」でドキュメントを参照してください。

ここでは、この KumaROOT を作成する際に必要だったことをメモする感じにします。

LaTeX エクスポートの設定

pLaTeX（pTeX 系）と pdfLaTeX（pdfTeX 系）を使う場合を紹介しておきます。日本語を使う場合、現在は pLaTeX（+dvipdfmx）を使う設定が簡単だと思います。でも、調べてみると世の中は pdfLaTeX に向かっている気もするので、合わせて紹介しておきます。

[*8] Org-mode から LaTeX にエクスポート → YaTeX 環境でコンパイルしています。すべての作業が Emacs 内でできるので大変便利です

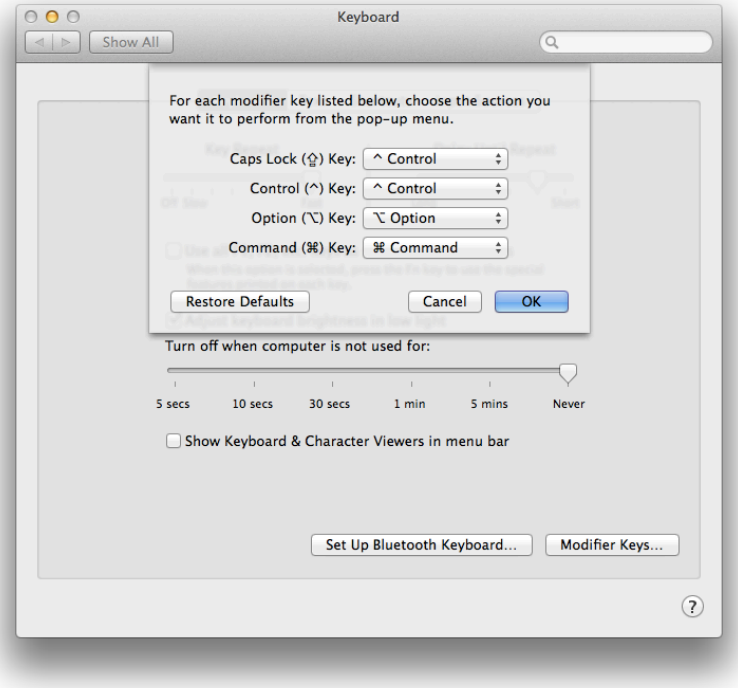

図 13.1　CapsLock → Control に変更

この辺は調べだすとキリがないので、軽い気持ちで選んだらいいと思います。

1. pLaTeX を使う場合
2. pdfLaTeX を使う場合

LaTeX エクスポートコマンド

‘C-c C-e l l (‘org-latex-export-to-latex’)’ ‘C-c C-e l p (‘org-latex-export-to-pdf’)’

使うクラスの指定

org-latex-classes **の中にあるクラスを使うことができる。新しいクラスを使いたい場合は、この** alist **に追加する必要がある。**

org-latex-default-class **でデフォルトで使うクラスを指定できる。デフォルトは** article**。**

```
#+latex_class: jsarticle
#+latex_class_options: [12pt]
#+latex_header: \usepackage{hoge}
```

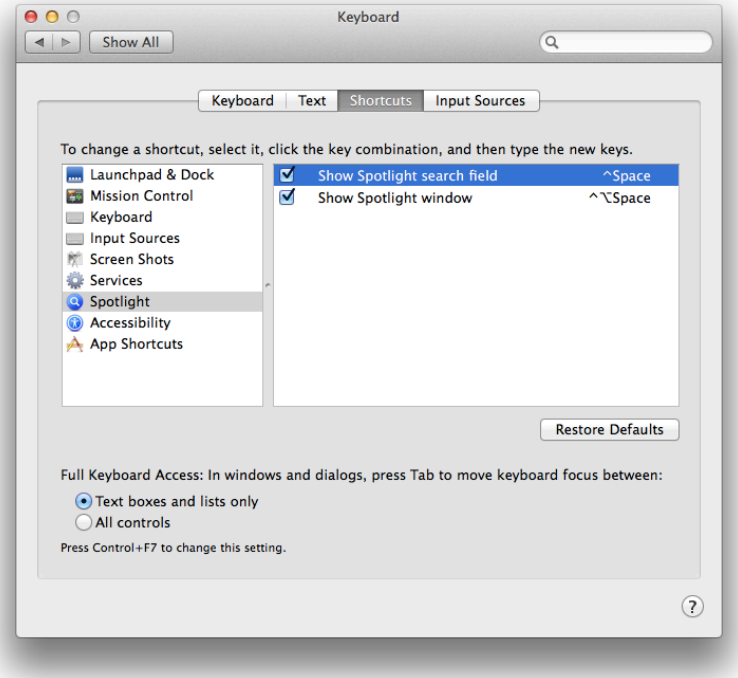

図 13.2　スポットライト検索のショートカットキーを変更する前。デフォルトは「Control + Space」

脚注の挿入

画像の挿入

説明なしのリンクは画像として挿入される。挿入のときのオプションは「#+ATTR_LATEX:」で指定できる。画像の大きさは「:width 5cm」、「:height 10cm」のようにキーワード指定できる。他は「:options angle=90」のように :option キーワードで指定できる。キャプションは「#+caption:」で指定できる。

まぁ、くどくど説明するより、以下の Org 文書での入力内容（13.1）と、それを LaTeX エクスポートしたときの出力内容（13.2）を見てもらった方が早いと思います。

Listing 13.1　Org 文書での入力内容

```
#+attr_latex:  :width 0.7\textwidth
#+caption: CapsLock \rightarrow に変更Control
[[./fig/mac-keyboard04.png]]
```

Listing 13.2　LaTeX エクスポートしたときの出力内容

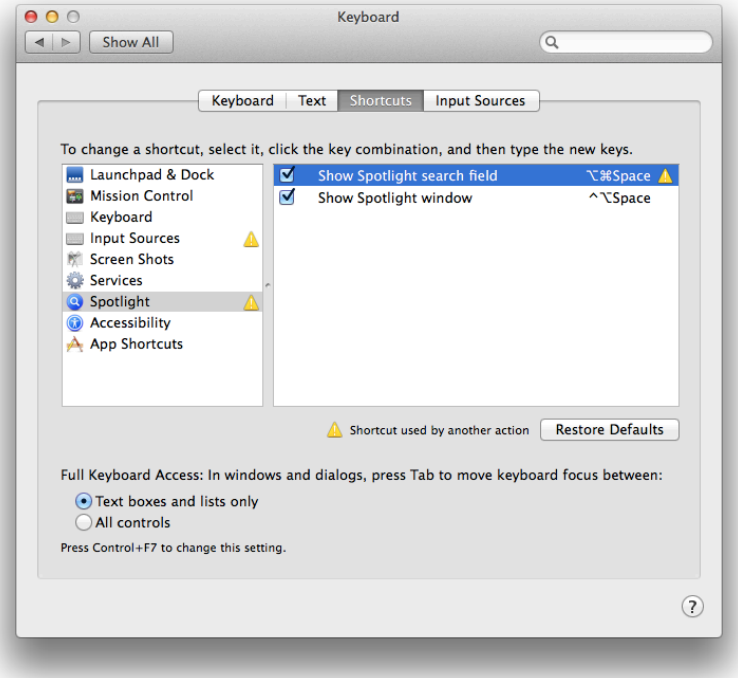

図 13.3　スポットライト検索のショートカットキーを「Option + Command + Space」に変更。重複するキーがあるため黄色い警告がでている

```
\begin{figure}[htb]
\centering
\includegraphics[width=0.7\textwidth]{./fig/mac-keyboard04.png}
\caption{CapsLock $\rightarrow$ に変更Control}
\end{figure}
```

表の挿入

13.1.5　Emacs + Prelude

Emacs **の設定を一からするのってめんどくさいですよね。そんな場合はとりあえずググってみましょう。いろんな人が、いろんな形で公開しています。**

Prelude **もその１つで、**GitHub **で公開されています。いろいろあって違いがよく分からなかったので、名前がかっこいいなーと思ってこれに決めました。ほんとそれだけです。**

複数のマシンで同じ Emacs **設定を使いたい場合は、**Prelude **を自分の** GitHub **に** Fork **して、**Clone **するとよいと思います。**

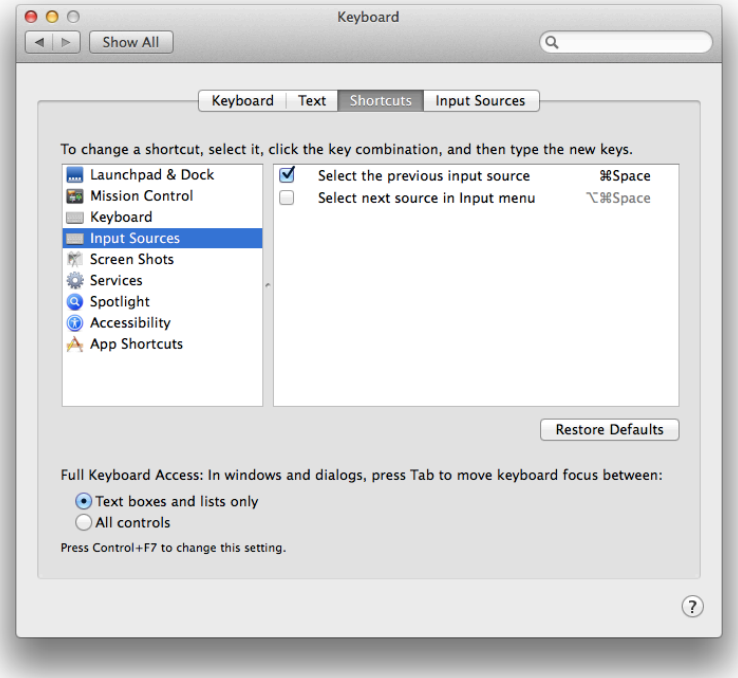

図 13.4　「Select next source in input menu」なんてショートカットキーはこれまで使ったことない。なので無効にしても問題ない

13.2　LaTeX

ほとんどの人は修論の時に LaTeX をがしがし使うことになると思います。その時に「インストールできないー」などと焦っていては時既に時間切れ[*9]なので、簡単にまとめておきます。

13.2.1　MacTeX を使おう

MacOSX で LaTeX を使う場合は、これで決まりです。TeX 環境の本体である TeXLive と一緒に、TeX 編集の統合環境である TeXShop[*10]や TeXworks、TeX 関連のパッケー

[*9] ピンと来ない人は「ブロント語」でググってください
[*10] 修論の頃はお世話になりました。現在は YaTeX に移行したので全く使っていません

ジ管理ツールである TEX Live Utility[*11]、文献管理の BibDesk、スペルチェックの Excalibur、そして、Keynote に数式を貼り付けるのに必要な LaTeXiT[*12]もついてきます。

日本語と LATEX

日本語はマルチバイトコードであるため、LATEX でコンパイルするのが難しかったみたいです。それに対処する歴史的な紆余曲折から日本語版 LATEX にはさまざまな派生品が存在します。この歴史の詳細に関しては、三重大学の奥村さんのウェブサイトをはじめ、ググってみるとよいでしょう。

つい最近までは「Mac LATEX インストール」などでググると、なんだかまとまりのない情報で溢れていました。しかし、現在はそれらを取りまとめようということで開発が進んでいるようで、これからは TeXLive 一択で良いみたいです。

TeXLive は MacPorts からインストールすることもできますが、うまく設定できた試しがありません。なので、MacTeX 公式ページ に置いてある MacTeX パッケージをダウンロード[*13]するのが一番簡単で良いと思います。

13.2.2 ptex2pdf を使おう

日本語の LATEX 文書をコンパイルするには pLaTeX コマンドを使います。コンパイルが成功すると dvi ファイルが作成されるので、dvipdfmx コマンドを使って PDF ファイルに変換します。

これをまとめてやってくれるのが ptex2pdf コマンドで、（おそらく）MacTeX をインストールすると勝手についてきます。ただのシェルスクリプトなので、気になる人は中を見てみるとよいでしょう。

13.2.3 YaTeX を使おう

MacTeX をインストールすると TeXShop.app がついてきます。すぐに使えるので、時間に余裕がないときはこちらを使うと良いでしょう。

[*11] コマンドラインから tlmgr として使えます。パッケージのインストールがとても楽ちん。ただし、TeXLive のバージョンが上がるたびに動かなくなるのでちょっとめんどくさい

[*12] これが一番重宝してます

[*13] フルパッケージは 2GB ちょいあるので、ダウンロードに少し時間がかかります。細い回線で行うのはオススメしません

設定に時間を割けるようであれば、Emacs+YaTeX をおすすめします。YaTeX の設定や操作コマンドを覚えるための時間は必要ですが、ある程度慣れてしまえば編集作業が格段に捗るはずです。

http://ichiro-maruta.blogspot.jp/2013/03/latex.html http://qiita.com/zr_tex8r/items/5413a29d5276acac3771

13.2.4 pdfLaTeX

13.2.5 jsarticle ドキュメントクラス

```
\documentclass[dvipdfmx,12pt]{jsarticle}
```

日本語の LaTeX 文書には jsarticle ドキュメントクラスを使います。ドライバは dvipdfmx を使います。

13.2.6 graphicx パッケージ

```
\usepackage[hiresbb]{graphicx}
```

画像を扱う場合は graphicx パッケージを使います。ドライバは dvipdfmx を使います。ドキュメントクラス指定時に宣言していてれば、ここで指定する必要はありません。画像のバウンディングボックスに「HiResBoundingBox」を使う場合は、hiresbb オプションを付けておきます。

13.2.7 hyperref パッケージ

LaTeX 文書内にハイパーリンクを置く場合には、hyperref パッケージを使うとよい。しかし、日本語のドキュメントクラスを使うとページから文章がはみ出たり、目次のしおりが文字化けしたりするので、以下のように対処する必要がある。

PXjahyper パッケージも読み込む

```
\usepackage{pxjahyper}
```

PXjahyper パッケージを読み込んでおけば、万事解決するみたい。それでも解決しない場合は、以下のように個別に対処する。

ページサイズ対策

```
\hypersetup{setpagesize=false}
```

しおりの文字化け対策

```
\usepackage{atbegshi}
\AtBeginShipoutFirst{\special{pdf:tounicode EUC-UCS2}}
\AtBeginShipoutFirst{\special{pdf:tounicode 90ms-RKSJ-UCS2}}
```

何をしてくのか全く分かっていないけれど、両方書いておけばいい。それでだめな場合は片方にする。

13.3 KiNOKO

CAMAC や VME **でデータ収集を行うためのドライバをインストールします。詳細に関しては「**KiNOKO **プロジェクト」を参照してください。**

```
$ cd ~/Downloads/
$ wget http://www.awa.tohoku.ac.jp/~sanshiro/kinoko-download/files/kinoko
    -2014-01-29.tar.gz
$ tar zxvf kinoko-2014-01-29.tar.gz /usr/local/heplib/
```

僕の場合、ダウンロードしたファイルはとりあえず ~/Downloads **に保存することにしています。また、高エネルギー物理関連のプログラムは** *usr/local/heplib* **以下にインストールすることにしています。**

13.3.1 CAMAC ドライバのインストール

13.3.2 VME ドライバのインストール

13.4 Geant4