2021 Q4 – 最終版

2021年度

計 算 物 理 学

東京工業大学　理学部物理学科

担当教官

2021 年度の計算物理学の授業を担当する教官とティーチングアシスタント（TA)、それぞれの連絡先を以下に示す。授業は実習形式で行うため、何か疑問に思ったり、エキスパートの補助が必要な事があれば、遠慮なく問い合わせてほしい[1]。

教官	板橋 健太（いたはし けんた）	itahashi(@)ag.riken.jp
TA	Yin Zanpeng（いん ざんぺん）	yin.z.aa(@)m.titech.ac.jp
TA	田中 智也（たなか ともや）	tanaka.t.bp(@)m.titech.ac.jp

但し (@) は @ で置き換えよ。

また、ホームページには、授業で必要となる講義ノートやレポート、休講、自習についての情報を掲載する。

ホームページアドレス	http://ag.riken.jp/cp

[1]質問するときのコツは、質問する内容を相手が再現できるよう、疑問を抱くに至った状況を含めて質問することである。最初はそのコツをつかむのが難しいかも知れないが、質問を繰り返すうちに馴れよう。

目 次

第1章　はじめに

1.1　本講義のねらい

物理学における、コンピューターの役割は多様である。書類を作成したり、遠隔地との通信を行うための汎用的なものから、例えば 数千本の信号線が伝達する情報をキロヘルツを上回るレートで記録するためのデータ収集システムや、数千を超える演算処理ユニットを用いて非線形事象の解析を行う専用のシステムまで存在する。

本講義では、コンピューターという道具を使って物理学の研究を進める基礎知識と技術を身につける。まず、コンピューターを計算機として用いるために、プログラミング言語を一つ習得する。実際のプログラミングを通じ、現代の推奨されるプログラミング作法を身につける事をめざす。その上で、物理学における計算機の役割の一端を担う、微分方程式の数値解法や、シミュレーション、初歩的なデータ解析について学ぶ。

その際重視するのは、

> (1) 正しいプログラミング作法を習得すること
>
> (2) 計算機を用いて問題を解決する手段を自分で探す能力を磨くこと
>
> (3) 計算機の出力した結果を理解する姿勢を身につけること

の三点である。

(1) プログラミング作法の習得
現代のプログラミング環境においては、プログラム全体を一人で書くことは少ない。部分部分を複数人が記述して、最後にまとめ上げる必要に迫られる場合がある。「常識的」なプログラミング作法を習得しておく事で、不要な間違いや混乱を防ぐ事ができる。

(2) 解決手段を探す能力
実際の問題を目の前にした時、様々な数値解法の経験があるのは悪い事ではない。しかし、より重要なのは、その問題に最適な手法を自ら検索し、適用できる能力を身につける事であろう。

(3) 結果を理解する姿勢
計算機の出力は、時に決定的に見える数値であったり、“きれいな”画像であったり、するわけだが、それらをそのまま “正しい” と信じてはならない。常に、手計算や物理的な考察を通して、期待される結果である事を確認する姿勢が重要である。

以下の講義ノートには、**課題 1** のように、ステップごとに課題を記載してある。課題には、**課題 10** のように黒字のものと、**課題 12** のように青字のものがあり、青字のものは提出を要する。その提出期限はホームページ (`http://ag.riken.jp/cp`) に記載する。

1.2 計算機とプログラミング言語

計算機とは、言葉通り計算する装置という意味であり、それはつまり、何らかの演算規則に従って、演算する装置の事である。演算の種類は多様で、数値の加減乗除から始まり、種々の方程式を代数的に解くようなものまである。そのうち、この講義で取り扱う範囲は、数値計算である。数値計算の手法とその適用は計算機を使うためのさまざまな基本的要素を含んでおり、他の技法の基礎ともなる。数値計算には、与えられた方程式を解き実数あるいは複素数の解を近似的に得る方法や、常微分方程式を与えられた初期条件の元に解決する方法、乱数を用いた数値シミュレーションなどが含まれる。

数値計算を計算機で実現する場合、一般に使われる手法は、まず人が容易に理解可能な “ある規則” に従った文章を作成し、次にそれを計算機を使って計算機が実行可能な形式に変換するという手法である。この “ある規則” は、プログラミング言語と呼ばれ、それに従った文章はプログラムと呼ばれる。また、プログラムすると言えば、プログラムを記述するという意味である。プログラミング言語は、目標の違いや、環境によりさまざまなものが存在し、その時々の流行も存在する。

コンピューターが登場した初期には、コンピューターを物理的に構成

する各装置を名指しで指定して演算を行った。例えばレジスタ[1]A に十六進数値 20 を格納せよとか、レジスタ A の値とレジスタ B の値を加算しレジスタ A に格納せよとか、そういう書き方しかできなかった。ハードウエアの構成そのものに基づいてプログラムするため、ハードウエアが変わると、プログラムも変更する必要があり移植性、保全性に欠けていた。反面、ハードウエアの性能を最も反映したプログラムを記述する可能性もあった。

高級プログラミング言語と呼ばれる種類の最初の言語として登場したのが FORTRAN[2]（命名は FORmula TRANslator に由来する）である。この “高級” な言語の特徴はハードウエアからある程度切り離されているという事であり、それは仮想化の第一歩であったと言える。プログラミング言語は、FORTRAN の誕生以降も様々な種類が、派生的にあるいは発生的に設計され、現在使われているものだけでも数十種類存在する[3]。

高級プログラミング言語では、数値あるいは何らかの情報を保持しうる記憶域を表す記号と、それらの演算や条件分岐などを指示するオペレータを用いて、演算と装置への命令をプログラムする。出来上がったプログラムは、ハードウエアを効率よく使って演算を行うように、コンピューターのハードウエアに応じた変換がなされてから実行される。この変換を行うのも、“とある” プログラムであり、通常はコンパイラと呼ばれる。コンパイラによって変換（コンパイル）された出力は[4]実行形式と呼ばれ、コンピューターはその内容に従ってハードウエアを制御し演算を行う。

1.3 本講義で推奨するプログラミング言語

本講義で推奨するプログラミング言語は C 言語である。より科学技術計算に寄った選択として FORTRAN、より現代風のプログラミング言語として C++ の使用も認められるが、積極的なサポートはできない。

C 言語の特徴は、まず汎用的であること。コンピューターの OS （オペレーティングシステム）と呼ばれるハードウエアの制御を司るプログ

[1]記憶素子。情報を記憶する装置で、通常 8 ビットから 64 ビットの記憶容量を持つ。CPU（中央演算処理装置）から情報を書き込んだり、読み出したりできる。

[2]1957 年に IBM（International Business Machines ） 社によって開発された。

[3]プログラムの中で最も有名なものが Hello World である。例えば Wiki を見ると、プログラミング言語の多様さが分かる（http://ja.wikipedia.org/wiki/Hello_world）。

[4]ほどんどのコンパイラは機械語のファイル、オブジェクトファイルを出力する。オブジェクトファイルは他のオブジェクトファイルやライブラリと結合されて実行形式が生成される。

ラム自体が、C 言語で記述されている場合も多く、その柔軟な言語仕様はあらゆる種類のプログラミングを可能にする。また、C 言語の習得なしに、より現代的な言語である C++, Java, Python などを習得する事は出来ないとも言える。

FORTRAN と同じく C 言語にも幾つかの “方言” が存在するが、本講義で用いるコンパイラは gcc 4 であり、一般に ANSI 形式と呼ばれる言語仕様に沿った記述が可能である。

第2章　最初の準備

本講義では、東京工業大学・南4号館3階情報ネットワーク演習室に設置された端末からTSUBAME3.0にログインして実習を行う。端末は、どれを使用しても同じ環境となるように設定されているので、自分のIDとパスワードを利用して端末を起動してみよう[1]。

2.1　一度だけ行うセットアップ

一番最初の授業では、別途配布するスタートアップガイドを利用してTSUBAME3.0での作業環境を整備してから授業を開始する。TSUBAME3.0の概要については触れないが、世界有数のスカラ型スーパーコンピューターであり、64bit SUSE Linuxという UNIX系のオペレーティングシステムを搭載している。本講義ではTSUBAME3.0をUNIXとして使用しながら実習形式で講義を進める。

UNIXは、長い歴史を持つオペレーティングシステムであり、その標準のユーザーインターフェース、つまり使用方法は、文字ベースのものである。これは、お馴染みのWindowsやMacintoshのものとは大きく異なり無愛想ではあるが、使い込めば使いやすくなるという特徴も持つ。

まず別冊のスタートアップガイドを参照しながらTSUBAME3.0のXターミナル[2]を起動してみよう。

ターミナルが起動すると、`tsubame%` のように `%` 記号で終わる文字列が表示されているはずだ。これはプロンプトと呼ばれ、`%` 記号に続いてキーボードから入力する事で、コマンドを入力することができる。ターミナルを終了するには、`exit` と打つ。

[1] IDは全て大文字で入力する必要があるので注意して欲しい。

[2] ここで言うターミナルとは正確には擬似端末である。もともとUNIXシステムには、システムに一つ以上の端末装置、則ちキーボードと文字の表示装置が接続されていたが、これらをソフトウエアで擬似的に模したのが擬似端末である。

2.2 UNIX 手引き

2.2.1 ファイルとフォルダの取り扱い

UNIX のファイルシステムは樹状構造となっており、一番上位のディレクトリはルートディレクトリ (root directory) と呼ばれ “/” で表す。ファイルには二種類があり、一つは書類や実行形式、もう一つはディレクトリである。ディレクトリは、Windows や MacOS のフォルダと同義のもので、ファイルを階層的に管理するための「箱」に相当する。現在作業対象としているディレクトリをカレントディレクトリと呼び “.” とピリオドで表す。システムにログインした直後のカレントディレクトリをホームディレクトリと呼び、“~” (チルダ) で表す。カレントディレクトリが含まれるディレクトリを親 (上位) ディレクトリと呼び “..” とピリオド二つで表す。

あるファイルを表すには二種類の表し方があり、例えば `/home/1/19OT0011/.bashrc` のように、ルートディレクトリにある home ディレクトリ内の 9 ディレクトリ内の... というふうにルートディレクトリから指定する絶対パス指定と、カレントディレクトリから始まってどこにあるかを指定する `./.bashrc` のような[3]相対パス指定がある。

UNIX システムでは、実行形式のうち標準で用意されている物をコマンドと呼ぶ。コマンドを使った操作は覚えるというより慣れるものだが、以下によく使うものを紹介しておこう。

```
tsubame% ls

Desktop/        Library/        Music/          Public/
Documents/      Movies/         Pictures/       R14SP3/
```

ls (エルエス) コマンドは、ファイルのリスト (list) を表示する。ただし、ls コマンドは通常、ピリオドから始まるファイルを無視して表示しない。ls コマンドをオプション付きで起動するため `ls -a` と入力すれば、見えなかった “.” や “..” ディレクトリも含めピリオドから始まるファイルも表示される。ls コマンドには、他にも多くのオプションが存在する[4]。`ls -l` とすれば、ファイルの大きさ、所有者や変更日時が表示され、`ls -ltr` とすれば、変更日時に従って順に表示される。

[3] この場合 ./ は省略可能で、.bashrc としても良い

[4] `man ls` する事でシステムのマニュアルを参照すると良い。マニュアル中では、スペースバーでページを進み、b でページを戻る。マニュアルを終えるには、q とする。

少しディレクトリを散歩して見よう。カレントディレクトリを表示するには

```
tsubame% pwd
/home/1/19OT0011
```

のように、pwd コマンド (print working directory) を用いる。カレントディレクトリを変更するには、

```
tsubame% cd ..
```

のように、cd コマンド (change directory) コマンドを用いる。この場合には、カレントディレクトリを一つ上位のディレクトリ “..” に移動する。

```
tsubame% cd
```

のように、cd コマンドの後に何も続けなかった場合、ホームディレクトリに移動する。

それでは、ls コマンドの実行形式を納めたファイルは、どこにあるのだろうか。ls コマンドは、/bin/ls が実行形式のファイルであり[5]、`cd /bin` として “/bin” ディレクトリに移動した後、`ls` とする事で大量のコマンドの中に ls というファイルが発見出来るはずだ。

カレントディレクトリ内に、新たにディレクトリを作成するには `mkdir new_dir` のようにする事で、*new_dir* という名前のディレクトリを作成する事が出来る。名前には、英数字、アンダースコアなどは使用できるが、スペース、日本語などは使用できない[6]。

2.2.2 サンプルプログラム

それでは、授業のために専用のディレクトリを作成して見よう。

```
tsubame% mkdir buturi
```

として、buturi という空のディレクトリを作成し、

```
tsubame% cd buturi
```

として、カレントディレクトリを、新たに作成された buturi ディレクトリに移動してみよう。

いきなりプログラムを書くのは難しいので、まずは既にある物を動かしてみよう。以下の手順に従って、サンプルプログラムをコピーし、動かしてみよう。

まず、最初にサンプルプログラムを実行するための準備を行う。UNIX

[5] `which ls` とすれば分かる。
[6] 実際には可能だがお勧めできない。

は文字ベースのユーザーインターフェースが基本であるが、グラフィカルな作業も可能である。そのための準備として先ほどターミナルを起動したのと同じ【ユーティリティ】の中に【XQuartz】というのがあるのを探して欲しい。見つかれば、XQuartz を起動しよう。XQuartz は UNIX でグラフィカルな作業環境を提供するシステムである。

次に、XQuartz 起動時に現れた、xterm の中で、以下のようにして、`$SAMPLE/chapter2` に置いてあるサンプルプログラムをコピーしてみよう[7]。xterm はターミナルとほぼ同等の機能を持つが、グラフィックシステムを使うには xterm から操作する必要がある[8]。

```
tsubame% cp $SAMPLE/chapter2/welcome* .
```

とすると、`$SAMPLE/chapter2` に置いてある名前が welcome から始まる全てのファイルをカレントディレクトリ "." にコピーする。**実行する時には `welcome*` に続いて、スペースとピリオド " ." がある事に注意せよ。**

実行形式 `welcome` を起動するには

```
tsubame% ./welcome
```

とすれば良い。何かメッセージが画面に表示されていれば成功である。

実行形式 welcome を生成する元になったプログラム、ソースプログラムを見るには

```
tsubame% less welcome.c
```

とする。場合によってはターミナル下部に ":" が表示されたと思う。これは、less というコマンドがファイルの中身を表示している状態である事を示す。この状態で、ソースプログラムを見終わったら `q` とすると、less を終了する事が出来る。less で表示している最中に、`j` とすれば下方に一行、`k` とすれば上方に一行、スペースで下方に一ページ、`b` で上方に一ページ、移動しながらファイルを表示する事が出来る。また、`/`に続けて文字を打ち込めば下方に検索、`?`に続けて文字を打ち込めば上方に検索する事が可能である。

先ほど `welcome*` をコピーした際、実は余計なファイル `welcome.o` もコピーされてしまった。不要なのでこのファイルは消去しよう。

```
tsubame% rm welcome.o
```

とすると、このファイルを消去する事が出来る。ただし、UNIX におけるファイルの消去は、"完全な消去" を意味する。ゴミ箱フォルダに移動

[7] $SAMPLE ディレクトリ以下には、授業で参考に使うサンプルコードを配置したので適宜参照すると良い。

[8] xterm を新たに開きたい時は、XQuartz のメニューバーのアプリケーションメニューからターミナルを選ぶと、起動する。

するといった緩衝的な措置はないので、消去する際は注意して欲しい[9]。

このまま buturi ディレクトリに welcome 関係のファイルを置いておいても良いが、少々分類しておいた方が後のためにもなるので chapter2 というディレクトリを作成し、そこに全て移してしまうのが良かろう。

tsubame% mkdir chapter2 として、chapter2 ディレクトリを作成した後、

```
tsubame% mv welcome* chapter2
```

のようにして、ファイルを chapter2 ディレクトリに移動しておこう。

2.2.3 ファイルの編集

それでは、手元にコピーしたファイル welcome.c の中身を少し改変してみよう。ファイルを編集するには、emacs というプログラムを使う。emacs は非常に高い機能と拡張性を備える編集プログラムで、UNIX 環境では標準的に用いられているものである。

まず

```
tsubame% cd chapter2
```

として、chapter2 に作業ディレクトリを移動し、

```
tsubame% ls
```

として、welcome.c というファイルが存在する事を確認する。もし、存在しなければ、前節までの手順を再確認して欲しい。

次に、

```
tsubame% emacs welcome.c
```

として、welcome.c の編集を開始する。すると画面には、

[9]空のディレクトリを削除するには rmdir *xx* とする。また、中にファイルのあるディレクトリを再帰的に完全に消去するには rm -rf *xx* とすれば良い。が、このコマンドの結果どういう事態になっても関知しない。

```
FILE EDIT ...

#include <stdio.h>
int main()
{
  fprintf(stderr,"****************************************\n");
  fprintf(stderr,"****************************************\n");
  fprintf(stderr,"  Welcome to \"Computer in Physics\"    \n");
  fprintf(stderr,"****************************************\n");
  fprintf(stderr,"****************************************\n");
}
```

```
-S:** welcome.c  ..%.....
```

のような画面が表示されたと思う。ここでは、“Welcome to” の表示を編集して、“Hello” に改変してみよう。

改変するには、カーソルを矢印キーを動かして Welcome to の位置に移動し、delete キーで文字列を削除し、新たな文字列 “Hello” を入力すれば良い。終わったら、左上の File メニューをマウスでクリックし、保存 (save) しよう[10]。正しく改変されている事を less を用いて確認しよう。編集が終わったら、File メニューから Exit Emacs を選択して、emacs を終了する。

これだけでは `./welcome` とした時に表示される内容を変える事は出来ない。そのためには、welcome.c をコンピューターが実行可能な形式に変換する必要がある。

```
tsubame% gcc -o welcome welcome.c
```

として、実行形式 welcome を作り直してみよう。

[10] マウスでの操作が面倒であれば（エキスパートはえてしてマウスを使わない場合が多い）Ctrl-X Ctrl-S を続けて入力すれば上書きして保存される。同様に Ctrl-X Ctrl-C によって、emacs を終了する事が出来る。

2.2.4 トラブルの時には

良く管理された UNIX は非常に安定度が高く、システムをシャットダウンするのは停電の時くらいなものだが、UNIX を使っている内に、なんらかの理由でターミナルが応答しなくなる場合がある。その場合に試みるべき事は、まず Ctrl-C （コントロールキーを押しながら、c キーを押すこと）によってプロセスの強制終了することである。それで駄目な場合、Ctrl-Z を押して、

```
^Z
Suspended
```

となれば、続いて、jobs コマンドで

```
tsubame% jobs
[1] +Suspended ....
```

のように、中断されたジョブの番号を調べ

```
tsubame% kill -9 %1
```

として、対応するジョブに KILL シグナルを送って終了しよう[11]。KILL シグナルを送る代わりに、fg コマンドで通常の実行に復帰させたり、bg コマンドでバックグラウンド実行させたりもできる。どうしようもない場合、ターミナルを終了したり、ログアウトしたり、さらには自分が実行している全てのプロセスを kill -9 -1 というコマンドで強制的に終了する方法も選択肢としてあり得る。この方法は、編集中のファイルなどはセーブしてから行うように気をつけたほうが良いだろう。自力で回復困難な状況になれば、周囲で助け合ったり、教官、TA に助力を頼むと良い。

UNIX で良く用いられるコマンドについては、付録 A.1 に簡単にまとめたので適宜参照してほしい。

[11] この場合の %1 の 1 は、先の表示で [1] と表示されたジョブ番号と対応している。

第3章　数値計算 事始め

3.1　C 言語プログラミング

まず最初に、C 言語のプログラム例と、実行形式への変換方法を記載するので、前章で紹介したエディターを使って、サンプルプログラムを作成してみよう。最初に buturi ディレクトリに、chapter3 というディレクトリを作成し、chapter3 ディレクトリに移動した後、ファイル example1.c を[1]エディターで作成しよう。

サンプルコード 3.1: example1.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6    int i;
7    double x;
8
9    x = 0.5;
10   printf("Ans=␣%f\n",asin(x));
11   return(0);
12 }
```

上記のプログラムでは、1 行目と 2 行目は、システムに装備された、標準入出力ヘッダーファイル（stdio.h）[2]と、数学ライブラリヘッダーファイル（math.h）を読み込むという意味で、キーボードや端末への入出力関数（このプログラムの場合 9 行目の printf() 関数は、端末への出力を意味する）と、数学関数（9 行目の asin()= 逆正弦関数）の使用時にそれぞれ必要となる。4 行目は、main() 関数の宣言。C 言語では、main() は特別な名前の関数である。C 言語のプログラムには、それぞれ一つだけ main() 関数が存在しなくてはならない。実行形式の演算時にはまず最初

[1] C 言語のファイル名は、普通 小文字の c を拡張子として持つ。
[2] 普通 スタンダード　アイオー　ヘッダー　ファイルと呼称される。

に main() 関数の内容から実行される。この例の場合、整数を返す main() 関数の記述がここから始まる。

`int i;` と書いてあるのは整数の記憶領域 `i` という意味である。他にも倍精度実数 `double` や文字型などの変数が存在するが、これら変数の「型」は表 3.1 にまとめた。

5 行目 main() 関数の中身を包括する始まりの括弧

7 行目 倍精度の実数（double）型の変数 x を用意する。

9 行目 変数 x に 実数 0.5 を代入する。

10 行目 printf(...) 関数は ... の内容を端末に表示する。ダブルクォート (") で囲まれた内容が表示されるが、そのうち `%f` のように `%` から始まる部分は、カンマ “,” の後に続く内容が順次置き換えて表示される。今の場合は、$x = 0.5$ の逆正弦（asin() が関数名である）が`%f` の部分に表示される。このような`%` から始まる部分は書式指定子と呼ばれ、表 B.1 にまとめた。`\n` は、改行を意味する。

11 行目 main() 関数の実行結果として 整数 0 を返すという意味（ここでは深い意味はない）

12 行目 5 行目と対応し、main() 関数の中身を包括する終わりの括弧である。

プログラムを実行形式に変換するには、コマンドラインから以下のようにする。（以下 `tsubame%` はプロンプトである）

```
tsubame% gcc -c example1.c
tsubame% gcc -o example1 example1.o -lm
```

これにより、実行形式 example1 が生成される。この実行形式は、

```
tsubame% ./example1
Ans=0.523599
```

のようにして演算を実行できる。

課題 1

上記のプログラムを改変して、0.5 の逆余弦（関数名は acos()）を求めよ[1]。

[1] MacOS では、バックスラッシュ “\” を入力するには、option キーを押しながら、“\” キーを押すと良い。

3.2 変数

プログラム中では、(数) 値を保持するために変数を利用する。変数には「型」と呼ばれる属性があり、型に応じて整数や実数、文字列などを保持する。

変数は使用する前に宣言しなければならない。宣言できるのは、プログラム中の中括弧で囲まれた範囲、ブロックの先頭部分のみである。

型には種類があり、保持できる値の範囲や精度、長さにより使用すべき型が異なる。以下の表は、それらをまとめた。整数であれば int 型、実数であれば、double 型を使えば良い。

型	型名	サイズ (byte)
整数	short	2
整数	int	4
整数	long	8
整数	long long	8
実数	float	4
実数	double	8
実数	long double	16
文字	char	1

表 3.1: 64bit Linux 上の C 言語で使用可能な主な型。整数にはそれぞれ符号ありと符号なしがあり、後者は (unsigned int) のように unsigned を付けて表記する。サイズに応じて表現できる数値の範囲は異なる。例えば符号付き short は $-32,768 \sim 32,767$ の範囲だけ表現できる。

3.3 スコープ

C 言語では変数が有効な範囲＝スコープというものがある。変数はブロックの先頭で宣言しなくてはならないが、例えば、以下のサンプルコードを見ると main() 関数のブロックの中にある if 文のブロックで倍精度実数の変数 x が宣言されている。この変数はそのブロック内部でだけ (つまり入れ子の中だけで) 有効である。このようにブロック内でだけ有効な変数を自動変数と呼ぶ。一つのブロックで同じ名前の変数を二つ以上宣言する事は出来ない。

```
#include <stdio.h>
#include <math.h>

static double Sum;

 int main()
 {
   ...
   if(y==0){
      double x;
      ...
      x = 10.0;
      y = 100.0;
   }
 }
```

異なるブロック内で同じ変数名があっても、それらは別物として取り扱われる。入れ子になったブロックの場合には、入れ子になったブロックの “内側で定義された変数” が優先される。

ファイルの中どこでも有効なのが、最も有効範囲の広い大域変数である。#include <math.h> の直後など、関数に含まれない部分で宣言する事により大域変数を宣言する事ができる。上記の例では、static double Sum; として宣言されている部分である。大域変数は、ファイル内部のどこでも有効であり、便利であるが、関数の独立性を損ねるので注意して使う必要がある。

TIPS
さきのプログラム example1.c で、例えば 5 行目に対して 6 行目が字下げされ、少し右側から書き始められているのはインデントと呼ばれる。この場合 5 行目の中括弧が示すあるまとまった区間、ブロックがある事を視覚的に表現している。括弧が増えてくると対応が見えにくくなる場合があるので、習慣上字下げを行うが、コンパイラには改行があろうと、字下げがあろうと関係ない。
よって上記のプログラムを

```
#include <stdio.h>
#include <math.h>
int main(){double x;x=0.5;
printf("Ans= %f\n",asin(x));return(0);}
```

のように書いたところで動作に影響はない[1]。しかし、これでは読みやすいとは言えない。インデントは、初歩的なミスを防ぐのに有効であるし、後にプログラムを見直す際の可読性にも大きく影響を与えるので、正しいインデントの作法を身につけよう[2]。

[1]但しコンパイラが解釈する文である`#include` 文は一行に一つしか許されない。

[2]Emacs のような高機能編集環境では、ほぼ自動でインデントを付ける機能を持っている。

3.4 四則演算と数学関数

変数同士の加減乗除はそれぞれ `a+b; a-b; a*b; a/b;` のようにする。整数の剰余は `a%b;` とする。一つ注意が必要な点は、整数を整数で割った商は整数となりその際、小数点以下が切り捨てられることだ。例えば 14/3 は 4 であり、**97/120 の結果は 0 である。**このような場合に、小数点以下を計算したい場合は、97.0/120 のように小数点をつけて片方を実数として表現すると良い。これは、整数の変数同士を計算する場合も同様で、整数変数 int i,j を用いて i/j を計算すると、上記と同様の切り捨てが発生する。この場合には (double)i/j のようにすればよい。変数の前に (double) と書くのは、続く変数の値を倍精度実数と解釈せよという指

示を表す。倍精度に変換してから割り算する事で、小数点以下を切り捨てるような問題を回避出来る。

	記号	計算例
和	$+$	$x+y$
差	$-$	$x-y$
積	$*$	$x*y$
商	$/$	x/y
剰余	$\%$	$x\%y$

表 3.2: 加減乗除と剰余の計算

(double) の部分は 型変換とかキャストと呼ばれ、上記のように明示的に変換する場合もあれば、暗黙の内に型変換が行われる場合もあるので注意したい[3]。

C 言語には、たいていの数学関数が用意されている。よく使うものを以下の表 3.3 に示す。

	関数	意味
累乗	pow(x,y)	x の y 乗
平方根	sqrt(x)	$\sqrt{x}$
指数関数	exp(x)	e^x
自然対数	log(x)	$\log x$
正弦	sin(x)	$\sin x$
絶対値	fabs(x)	$\|x\|$

表 3.3: 数学関数の例

3.5 制御構文

3.5.1 for ループ

C 言語では、容易に同じような計算を繰り返し実行する事が出来る。例えば、整数値を 0 から 9 まで加算するプログラムは、以下のように記述

[3]「暗黙の型変換」で検索すると沢山の例が見つかる

できる。

サンプルコード 3.2: example2.c

```
1  /*****************************************
2   Example
3                Author K. Itahashi
4
5  *****************************************/
6  #include <stdio.h>
7  #include <math.h>
8
9  int main()
10 {
11   int i;
12   double sum;
13
14   sum = 0.0;
15   for(i=0;i<10;i++){
16     sum = sum + i;
17   }
18   printf("Total␣is␣%f\n",sum);
19 }
```

まず、1 〜 5 行目までがコメント（/* と */ で囲まれた部分と、// から改行までの部分はコメントと呼ばれコンパイラは内容を無視する）である[4]。

この例では、for ループを用いて、ループ変数 i を 0 から 9 まで変えながら、いちいち sum との和をとり、sum に代入する事を繰り返している。`sum = sum + i` は、数学的には奇妙な数式であるが、プログラミング言語の文法では、sum という変数に対して新たに【現在の sum の値と x の値の和】を代入せよという意味である。

for 文は、

```
for(初期化; 継続条件; ステップ文){
    ループ内部の文;
}
```

のような形式で、最初の一度だけ初期化を行い、継続条件が満たされる限り、ループ内部の文を実行してから、ステップ文を実行する。ステッ

[4] もっとも// は ANSI 標準ではない。

プ文中の i++ は i=i+1 の省略形であり、変数 i を 1 ずつ増やす事に対応する。

for ループのループ変数には、実数を用いる事も出来る。

サンプルコード 3.3: example3.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  static double DX=0.01
5
6  int main()
7  {
8      int i;
9      double x;
10     double sum;
11
12     sum = 0.0;
13     for(x=0;x<=5.0;x=x+DX){ // for loop
14         double y;
15         y = cos(x);
16         sum = sum + y;
17     }
18     printf("Ans␣=␣%f\n",sum);
19 }
```

この例[5]では、変数 x を DX (= 0.1) ずつ増やしながら、y = cos(x) を計算し、x<5.0 の間 y の和を計算、最後に総和を画面に表示する。ここで、DX は 4 行目で大域変数として宣言されている。プログラム全体の振る舞いを決めるパラメーターのような変数は、大域変数として宣言する場合が多い。13 行目の//以降はコメントとして、プログラム上では無視される。

課題 2

上記のサンプルコード 3.3 を参考にして、積分 $\int_0^h \sin^2(x)dx$ の近似値を数値和を用いて計算せよ。$\sin^2(x)$ は、`pow(sin(x),2.0)` と記述する。但し積分の増分 (DX) は 0.01 とし、積分区間上限は $h = 3.14/2$, 3.14 の 2 通りとする。誤差はどの程度か、DX がより小さい場合どうなるか。

[5]一カ所構文間違いがあるので、注意して見て欲しい。

3.5.2 if 文による条件分岐

ある条件が満たされたときに何かを実行したい場合には、制御文を用いる。たとえば、if 文を用いて条件分岐するには

```
i = 1;
if (i==0){
  x = 10.0;
}else{
  x = -10.0;
}
```

のように記述すれば良い。“==” 演算子は左右の値を比べ、同じ時にだけ真となる。真の場合には、if 文の直後のブロックが実行され、偽の場合には else 以下の文が実行される。この例では、当然 x に -10.0 が代入される事になる。よく使われる論理演算子には < や > があり、それぞれ左辺値と右辺値の大きさを比べる。>= や <= のような記述も可能である。

詳しくは付録 B.3 を参照すること。

3.6 関数

続けて、C 言語でプログラムを構造的に記述するための道具立てについて述べておこう。ここで紹介するのは、関数である。プログラム中で関数はほぼ独立した機能を抽出した機能の集合体とでも表現されるものであり、プログラムを機能ベースでまとめるのに役立つ。

関数の簡単な例は、既に登場している `acos()` である。これは、システムが標準で用意している数学ライブラリ内で関数として提供されているもので、acos(x) とすれば、x の逆余弦を値として返す関数であった。同様にユーザーが自分でひとまとまりの機能を関数として記述することも出来る。このような関数をユーザー定義関数と呼ぶ。

ユーザー定義関数は、以下のような形式で記述される。

サンプルコード 3.4: example4.c

```
1 #include <stdio.h>
2 #include <math.h>
3
```

```
4  double u_tan(double dx, double dy)
5  {
6    double dz;
7      ...
8      ...
9    dz = tan(dy/dx);
10   return(dz);
11 }
12
13 int main()
14 {
15   ...
16   ...
17   z = u_tan(x,y);
18   ...
19 }
```

まず 4 行目で、倍精度実数 (double) を値として返す関数 u_tan を記述する事を宣言する。この u_tan は、入力として倍精度実数 dx と dy を受け付ける。これらの入力変数の事を引数と呼ぶ。関数内部では何らかの計算がなされ、１０行目で変数 dz の値が関数 u_tan の値として呼び出し元に返される。

関数を使用する側では (呼び出し側とも言う) 以下のようにする。27 行で実際に u_tan() 関数に x, y の二つの実数を渡し、z に値を戻している。

このように関数を使う事で、機能ごとにほぼ独立した構造を持ったプログラムの作成が可能になり、プログラムの見通しが良くなったり、関数ごとに機能を確認しながらプログラムする事が出来るようになる。

以下では、関数を用いることで、プログラムの構造化を意識しながら、プログラムを記述するようにしよう。

課題 3
課題 2 のプログラムを改良し、$f(x) = \sin^2(x)$ をユーザー定義関数として記述し、[0.0:π] の区間で積分せよ。ただし微小単位 DX は大域変数で与えること。

3.7 数値計算による円周率の導出

小学校以来お馴染みの無理数である円周率 π は様々な形で定義され、数値計算ライブラリでは 定数 M_PI として利用可能だが、ここでは敢えて 1 の逆正接の 4 倍 = 4× arctangent（1） として定義しよう。C 言語では

逆正接は数学ライブラリに用意されているが、試しに古人に倣(なら)って級数を用いて数値計算する事を試みよう。逆正接を、原点の周りに テイラー展開すると

$$\text{arctangent}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} x^{2i+1} = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \frac{1}{9}x^9 ... \quad (3.1)$$

の様にテイラー級数として書ける。これに $x = 1$ を代入することで、$\pi = 4 \times \text{arctangent}(1.0)$ を求める。無限次までの和は、計算機上に表現することが出来ないので、ここでは打ち切りの次数 N を設定してその次数までの和をもって近似する事とする。

課題 4
上記を参考にして、テイラー級数を用いて π を求めるプログラムを作成せよ。但し、$\sum_{i=0}^{N} \frac{(-1)^i}{2i+1} x^{2i+1}$ の $f_i(x) = \frac{(-1)^i}{2i+1} x^{2i+1}$ の部分はユーザー定義関数として記述し、最初 N は 100 とし大域変数として与えること。
C 言語の数学ライブラリ (atan()) を用いた正確な値と、テイラー級数による値の差の絶対値を計算し、精度と打ち切りの次数の関係を示せ[1]。

[1]絶対値を求める関数は fabs() である。

3.8 円周率計算の精度

それでは、次に課題 4 の方法で求めた π の精度について議論しよう。テイラー展開の式から分かるように、この数値計算の精度は、級数を部分和で近似する打ち切りの次数 N を用いて $\sim O(1/N)$ である。この事は、C 言語の数学ライブラリを用いて正確に計算される逆正接と、有限次までのテイラー級数の和との差を計算する事で確認される。

数学ライブラリの逆正接を表す関数を、調べるにはどうしたら良いだろうか？一つの手段は、既に知っている逆余弦 acos（）や逆正弦 asin（）関数を用いてコマンドラインから

```
tsubame% man acos
```

として、システムのマニュアルを利用する事である。マニュアルを終了するには、q とタイプすれば[6]良い。マニュアルの SEE ALSO の節に、atan (3) とそれらしい名前[7]を見つけたら

```
tsubame% man atan
```

として、使用方法を確認すると良い。

マニュアルを使った別の方法としては、

```
tsubame% man -k tangent
```

として、tangent というキーワードで検索する手法もありうる。

またこれらに加え、ネットワーク検索は自力で正解に到達するための非常に強力な味方である。

コマンドラインで、実行形式を実行する際、

```
tsubame% ./example1 > example1.dat
```

のように、> 記号を用いて[8]端末への出力をファイル（example1.dat）に格納する事が出来る。

---

[6]また、マニュアルの上部を見たい場合は k 下方は j をタイプすれば良い。マニュアルは、内部で less を起動するのであるが、less については付録を参照せよ。

[7]atan（3） の 3 は、マニュアルのセクション 3 に記載されているという意味で、セクションを指定して man を tsubame% man 3 atan の様にする事も出来る。これは例えば、printf のように複数のセクションに異なる内容で同じ名前が記述されている時に、指定する必要がある。man printf としても、C 言語の printf（）関数のマニュアルは表示されない場合が多い。課題 1 で登場する C 言語の printf（）関数のマニュアルは、man 3 printf とすると見ることが出来る。C 言語の関数のマニュアルはセクション 2 か 3 に記載されている。

[8]リダイレクトと呼ぶ。

課題 5
課題 4 を変更し、打ち切りの次数を 10000 まで 1 ずつ変化させながら、精度を計算し画面に書き出せ。

```
1 0.858407346410206884
2 0.474925986923126153
3 0.325074013076873669
4 0.246354558351697506
5 0.198089886092747136
6 0.165546477543616621
7 0.142145830148691310
8 0.124520836517975297
...
...
```

のような、打ち切りの次数と差” の数字列だけからなるファイルを作成せよ。付録 C を参照し、gnuplot を使ってファイルのデータをグラフとして可視化せよ。

課題 6
課題 4 の級数は、収束は遅いが正しく円周率を求められることが分かった。円周率を求める級数には様々なものがあるはずである。そのようなものを一つ探し、プログラムを作成して、実行せよ。本章で行ったのと同じ様に、数値計算の精度、収束速度について可視化して考察せよ。

第4章　数値表現と計算誤差

物理学に限らず、数理的なモデルを計算機上で扱う際には、それを計算機が理解できる形式にする必要がある。本章では、その際に必要な知識として

1. 計算機内での数の表現
2. 数値計算における誤差

について簡単に紹介する[1]。

4.1　計算機における数値表現

この授業ではこれまでに、数値を表す型として「整数型 (int, long long など)」と「実数型 (double, float など)」を用いてきた。これらの型はいずれも、いくつかの「ビット (bit: binary digit)」を集めたものを表している。

ビットとは情報を表す最小単位で、1 つのビットは (0, 1) の 2 値いずれかをとることで 2 通りの状態を表す。ゆえに、ビット N 個を集めたもので 2^N 通りの状態を表すことができる。計算機上で数値を扱うということは、これら「2^N 個の状態」と「実数」との対応をつけていることに他ならない。

4.1.1　数値表現の実例：整数型

正の整数 X を N 個のビットを使って表現するには次のようにする：

[1] 本章は主に河田鷹介によって記述された。

正整数の二進表現

$m \quad (N \geq m \geq 1)$ 桁目のビットを、$X/2^{m-1}$ の商が奇数のときは 1、偶数の時は 0 とする。ただし割り算は小数点以下切り下げとする。
例：

$$5/2^2 = 1, \qquad 5/2^1 = 2, \qquad 5/2^0 = 5$$
$$\text{よって} \quad 5 = 2^2 + 2^0 \to 101 \tag{4.1}$$

0 から始まる連続した整数を表すとき、N ビットで表現できる正整数の範囲は $0 \leq X \leq 2^N - 1$ となる[1]。
練習問題 [1]
C 言語において、unsigned int 型、unsigned long 型と unsigned long long 型が表せる最大の整数はそれぞれいくつか。

整数型の上端と下端は「繋がっている」。例えば、その型での最大値に 1 を足すと、その型での最小値になる。

[1] 全部合わせて 2^N 個

おそらく馴染みがないだろうと思われるのは符号 (sign) の表現である。二進数で正負両方の整数を表現する方法にはいくつかの流儀があるが、よく使われるのは「2 の補数」を用いる手法である。

2 の補数:負数表現

X を N ビットの 2 進正整数とする。X の「2 の補数」を $\bar{X}$ とすると[1]、

$$\bar{X} = 2^N - X \tag{4.2}$$

こうして得られた $\bar{X}$ を $-X$ とみなすのが一般的な流儀である。

[1] この記号は一般的な表現ではないので注意すること

この一見まわりくどい方法を用いる利点の 1 つは、$\bar{0} = 0$ となることである。よりシンプルな、「N ビットのうち 1 ビットを符号に用いる」方法では、ゼロの表現として “+0” と”−0” の 2 つが可能であるが、本来 1 つ

の数に対し複数の表現があるのは好ましくない。$-0 = +0$ となる 2 の補数法ならこの問題を回避できる[2]。

練習問題 [2]
int 型、long 型、long long 型が表せる整数の範囲はそれぞれどれだけか。

最大値 $=-$ 最小値 となっていないことに注意。

4.1.2 数値表現の実例：実数型

本節では、実数型の内部構造を追いながらそれを説明していくことにする。

2 進小数

整数を表すのと同様に、小数も 2 進数で表すことができる。

(正の) 小数の二進表現

X を正の実数とする。簡単のために $X < 1$ とする。N 個のビットを用いて X を表現するには、$-m \quad (1 \leq m \leq N)$ 桁目のビットを、$X * 2^m$ の積が奇数のときは 1、偶数の時は 0 とする。ただし積は小数点以下切り下げとする。
例：

$$0.625 * 2^1 = 1, \qquad 0.625 * 2^2 = 2, \qquad 0.625 * 2^3 = 5$$
$$\text{よって} \quad 0.625 = 2^{-1} + 2^{-3} \to 0.101 \tag{4.3}$$

整数の時と同様に、表現できるのは有限の範囲の数のみである。上記の方法で表現できる最小の数は、$X * 2^N = 1$ となる X, すなわち 2^{-N} である。

[2] 2 の補数を用いるもう 1 つの (より重要な) 利点は、減算を「負数の加算」で処理できることであるが、本章では説明は割愛

2進小数はかなり「飛び飛び」な数である。2進小数で表せる数の範囲は、実数全体から見ればかなり狭い一部に過ぎない。どのような範囲かは次のようにしてわかる。

X を N ビットの2進小数とすると、X は

$$X = \sum_{k} 2^{-k} \quad (k : \text{適当}) \tag{4.4}$$

と書ける。この式の右辺の分数を通分するとその分母は必ず $2^n\,(n \leq N)$ の形をしている。この事から次のことがわかる。

2進有限小数の領域

1未満の実数 X が N 桁未満の2進有限小数で表せる条件は、X が有理数であり、かつその既約分数表現の分母が2の冪乗で、さらにそれが 2^N 以下となることである。
例：

$$\begin{aligned} & 0.1 = \tfrac{1}{10} \\ \rightarrow \quad & 0.1 = 0.0\dot{0}01\dot{1}... \quad \left(= \frac{3}{32}\left(\frac{1}{1-\frac{1}{16}}\right)\right) \end{aligned} \tag{4.5}$$

0.1ですら2進数では無限小数となる。

ある数が2進有限小数ならばその数は10進有限小数でもあるが、逆は成り立たない。

練習問題 [3]
何故か？

4.1.3 浮動小数点数

前節では2進法を用いて1未満の小数を表現する方法を説明した。1よりも大きな小数を表現するときも同様の手法を用いるが、「小数点」の扱い方に応じて2通りの流儀が存在する。

浮動小数点数は、読者の方には馴染深いであろう「科学的表記法」に近い方式であり、数 X を次のように分解して表現する:

$$X = \text{sign} \times B \times 2^E \tag{4.6}$$

それぞれの記号の意味は次の通り：

- sign: 符号。± 1 のいずれか。
- B: 仮数部 (mantissa)。2 進小数。
- E: 指数部 (exponent)。2 進整数。

E に多くのビットを割り当てることで、表現できる数の範囲を大幅に広げることができる。

4.2 計算誤差

誤差とは何か、をきちんと説明するのは難しい。当てはまらない場合も多々あるが、ここではひとまず「本来求めたい数値と、実際に得られる数値との隔たり」とでもしておくことにする。

4.2.1 誤差の分類

誤差を、その発生段階・要因などで大まかに分類してみると、次のリストのようになる。

- モデル誤差
- 近似誤差
- 計算誤差
- 偶然誤差 (この章では扱わない)
- 系統誤差 (この授業では扱わない)

モデル誤差

物理学に限らず、自然科学というものは自然現象を数理的なモデルに落とし込んで (モデル化) 分析するものである。このモデル化の際に捨象される要素がもたらす誤差がモデル誤差である。

> 例:
> ニュートン力学は現実世界のモデルであるが、物体の速さが増加するに従って次第に現実を反映しなくなる。これは、ニュートン力学が速さの上限が有限値 (c) であることを捨象したモデルだから。

物理現象を扱う限り、モデル誤差は必ず付きまとう。

近似誤差

ほとんど名前で説明が尽きている感があるが、近似誤差とは近似式を用いることによる誤差である。

> 例:
> $\sqrt{1.0201} = \sqrt{1 + 0.0201}$ をよく使われる近似式 $\sqrt{1+x} = 1 + x/2$ を用いて計算すると $\sqrt{1 + 0.0201} \sim 1.01005$ となるが、実際には $\sqrt{1.0201} = \sqrt{1.01^2} = 1.01$ である。このとき、$1.01005 - 1.01 = 0.00005$ が近似誤差。

計算誤差

計算機で計算を行うことによる誤差の中でも特に、精度の有限性に起因する誤差を計算誤差と呼ぶ。

本章後半では、この計算誤差について詳しい説明を行う。

4.2.2 計算誤差

計算誤差は、計算機上では処理の回数や数値の範囲・精度が有限になってしまうことに端を発する種々の誤差の総称であり、起こりかたの細かな違いによる分類がある。

- 打ち切り誤差
- 丸め誤差
- 桁落ち
- 情報落ち

4.2.3 打ち切り誤差

極限 $\lim_{\delta\to 0}$ や級数和 $\sum^{\infty}$ は、正しい値を求めるためには無限大/小を用いた計算あるいは無限回の操作を行う必要がある。しかし計算機で扱える事物は有限のものだけである。極限や級数和を求める際には、無限を有限で打ち切って妥協を図る必要がある。その際に生じる誤差の事を打ち切り誤差と呼ぶ。

例:
この講義の始めの方にも登場したライプニッツの公式:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \tag{4.7}$$

右辺の無限級数を実際に無限個すべて計算できれば、exact な $\pi/4$ の値が求められるだろう[1]が、それには無限の時間がかかる。そこで我々は実際には計算を途中で打ち切って、すなわち有限個の和を計算して、その値を以って $\pi/4$ とする。この時「計算されなかった部分」の寄与が打ち切り誤差である。

[1]精度の問題は抜きにして。

4.2.4 丸め誤差

数値を有効数字内に収めるときに有効数字範囲外の部分が捨てられることによる誤差。

先に見たように、実数型の変数は実数を表現する能力をもたず、ほんの一部の有限小数を飛び飛びに表せるにすぎない。よって、ほとんど全

ての数は、実数型として扱われるときにその数に最も近い表現可能な数に変換されてしまう。

と書いてもわかりづらいので、次の例では 10 進数で 5 桁の有限小数を使って実数を表す場合に何が起こるかを示す。

例:
有効数字が 5 桁のとき、0.3333333...→ 0.33333 となる。このとき、0.0000033... が丸め誤差。

電卓などで $1/3*3$ を計算させると 1 にならない、というのも丸め誤差が原因である[1]

[1]かしこいシステムだと、先に計算式全体を簡約化してから計算するので 1/3*3 くらいなら 1 にしてくれるかもしれない。しかし見かけ上 1 でも切り上げの結果だったりするかもしれない。

プログラム例

```
double a = 1, b = 3;
float c;
c = a/b; // "1/3"
printf("%1.15f\n", c*b);
```

出力:

```
1.00000002980232
```

c が float 型なのは、切り上げ処理などで 0.999... が 1 と表示されるのを回避するため。

丸め誤差にはある意味での上限が存在する (おまけ参照)。

4.2.5 桁落ち

非常に近い 2 つの値を引き算することで起きる誤差。この誤差の厄介なところは、値が近ければ近いほど被害が大きくなることと、従って回避が難しい (不可能なこともある) こと。

計算結果に派手なエラーが起きている場合は桁落ちの可能性もある。

例:
有効数字 5 桁の数 $a = 0.12345678...$ を $b = 0.12346789...$ から引くと、

$$\begin{aligned} b - a &= 0.12346789... \\ &- 0.12345678... \\ &= 0.00001111... \end{aligned} \tag{4.8}$$

ところが、計算結果で信頼できるのは小数点以下 5 桁目までである。よって結局、計算結果の有効数字は最良でもたった 1 桁に落ちてしまう。

一般に有効数字 N 桁の場合、M 桁目までが一致している 2 数の引き算により有効数字は最良でも $N - M$ 桁になる。

プログラム例:
$\sqrt{10001} - \sqrt{10000}$ を計算する。そのまま計算したのが a の値、$x^2 - y^2 = (x+y)(x-y)$ を用いて変形し、引き算を回避したのが b の値[1]。

```
  float a=0, b=0;
  float c=sqrt(10001), d=sqrt(10000);
  a = c-d;
  b = 1/(c+d);
  printf("%1.7f\n", sqrt(10001));
  printf("%1.7f\n",  sqrt(10000));
  printf("%1.7f\n", a);
  printf("%1.7f\n", b);
```

出力:

```
c: 100.0049999
d: 100.0000000
a: 0.0049973
b: 0.0049999
```

b の値は a よりも正確な値に近い。有効数字が 7 桁であることを考えると桁落ちを回避できていると考えてよい。

[1] 桁落ちを回避することは一般に難しく不可能な場合もある。

4.2.6 情報落ち

“積み残し” とも呼ばれる。桁落ちとは逆に、絶対値に大きな差のある 2 数の足し引きを行った場合小さな方の影響が無視されてしまうことによる誤差。

例:
有効数字 6 桁の数 $a = 0.123456...$ と $b = 0.123456... \times 10^{-8}$ を足すと、

$$\begin{aligned} a+b &= 0.123456... \\ &+ 0.0000000123456... \\ &= 0.123456... = a \end{aligned} \tag{4.9}$$

となり、b を足さなかったのと変わらない。

プログラム例:
級数[1] の部分和を、始めからと終わりからの 2 通りの足し方で計算する

```
#include <math.h>
#include <stdio.h>
int main(){
  double a=0, b=0;
  int N = 1000000;
  double i;
  for(i = 1; i < N; ++i)
    {
      a += 1.0/pow(i,4);
      b += 1.0/pow(N-i,4);
    }
  printf("%1.15f : %1.15f\n", a, b);
  printf("%1.15f : %1.15f\n",
         sqrt(sqrt(90*a)), sqrt(sqrt(90*b)));
}
```

出力:

```
1.082323233710861 : 1.082323233711138
3.141592653589592 : 3.141592653589793
```

a の方は情報落ちが起きている。b でも起きているがその度合は少ない為 $a < b$ となっている。

[1] $\zeta(4) = \pi^4/90$.

実数型の比較

計算誤差の累積により、理論上は等しいはずの 2 つの数が計算機上では等しくない、ということがしばしば起こる。実数値同士が等しいかどうかを見るときは、まず「2 数がどれだけ近ければ等しいとみなすか」の許容値を決め、2 数の差の絶対値がその許容値より小さいかどうかで判定

すると良い。

4.3 課題・数値微分

元の関数からその導関数の値を数値的に求めることを数値微分と言う。微分の定義からすれば、

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} = \frac{f(x+\Delta) - f(x)}{\Delta} \tag{4.10}$$

として Δ を十分に小さくとれば、いくらでも正確な $f'(x)$ の値が得られるように思えるが、事態はそう単純ではない。
導関数が簡単に求められる関数を用いて導関数の解析的な値と数値微分による値を比較し、その精度を確かめてみよう。

課題 7

a) 関数 $f(x)$ の $x = 4$ における導関数の値を、前進差分式

$$f'(x) = \frac{f(x+\Delta) - f(x)}{\Delta} \quad (\Delta > 0) \tag{4.11}$$

を用いて数値的に求め、解析解と比較せよ。ただし $f(x) = \cos(\cos(\cos(x - 1/2) + 1/2) - 1/2)$ とする。Δ の値を “急激” に小さくしながら、精度の変化をプロットせよ。精度が良くなった後、突然悪くなるのを観測し考察せよ[1]。

b)

$$f'(x) = \frac{f(x+\Delta) - f(x-\Delta)}{2\Delta} \tag{4.12}$$

この式を中心差分[2]と呼ぶ。この式を用いて前問と同じように $\cos(\cos(\cos(x - 1/2) + 1/2) - 1/2)$ の導関数の値を求め、前進差分による値と比較してみよう。精度のふるまいが異なるのはなぜだろうか。

[1] データを書き出す部分では、printf(”%g %g \n”,x,y); のように、%f のかわりに%g を使って、科学的表記を使うと良い
[2] 正確には中心差分型公式の 1 つ。特にこの式を指して「3 点公式」と呼ぶ。

参考文献

- http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/server/longdouble.html long double について。
- http://www.math.meiji.ac.jp/~mk/labo/text/ieee_format.pdf
  浮動小数点数規格IEEE754についての日本語文献。doubleにしぼった記述になっている。

第5章　微分方程式の数値解法

微分方程式の解を数値計算により求める。この節で取り扱うのは、

$$\frac{dx}{dt} = f(t, x) \tag{5.1}$$

のような常微分方程式を与えられた初期条件の下に数値的に解く手法である。

常微分方程式は、古典力学の運動方程式 $md^2x/dt^2 = F(t, x)$ や RC 回路中の電流 $dI/dt = ...$ のように、多くの物理現象を記述する。ここで取り扱うのは、ある時間 $t = t_i$ での変位 (あるいは電流など) の大きさ $x(t_i) = x_i$ が与えられた時に、その時間発展を常微分方程式に従って計算する方法である。

5.1　Euler 法

微分方程式の数値計算の内、最もシンプルで高速なのが Euler (オイラー) 法である。Euler 法では式 5.1 を以下のようにして時間発展させる。現在の時間と位置 (ここでは便宜上 x は位置を表す事にする) を (t_i, x_i) として、時間差 h 後の時間と位置 $(t_f = t_i + h, x_f)$ を

$$x_f = x_i + h \times f(t_i, x_i) \tag{5.2}$$

とする事である。これは則ち、変位の時間発展が短い区間 $[t_i, t_f]$ で一次で単調増加するとして、現在の微係数 $f(t_i, x_i)$ のみから時間発展させた事に相当する。図で書くと、図 5.1 のようになる。Euler 法を用いて計算した解の誤差の程度は h^2 となる

それでは、Euler 法を用いた例を見ていこう。式 5.1 で、$f(t, x) = 1 - x^2$ の場合を例にしてみよう。この場合の式 5.1 の解析解は、$\tanh(t)$ となる事が知られている。

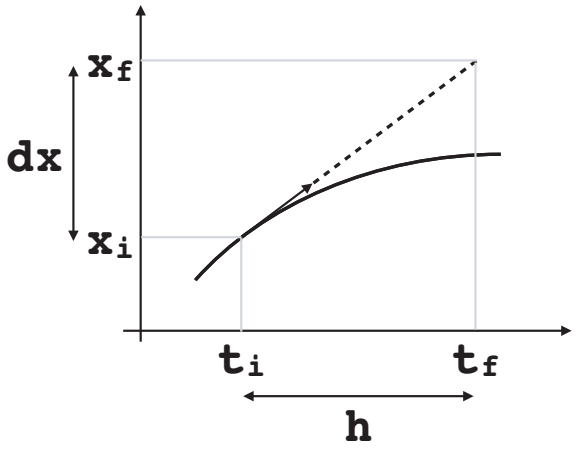


図 5.1: Euler 法の概念図。時間発展 h の間の変位の増分 $dx = h \times f(t_i, x_i)$ を現在の変位 x_i に加算することで、時間発展の結果の変位 x_f を得る。

サンプルコード 5.1: Euler.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  static double h;
5  static double xf;
6
7  double dxdt(double t, double x)
8  {
9    double bibun;
10   bibun = (1.0 - x*x);
11   return(bibun);
12 }
13
14 int ProceedOneStep(double t, double xi)
15 {
16   double dx;
17
18   dx = h * dxdt(t,xi);
19   xf = xi + dx;
20   return(0);
21 }
22
23 double Ana(double t)
24 {
25   double x;
26   x = tanh(t);
27   return(x);
28 }
```

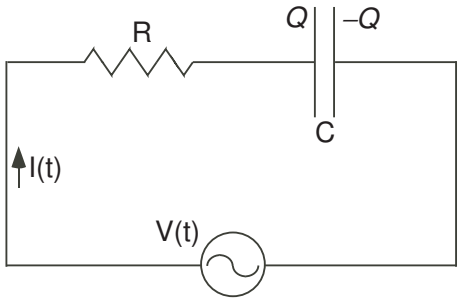
```
29
30 int main()
31 {
32   double t;
33   double xi;
34
35   h=1.0E-1;
36   xi=0.0;
37   for(t=0.0;t<5.0;t=t+h){
38     printf("%f␣%f␣%f\n",t,xi,Ana(t));
39     ProceedOneStep(t,xi);
40     xi = xf;
41   }
42 }
```

上記のサンプルコード 5.1 では、`int main()` を含め、４つの関数が使われている。それぞれは短いが、`double dxdt()` は微係数を与える関数、`double ProceedOneStep()` は、Euler 法に従って変位をステップサイズ分時間発展させ、結果を `xf` に代入する関数、`double Ana()` は解析解を返す関数である。計算結果は大域変数の `xf` に代入され、`main()` 関数の 41 行目で、次の計算の初期値を更新する。

肝となるのは `double ProceedOneStep()` である。中身は、変位の増分 `dx` を計算し現在の変位と加算することで時間発展を計算する。20 行目で結果を大域変数 `xf` に代入しているのが分かるだろう。

実行結果は、次の修正 Euler 法と同時に示す。

課題 8



図のような RC 回路を考える。ここで系に流れる電流の微分方程式は次のようになる。

$$\begin{aligned} V(t) &= Q/C + RI \\ dV/dt &= I/C + RdI/dt \end{aligned}$$

$V(t)$ として、$\sin \omega t$ を印加した場合、上式は、

$$d(\sin \omega t)/dt = \omega \cos \omega t = I/C + RdI/dt$$

となる。係数が 1 になるように取ると、

$$dI/dt = \cos t - I$$

に帰着する。
この微分方程式を $\mathrm{I}(0) = 0$ の条件で、Euler 法を用いて解くプログラムを作成せよ。ステップサイズを変えながら計算せよ。

5.2 修正 Euler 法

修正 Euler 法は、Euler 法に対して離散近似によるステップの大きさがもたらす誤差を改善する一つ目の手法である。修正 Euler 法では、時間発展を以下のように求める。

$$
\begin{aligned}
dx_1 &= h \times f(t_i, x_i) \\
dx_2 &= h \times f(t_i + h, x_i + dx_1) \\
x_f &= x_i + \frac{1}{2}(dx_1 + dx_2)
\end{aligned}
$$

$x_1 \equiv x_i + dx_1$ は Euler 法で求めた時間発展と等しい。修正 Euler 法の特徴は、現時点での微係数と Euler 法で求めた到達点 $(t_i + h, x_i + dx_1)$ での微係数の平均を使って時間発展させる点である。これを図示すると、図 5.2 のようになる。

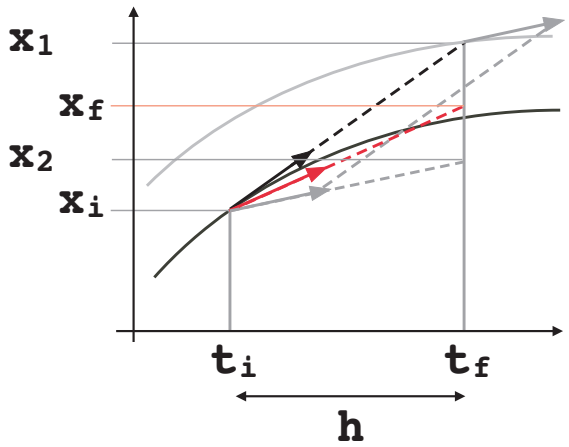


図 5.2: 修正 Euler 法の概念図。x_1 は、時間 t_i での微係数から求めた時間発展。x_2 は、x_1 での微係数から求めた時間発展。これらの平均 (赤色) を採用する。

さて、それでは Euler 法と修正 Euler 法を比較してみよう。図 5.3 が、計算結果とその比較を表す。一目瞭然なように、Euler 法は微係数の変化に追従できず、曲率の大きな部分で誤差が大きくなっている。

課題 9
課題 8 で作成したプログラムを修正 Euler 法を用いて解くものに変更せよ。余裕が有れば、解析解との比較を行え。

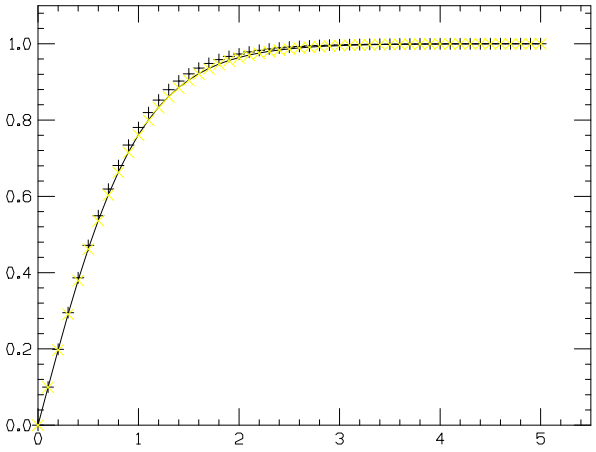


図 5.3: Euler 法と修正 Euler 法の計算結果。白線が解析解。白十字が Euler 法、黄色が修正 Euler 法。

5.3 Runge-Kutta 法

常微分方程式の数値解法で最も一般的に使われているのが (4次の)Runge-Kutta 法だろう。さらに高次の手法も存在するが[1]通常の使用では十分実用に耐える。

Runge-Kutta 法では時間発展は次式で記述される。

$$x_f = x_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{5.3}$$

但し、

$$\begin{aligned} k_1 &= h \times f(t_i, x_i) \\ k_2 &= h \times f(t_i + \frac{h}{2}, x_i + \frac{k_1}{2}) \\ k_3 &= h \times f(t_i + \frac{h}{2}, x_i + \frac{k_2}{2}) \\ k_4 &= h \times f(t_i + h, x_i + k_3) \end{aligned}$$

である。Runge-Kutta 法を用いて計算した解の誤差の程度は h^5 となる。

図 5.3 では、修正 Euler 法の適用で十分な精度が出ているように思えるが、本当であろうか。このような問題を視覚化する場合に注意すべきな

[1] Runge-Kutta-Fehlberg で検索すると良い。

のは、視覚化することで、“一見良さそう” に見える場合もある事である。常に適切な手段で視覚化する努力が必要だろう。

さて、手法の適用が適切であるかを判断するのに適した視覚化の方法は、解析解との差分をプロットする事である。図 5.4 に結果を示す。今回も一目瞭然であるように、修正 Euler 法では、曲率の大きい部分で完全には追従していない。Runge-Kutta 法では全域に亘って解析解と同等の精度を与える。

今回は、一階常微分方程式の解法を例示したが、二階以上の場合も、例えば $dx/dt = v$ のような変数変換を用いて、連立微分方程式の解法へと帰着する事が出来、同様の手法が適用可能である。実際の例は、付録に示した。こうして、Runge-Kutta 法を用いる事で複雑な微分方程式を数値的に解決する事が可能となるのである。

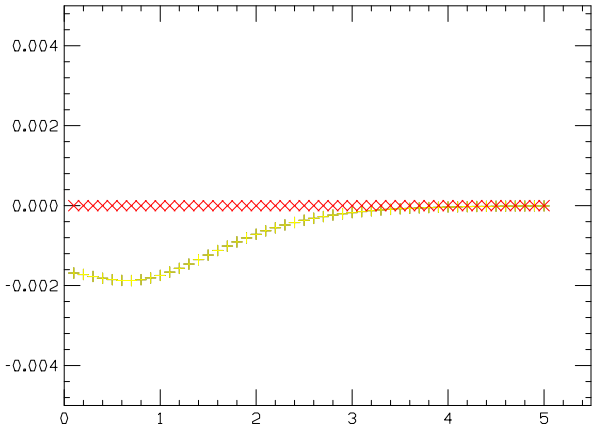


図 5.4: 修正 Euler 法と Runge-Kutta 法の計算結果の解析解からの差を解析解で割ったもの。黄色が修正 Euler 法、赤色が Runge-Kutta 法

5.4 二階微分方程式

多くの物理現象は、二階以上の微分方程式で記述される。例えば以下のような物理現象を考えよう。

バネ定数 k [kg/s^2] のバネにつながった質量 m [kg] の質点が、鉛直上方を正とする座標上で、重力加速度 $-g$ [m/s^2] と速さに比例する大きさの摩擦力を運動方向と逆向きに受けているとする。

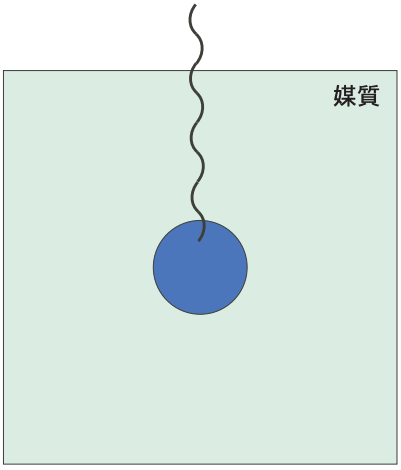


図 5.5: 粘性のある媒質中で運動する質量 m [kg] の振動子を想定する。

摩擦の比例常数は、f [kg/s] とすると、この質点の運動方程式は

$$m\frac{d^2x}{dt^2} = -mg - kx - f\frac{dx}{dt} \tag{5.4}$$

と表され、この常微分方程式は直ちに、

$$\frac{dx}{dt} = v \tag{5.5}$$

$$m\frac{dv}{dt} = -mg - kx - fv \tag{5.6}$$

という連立常微分方程式に変換される。

このような問題を、Euler 法、Runge-Kutta 法で取り扱う場合のポイントは、計算順序に気をつける事である。例えば、先に取り扱った修正 Euler 法で図 5.3 の x_i から x_f を計算する際、x と v の微分係数がそれぞれ $f_x(t,x,v)$ と $f_v(t,x,v)$ のように t、x と v の三変数関数で記述される場合には、次の順序になる。

1. x_i と v_i を用いて $dx_1 = f_x(t_i, x_i, v_i) \times h$ と $dv_1 = f_v(t_i, x_i, v_i) \times h$ を求める

2. dx_1 と dv_1 を用いて $dx_2 = f_x(t_i + h, x_i + dx_1, v_i + dv_1) \times h$ と $dv_2 = f_v(t_i + h, x_i + dx_1, v_i + dv_1) \times h$ を求める。

3. dx_1 と dx_2 の平均から x の増分 dx、v_1 と v_2 の平均から v の増分 dv を求め現在の値を更新する。

実際のプログラミングにおいては、位置 x, 速度 v のような複数の変数を配列として一般化して取り扱ってもよい。例えば、初期状態を要素数２の配列 u1[2]、h 秒後の状態を u2[2] のようにすればシンプルに記述できる。

課題 10
上記の質点の運動を、修正 Euler 法と Runge-Kutta 法により求め、運動の時間発展を求めよ。質量 m、ばね定数 k、摩擦係数 f は大域変数で与えること。初期値を含むパラメータの値を変えて解の変化を観察せよ。

課題 11

1. 二階以上の常微分方程式を用いて、何らかの物理現象を記述せよ。物理現象として期待される振る舞いはどのようなものか。もし、解析解があれば示せ。

2. 1. の常微分方程式をまず、 (i) Euler 法か又は修正 Euler 法で解いた後、(ii) Runge-Kutta 法か又は Symplectic 法[1] で解くことにより、後者の優位性を示せ。刻み幅を変えた場合の精度の変化を示し、考察せよ。

[1]本講義では取り扱わないが、ハミルトン力学系でエネルギーを保存する場合、現状で最も強力な時間発展を与える手法として知られる。興味が有れば調べて欲しい。

TIPS
授業では取り扱わないが、微分方程式の解法例でより一般性を増したプログラムを記述する場合、

```
17 int ProceedOneStepbyModEuler
18         (double t, double u1[RANK], double u2[RANK],
19          double (*dudt[RANK])(double [RANK], double))
20 {
21   int i;
22   double s1[RANK],s2[RANK];
23   double h;
24
25   h = Step;
26   for(i=0;i<RANK;i++)s1[i]= u1[i] + h*dudt[i](u1,t);
27   for(i=0;i<RANK;i++)s2[i]= u1[i] + h*dudt[i](s1,t+h);
28   for(i=0;i<RANK;i++)u2[i]= (s1[i] + s2[i])/2.0;
29   return(0);
30 }
```

のようにして 関数へのポインタの配列を使うと良いだろう。この場合 19 行目の表記は以下のように読み取る。dudt は 倍精度実数を返す “関数へのポインタ” の配列で、関数の引数は、一次元倍精度実数配列と倍精度実数である。メイン関数側では、

```
23 int main()
24 {
25   double t;
26   double (*func[RANK])(double[],double)={dxdt,dvdt};
27   double u1[RANK], u2[RANK];
...
34    ProceedOneStepbyModEuler(t, u1, u2, func);
```

のようにして、それぞれの変数の微分方程式を実装した関数を指定してやれば良い。このサンプルコードは、$SAMPLE/extra/Pointer-to-Function に配置した。

第6章　量子力学の数値計算例

物理学の問題を数値的にあるいは解析的に記述して行くと行列の固有値問題に帰着する場合が多い。つまり、状態を表す固有ベクトルや観測値である固有値を、状態の従う法則を表す行列の元に求める問題である。

本章では、物理学科の学生には馴染み深い Schrödinger 方程式の数値解法を通して量子力学の数値計算を行列により取り扱う方法を学ぶ。

6.1　Schrödinger 方程式

ポテンシャル $V(\vec{r})$ 中での、質量 m の粒子の運動は非相対論の場合、位置 $\vec{r}=(x,y,z)$、時刻 t における波動関数 $\psi(\vec{r},t)$ として記述され、ハミルトニアン H を用いて波動方程式

$$
\begin{aligned}
i\hbar\frac{\partial\psi(\vec{r},t)}{\partial t} &= H\psi(\vec{r},t) && (6.1)\\
&= \left(-\frac{\hbar^2}{2m}\nabla^2+V(\vec{r})\right)\psi(\vec{r},t) && (6.2)
\end{aligned}
$$

に従う。ハミルトニアンが時間に依存しない場合、$\psi(\vec{r},t)=\phi(\vec{r})\exp(-\frac{i}{\hbar}Et)$ のように変数分離され、

$$H\phi(\vec{r})=E\phi(\vec{r}) \tag{6.3}$$

のようにして時間に依存しない Schrödinger 方程式に帰着する。

以下では、一次元井戸型ポテンシャル中での定常解を数値計算で求める。式 6.3 を、一次元で書き下すと

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}+V(x)\right)\phi(x)=E\phi(x) \tag{6.4}$$

となる。$x=x_{\min}$ と $x=x_{\max}$ に無限に高いポテンシャルの壁があり、$x_{\min}\sim x_{\max}$ の区間では $V(x)=0$ である場合を考えよう。この条件は、

境界条件 $\phi(x_{\min}) = 0$　かつ $\phi(x_{\max}) = 0$　を意味するので、

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\right)\phi(x) = E\phi(x) \tag{6.5}$$

を、これらの境界条件のもとに解けば良い。

簡単のために、$m = \hbar^2/2$ の場合を考え、係数を 1 にしてしまおう[1]。

$$\left(-\frac{d^2}{dx^2}\right)\phi(x) = E\phi(x) \tag{6.8}$$

$x = [x_{\min}, x_{\max}]$ の区間を $x_0 = x_{\min}, x_1, x_2, ..., x_N = x_{\max}$ のように 長さ Δ の区間に N 等分し、$\phi'(x) = (\phi(x_i) - \phi(x_{i-1}))/\Delta$ のような一次の近似を二階の導関数まで適用すると、微分方程式は連立方程式: $i = 1...(N-1)$ に対して

$$\frac{-\phi(x_{i+1}) + 2\phi(x_i) - \phi(x_{i-1})}{\Delta^2} = E\phi(x_i) \tag{6.9}$$

へと変形できる。この連立差分方程式は、$\phi_0 = 0, \phi_N = 0$ の条件の下で

$$A = \begin{pmatrix} \frac{2}{\Delta^2} & -\frac{1}{\Delta^2} & 0 & 0 & ... & 0 \\ -\frac{1}{\Delta^2} & \frac{2}{\Delta^2} & -\frac{1}{\Delta^2} & 0 & ... & 0 \\ 0 & -\frac{1}{\Delta^2} & \frac{2}{\Delta^2} & -\frac{1}{\Delta^2} & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & -\frac{1}{\Delta^2} & \frac{2}{\Delta^2} & -\frac{1}{\Delta^2} \\ 0 & ... & 0 & 0 & -\frac{1}{\Delta^2} & \frac{2}{\Delta^2} \end{pmatrix} \tag{6.10}$$

と対角要素が $\frac{2}{\Delta^2}$、対角要素の左右が $-\frac{1}{\Delta^2}$ の行列と、ベクトル

$$\vec{\phi} = \begin{pmatrix} \phi(x_1) \\ \phi(x_2) \\ \phi(x_3) \\ ... \\ \phi(x_{N-2}) \\ \phi(x_{N-1}) \end{pmatrix} \tag{6.11}$$

[1]因みにこの方程式は、容易に解析的に解けて、境界条件内の区間を [0,1] とすると

$$\phi_n(x) = \sqrt{2}\sin n\pi x \tag{6.6}$$
$$E_n = (n\pi)^2 \tag{6.7}$$

となる。

定数 E を用いて、固有方程式

$$A\vec{\phi} = E\vec{\phi} \tag{6.12}$$

の形で表されるので、問題はこの行列の固有値問題として解くことができる。区間が [0,1] の場合の波動関数を計算すると、図 6.1 のようになる。

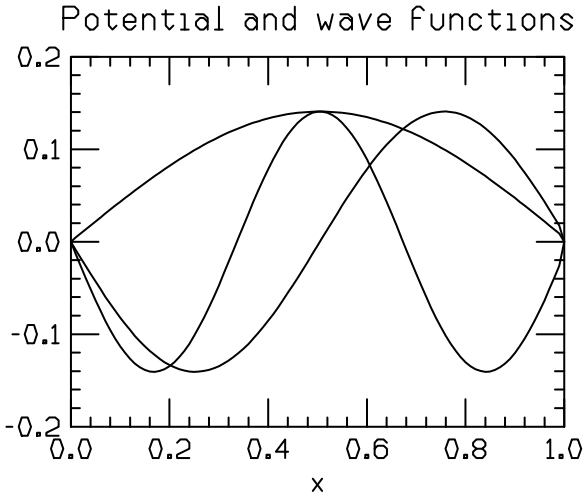


図 6.1: 波動関数の解法例 (規格化はしていない)。

6.2 LAPACK 数値計算ライブラリ

前節で登場した行列 6.10 は、実三重対角行列[2]と呼ばれる種類に分類される。その特徴は、0 以外の要素が少ない疎な行列である事で、対角成分とその隣にしか値をもたない。このような行列の固有値問題は大変ポピュラーな問題である。また、特に要素が実数で ${}^{T}A = A$ の場合は実対称三重対角行列と呼ばれる。今回は、この実対称三重対角行列の固有値問題を先人が残してくれた智恵の結晶ともいうべきライブラリを用いて解決する方法を、以下の課題を例として扱いながら学んでみよう。

[2]tridiagonal matrix で検索すると良い。

課題 12
実対称三重対角行列

$$\begin{pmatrix} 1 & 8 & 0 & 0 & 0 \\ 8 & 3 & 6 & 0 & 0 \\ 0 & 6 & 5 & 4 & 0 \\ 0 & 0 & 4 & 7 & 2 \\ 0 & 0 & 0 & 2 & 9 \end{pmatrix}$$

の固有値問題を LAPACK ライブラリを用いて解き、固有値と固有ベクトルを求めよ。

用いる LAPACK 代数ライブラリは極めて多種の機能を提供するが、調べてみると我々が興味のある対称三重対角行列の固有値問題に適している関数名は、DSTEV であることが分かる[3]。LAPACK のライブラリは FORTRAN で記述されているが、付録 B.7 に記述した幾つかの “決まり事” さえ守れば、ほとんど C 言語の関数を呼ぶ場合と同様に取り扱うことができる。

まず、DSTEV の仕様[4]を参照してみると、

```
      SUBROUTINE DSTEV( JOBZ, N, D, E, Z, LDZ, WORK, INFO )
*
*  Purpose
*  =======
*
*  DSTEV computes all eigenvalues and, optionally, eigenvectors of a
*  real symmetric tridiagonal matrix A.
```

と書いてある。SUBROUTINE は、FORTRAN の言葉で、関数の一種を表し、DSTEV の引数は、JOBZ 以下、8 個である事が分かる。引数の型や意味を調べると表 6.1 の通りである。

[3] 対称三重対角行列 固有値 lapack で検索してみると良い。
[4] http://www.netlib.org/lapack/double/dstev.f

引数名	型	入力時の意味	出力時の意味
JOBZ	char	固有ベクトルを計算する場合は 'V' しない場合は 'N'	—
N	int	行列の次数	—
D	double [N]	行列の対角成分	固有値を昇順に格納
E	double [N-1]	対角要素の隣の成分	—
Z	double [LDZ*N]	—	固有ベクトルを固有値の順に格納
LDZ	int	N と等しくしておけば良い	—
WORK	double [2*N-2]	作業領域メモリ	—
INFO	int	—	0 なら成功, 非 0 ならエラー

表 6.1: 実対称三重対角行列の固有値と固有ベクトルを求める LAPACK の `DSTEV( JOBZ, N, D, E, Z, LDZ, WORK, INFO )` の引数

課題を行う手順としては次のようになる。まず、表 6.1 に従い変数を適切に宣言する。次に、変数に課題に応じた値を代入する（関数を作成して代入するのが推奨される）。引数のうち、文字型である JOBZ は char JOBZ; として宣言し、JOBZ = 'V'; のようにして文字定数 V を代入して使えばよい。

大まかな流れをまとめると、以下の抜粋のようになる。

```
double D[N], E[N-1];
char JOBZ;
...
JOBZ = 'V';
...
SetMatrix(D,E); // 配列 D と E に値を設定する関数 SetMatrix()
...
dstev_(&JOBZ, &n, D, E, Z, &ldz, WORK, &INFO, 1);
...
```

このようにして、LAPACK ライブラリを使って固有値を解く。最終的

に得られた固有値と固有ベクトルを、必要に応じて規格化し、画面に書き出すなどすれば良い。

LAPACK ライブラリを TSUBAME2.0 上で使う時、例えばソースプログラムの名前を eigen.c とするとコンパイルするには、以下のようなコマンドを用いる。

```
gcc -c eigen.c
gcc -o eigen eigen.o -L$EDULIB -llapack -lblas -lg2c -lm
```

若干おまじないの感も漂うが、こうする事で LAPACK 数値計算ライブラリとリンク（結合）する事ができる。

毎回この長い呪文を唱えるのは面倒なので、Makefile を以下のように記述すれば良い。gcc の前の空白はタブ (tab キーで入力する) である必要があるので注意して欲しい。Makefile を用意すれば、`tsubame% make` と打つだけで自動でファイルの依存関係を解決し実行ファイルを生成してくれる。

サンプルコード 6.1: Makefile

```
1 eigen: eigen.o
2     gcc −o eigen eigen.o −L$(EDULIB) −llapack −lblas −lg2c −lm
3 eigen.o: eigen.c
4     gcc −c eigen.c
```

課題 13

式 6.12 で表される固有値問題の行列解法を実装したプログラムを作成し、

1) 調和振動子ポテンシャルの場合について、固有値問題を解き考察せよ[1]。その際、ポテンシャルは、位置の関数としてプログラムする事。基底状態から第 2 励起状態までの固有値を求め解析解と比較し、次いで波動関数を図示せよ。固有値の誤差は、どの程度か？

2) 自ら決めたポテンシャルについて、固有値問題を解き考察せよ。

※波動関数は、必ずしも規格化する必要はないが、されているほうが望ましい。

[1] Runge-Kutta 法を用いると波動関数の時間発展も考察可能である。

課題13を解く際のポイントは何を関数にするかを良く考える事である。お薦めは、波動関数を計算する”節”に相当する x_i の数列を定義する部分と、位置に依るポテンシャルを記述する部分を関数とする事である。前者は、i 番目の節の位置を与える関数であり、後者は与えられた位置でのポテンシャルを与える関数である。これらを用いて、式 6.12 に相当する行列を、LAPACK の入力に即した形式で記述する配列を定義して計算を行えば良い。

TIPS

MacOS (Mojave では確認済み) には、デフォルトで LAPACK がインストールされているので以下のような Makefile を用意すれば手元の iMac で、TSUBAME2.0 と共通の Makefile を使って eigen をコンパイルできる。

サンプルコード 6.2: Makefile

```
1  CC=gcc
2
3  ifeq ($(shell uname),Darwin)
4          LDOPT=-Wl,-framework,Accelerate
5          LIBS =
6  else
7          LIBS = -L$(EDULIB) -llapack -lblas -lg2c -lm
8  endif
9
10 eigen: eigen.o
11         $(CC) $(LDOPT) -o eigen eigen.o $(LIBS)
12
13 .SUFFIXES: .c .o
14
15 .c.o:
16         $(CC) -c $<
17
18 clean:
19         rm -f *.o eigen *~
```

第7章　モンテカルロシミュレーション I

乱数を用いて、統計的な分布を伴う物理現象を模擬的に取り扱う方法はモンテカルロシミュレーション[1]と呼ばれ、計算機上で仮想的に実験を構築する手法として非常に強力な手段である。モンテカルロシミュレーションでは、知られている物理法則に則(のっと)った事象の統計的な出現確率に応じて、乱数で事象を選択するので、未知の物理法則を取り入れる事が出来る訳ではない。ここでは、モンテカルロシミュレーションの基礎となる乱数を使ったモンテカルロ求積法を中心に学ぶ。

7.1　モンテカルロ求積法

モンテカルロシミュレーションは、その手法的特徴からして乱数と切り離して考える事は出来ない。より “良い” 乱数の発生方法自体、研究の対象となる程奥の深いものがあり、現代技術の基礎をなすものである。普通 UNIX システム標準の乱数は、乱数の特性という意味で高度な数値計算には全く使い物にならない[2]。

そこでここでは、高速の擬似乱数発生法Mersenne Twister(メルセンヌ・ツイスター)[3] を採用する。後の便利のためにまず、区間 [0,1] に一様に分布する実数を生成するプログラムを用意しよう。まず作業ディレクトリを作成し、$SAMPLE/chapter7/MT 以下にある `mt.c` と `mtwister.h` をコピーせよ[4]。

```
tsubame% cp $SAMPLE/chapter7/MT/mt.c .
tsubame% cp $SAMPLE/chapter7/MT/mtwister.h .
```

とする。

---

[1] 名前のモンテカルロは高級賭博で有名なモナコ公国の地区名に由来する。

[2] TSUBAME 2.0 のシステム標準乱数は、さほど悪くなさそうだが。

[3] `http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/mt.html`

[4] 例によって%はプロンプトを表すので入力しない。

サンプルコード 7.1: mt.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include "mtwister.h"
4
5  double drnd()
6  {
7
8    return(genrand_real2());
9  }
10
11 int main()
12 {
13   int i,j;
14   double r;
15
16   init_genrand(2525);
17   for(i=0;i<100;i++){
18     r = drnd();
19     printf("%f\n",r);
20   }
21 }
```

サンプルコード7.1の中身は至って簡単なものである。16 行目、init_genrand() では、乱数発生装置に初期化のための適当擬似乱数の種[5]を与える。5–9 行目は、倍精度実数で区間 [0,1) に分布する乱数を返す関数 drnd() を定義している。この関数は殆ど空っぽの関数であるが、乱数発生法を変更する可能性をみこし、クッションとしてユーザー定義関数を挟んで置くことが推奨される。このプログラムをコンパイルするには、以下のように入力する[6]。

```
tsubame% gcc -c mt.c
tsubame% gcc -o mt mt.o -L$EDULIB -lmtwister -lm
tsubame% ./mt
```

この場合の、Makefile の記述は以下のようになる。<tab> の部分は、タブキーにより入力する必要がある[7]

[5]種は正整数であれば何でも良い。

[6]TIPS　を参照して、Makefile を準備し、make ユーティリティを使用せよ。

[7]Emacs 以外の環境では改行コードに LF: Line Feed が要求されている事に注意せよ。

サンプルコード 7.2: Makefie

```
1 mt: mt.o
2 <tab> gcc −o mt mt.o −L$(EDULIB) −lmtwister −lm
3 mt.o: mt.c
4 <tab> gcc −c mt.c
```

モンテカルロシミュレーションを用いた最も簡単な例は、モンテカルロ求積法という閉空間内の体積を求める方法である。

たとえば、区間 [0,1) に一様に分布する乱数を、N コずつ組にする事で、N 次元空間上で、領域 [0,1:0,1...) に点を一様に分布させる事が出来る。これら発生した点が、半径 1 の球の内部にある確率は球の体積に比例する。従って、上記で発生した点と球内部に入った点の数の比を求めることにより球の体積を推測する事ができる。

N = 10 次元空間のベクトルを例えば以下のように配列を用いて表現しよう。

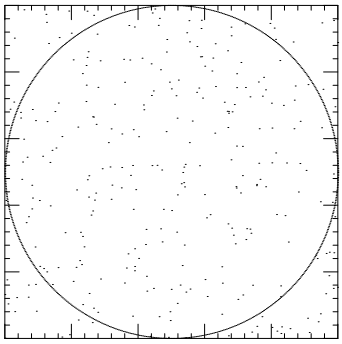
```
#define NDIM 10
...
int main()
{
  int i;
  double u[NDIM];
  for(i=0;i<NDIM;i++){
    u[i]....
  ...
  }
}
```

#define の行は定義文と呼ばれる形式の書き方であり、この場合コンパイラ[8]は、NDIM を　10 に置き換えてから処理を行う。このようにする事で、NDIM の定義を変更する事により、任意の次元に対して計算する事が出来るようになる。

[8]gcc -c で起動するコマンドの事である

課題 14

1) 与えられた次元に対して、球の体積を求めるプログラムを記述せよ。誤差はどのようになることが期待されるか。**2)** 特に二次元の場合、面積を考えると、原点からの距離が 1 以下の点は、$\pi/4$ の割合で発生するはずである。この事を利用して円周率の値を算出せよ。**3)** 2) の方法による円周率の計測を 10,000 回繰り返し、求めた円周率の計測値の分布を下記を参考にして頻度分布 (ヒストグラム) として作成せよ。分布の幅は点の数とどのような関係にあるか。



乱数を用いたシミュレーションの結果得られる頻度分布を吟味するには、ヒストグラムを生成し可視化する事が効果的である。ここでは、ヒストグラムを PGPlot ライブラリを用いて生成するサンプルプログラムを記述する。

サンプルコード 7.3: exp.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "cpgplot.h"
5  #include "mtwister.h"
6
7  #define N 10
8
9  int InitRandom(int iran)
10 {
11   void init_genrand();
12   unsigned long iseed;
13
14   iseed = iran;
15   init_genrand(iseed);
16   return(0);
17 }
18
19 double drnd()
20 {
```

```
21    double genrand_real2();
22    return(genrand_real2());
23  }
24
25  int main()
26  {
27    int ista;
28    int i;
29    float data[10000];
30
31    InitRandom(10);
32    ista = cpgopen("/xwindow");
33
34    for(i=0;i<10000;i++){
35      data[i] = -log(drnd());
36    }
37    cpghist(10000,data,0.0,10.0,100,4);
38    cpgclos();
39
40  }
```

このプログラムのポイントは、32 行目 cpgopen で、PGPlot の画面を開く点、35 行目で配列 data に乱数を保存し、37 行目で配列 data の内容を頻度分布のヒストグラムとしてプロットしている点である。cpghist の引数は順にデータの点数、データの入った配列、ヒストグラムの左端値、右端値、ビン数、オプションである。サンプルの場合、データ点数 10,000 で 100 ビンに分類したヒストグラムをプロットする。

このプログラムをコンパイルするには、

```
tsubame% cp $EDU/include/mtwister.h .
tsubame% cp $EDU/include/cpgplot.h .
```

として、mtwister.h と cpgplot.h ファイルをコピーした上で、

```
tsubame% gcc -c exp.c
tsubame% gcc -o exp exp.o -L$EDULIB -lmtwister ←
-lcpgplot -lpgplot -L/usr/lib64 -lX11 -lgfortran -lm
tsubame% ./exp
```

とする (但し、←部分は、ページ巾の関係で改行したが実際には改行しない)。もちろん対応する Makefile を用意しても良い。

サンプルコード 7.4: Makefie

```
1  exp: exp.o
2          gcc -o exp exp.o -L$(EDULIB) -lmtwister -lcpgplot ←
```

```
3     -lpgplot -L/usr/lib64 -lX11 -lgfortran -lm
4  exp.o: exp.c
5           gcc -c exp.c
```

(但し、←部分は、ページ巾の関係で改行したが実際には改行は不要である)

7.2 様々な乱数

棄却法

前節では区間 [0:1) に一様に分布する乱数を取り扱ったが、実際にシミュレーションを行う場合、一様ではなくある分布関数に従う乱数が必要となる事が多い。では任意の分布関数に従う乱数はどのようにして生成すれば良いのであろうか。一つ目の方法は棄却法と呼ばれる手法である。区間 [0:xmax) に分布する乱数 a の分布関数を $f(x) > 0$ とする。区間中の $f(x)$ の最大値を $f_{\max}$ としよう。今、xy 平面上の二次元区間 [0:xmax, 0,$f_{\max}$] を考える。この二次元区間に一様にランダムに分布する点を考え、図 7.1 のように点が $y = f(x)$ の描く曲線より下に来た場合のみ “採用” とすれば、点の x 座標の分布は $f(x)$ に従う。つまり、一様乱数により (x,y) を発生し y が f(x) より小さい場合のみ使用し、大きければもう一度 (x,y) を生成し直すという事を繰り返せばよい。

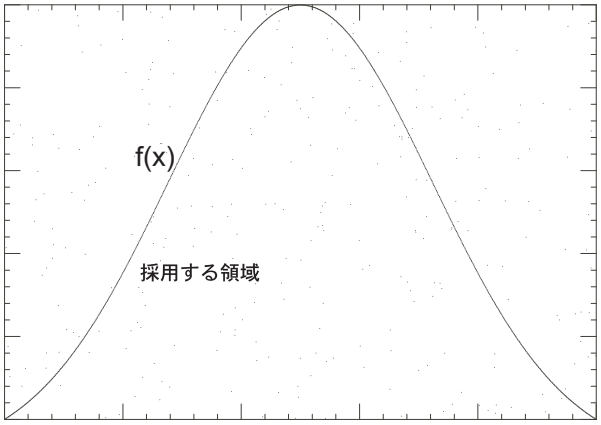


図 7.1: 棄却法の概念図

逆関数法

分布関数 f(x) に従う乱数 a を発生したとする。この乱数を用いて

$$a \to F(a) = \int_{\text{区間最小値}}^{a} f(x)dx \tag{7.1}$$

という変換を行うと、$F(a)$ は、区間 [0:1) に一様分布を形成する。逆に区間 [0:1) に一様に生成した乱数を上記の逆変換する事で分布 f(x) に従う乱数を生成可能である。この手法は全ての場合に適用可能であるわけではなく、$F(x)$ の逆関数を記述出来た場合のみしか適用できない点は注意が必要である。

例えば、粒子の崩壊事象 $f(x) = \exp(-x)$ $(x > 0)$ に従う乱数は、次のようにして発生すれば良い。まず、上式の不定積分は $F(x) = -\exp(-x)$ であるから、区間最小値を 0 とすれば、定積分は、$F(a) = -\exp(-a) + 1$ である。この逆関数は $F^{-1}(x) = -\log(-x + 1)$ である。よって、一様乱数 r を発生し $x = F^{-1}(r) = -\log(-r + 1)$ を計算すれば良い。

課題 15
上記の手法に従い、$f(x) = \exp(-x)$ $(x > 0)$ に従う乱数を発生せよ。

ガウス分布

多くの物理現象は、中心値の周りにある幅をもって分布する事が多く、ガウス型を用いて近似出来る場合が多い。ガウス型に従う乱数を発生するには次のように一様分布する乱数 r_1 と r_2 に対して Box-Müller 変換を行って生成すれば良い。

$$\begin{aligned} x_1 &= \sqrt{-2\log r_1} \times \cos(2\pi r_2) \\ x_2 &= \sqrt{-2\log r_1} \times \sin(2\pi r_2) \end{aligned}$$

上記の方法では、一度に同等の二つの乱数を発生する事が出来、どちらを採用しても等価であるが、一つずつ交互に使えば計算回数が少なくて済む。

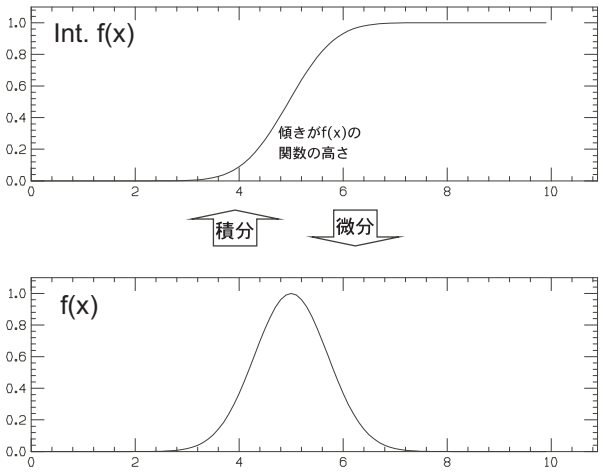


図 7.2: 逆関数法の概念図。上のパネルの関数の傾きが下の関数の高さになっている。従って、下の関数 f(x) の高さに従って乱数を発生させ、上の単調増加関数を使って変換すれば、一様乱数となるのである。この事を逆に利用するのが逆関数法である。

実際のプログラム例を `$SAMPLE/chapter7/Monte-Carlo/gaus.c, gaus.h` に示した[9]。gaus.c で記述される関数 gRand() は、中心が原点で幅 1 の正規分布に従う乱数を発生する。つまり

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp -\frac{(x-\mu)^2}{2\sigma^2} \tag{7.2}$$

の、特に $\mu = 0$ かつ $\sigma = 1$ の場合に相当する。これを任意のガウス型分布、つまり中心が μ 幅が σ の分布に変更するには、上記の gRand() 関数で発生させた乱数を σ 倍して、μ だけ足してやれば良い。

gaus.c の典型的な使い方を以下に示す。

サンプルコード 7.5: gaus.c

```
1 #include <stdio.h>
2 #include "gaus.h"
3
4 int main()
```

[9] コンパイルには、rand.c と rand.h が必要となる点に注意せよ。

```
5  {
6    int i;
7    double sigma,mu;
8    iniRand(1139);
9    mu = 1.0;
10   sigma = 3.0;
11   for (i=0;i<10000;i++){
12     double x;
13
14     x = gRand()*sigma + mu;
15     printf("%f\n",x );
16   }
17 }
```

乱数 x は 11 行目のループで 10000 回発生しているが、x は 幅 sigma 中心値 mu の正規分布に従う。14 行目で上記の方法で 幅 sigma 中心値 mu の正規分布に変換している。

TIPS
ライブラリを使用したり、プログラムを分割して、一つの実行形式を生成するために必要なファイル数が増えてくると、コンパイルのコマンド数が増えて煩雑さが増す。UNIX には標準で、ファイルの依存関係を一手に取り扱ってくれる便利なツール make が導入されている。make は、Makefile という名前のファイルに、実行形式を生成するのに必要な依存関係を列挙しておく事で、ほぼ自動的に更新したファイルのコンパイルを行う。
Makefile の中身は、例えば前記の例の場合

```
1  Integrate: Integrate.o Diffeq.o
2  <tab>     gcc -o Integrate Integrate.o Diffeq.o
3
4  Integrate.o: Integrate.c
5  <tab>     gcc -c Integrate.c
6
7  Diffeq.o: Diffeq.c
8  <tab>     gcc -c Diffeq.c
```

のように記述する (<tab> はタブキーの入力を意味するので注意して欲しい)。各行の意味はほぼ自明だが、一行目 Integrate: Integrate.o Diffeq.o は Integrate を生成するには、Integrate.o と Diffeq.o が必要という意味。二行目は、実際に Integrate を生成するには、`gcc -o Integrate Integrate.o Diffeq.o` というコマンドで生成するという意味である。
コマンドラインから

```
tsubame% make
```

とする事で、必要な依存関係に基づき更新したファイルのみコンパイルして、自動で実行形式 Integrate が生成される。Makefile には、より簡略化された記述を用いたり、逆に複雑な条件分岐を反映させたりする事も出来る。

第8章　モンテカルロシミュレーションII

7 章では、乱数を用いた積分法モンテカルロ求積法を取り扱った。本章[1]では、統計力学で特異な相転移現象を動的モンテカルロを使ってシミュレートしてみよう[2]。

8.1　イジング模型

物理学の研究において、現象とその背後にある機構を理解するためのアプローチの一つとして有効なのが、機構を比較的単純なモデルを用いて記述し、現象を再現する試みである。これによって、現象のうち再現できる部分と出来ない部分が明確になり、より深い理解に到達する事が出来る。

磁性体の電子スピンによる物性を、磁性をもつ物質を等間隔のスピンを用いて表現したモデルにイジング模型がある。ここでは主に二次元のイジング模型を取り上げる。イジング模型のハミルトニアンは

$$H = -\sum_{\langle k,l \rangle} J s_k s_l$$

と書ける。ここで和の $\langle k, l \rangle$ は最近接スピン間でとり、スピンは上か下かの二値 $s = \pm 1$ をとるものとする。定数である相互作用 J が 1 の場合は強磁性イジング模型と呼ばれる。

統計力学の処方箋に従えば、ある状態 $\boldsymbol{S}_i$ の温度 T における実現確率はボルツマン定数 k_b を用いて、

$$P\left(\boldsymbol{S}_i\right) = \frac{\exp\left(-E_i/k_b T\right)}{Z}$$

[1] この章の記述は西森研・松田佳希の協力による。

[2] 参考文献：福島孝治氏「モンテカルロ法の最前線」
http://www.smapip.is.tohoku.ac.jp/šmapip/2003/tutorial/textbook/koji-hukushima.pdf

となることが知られている。ここで E_i は状態 $\boldsymbol{S}_i$ のエネルギーであり、Z は分配関数

$$Z = \sum_i \exp\left(-E_i/k_bT\right)$$

である。この時任意の物理量 A の期待値 (熱平均) は、

$$\langle A \rangle = \sum_i \frac{A_i \exp\left(-E_i/k_bT\right)}{Z} \tag{8.1}$$

として計算される。

原理的には、すべての取り得る状態について和をとることで上式を計算してやれば、物理量の期待値を求めることができる。ところが一般に、多粒子系・熱力学的極限においてこの和を計算することは大変困難である。例えば 1 つのスピンが上か下か (二値) を取るイジング模型を考えよう。熱力学的極限の振る舞いを予想するために、スピン数を N として計算を行おうとすると、トレースは 2^N の和になり系のサイズとともに計算量は指数関数的に増加する。これは大変深刻な問題である。09 年 12 月現在世界最速のスーパーコンピュータであるアメリカの Jaguar は 1 秒あたり約 2.0×10^{15} 程度の計算 (2P flops) ができるが、1 回の演算で 1 回和をとれるとしてこれを 1 年占有しても、

$$2.0 \times 10^{15} \times 3600 \times 24 \times 365 = 6.31 \times 10^{22} = 2^{75.7}$$

という数の計算ができるに過ぎない[3]。つまりたった約 75 スピン程度の熱力学的性質しか解明できないのである。この困難を克服するための一つの方法として、この章で説明するマルコフ連鎖モンテカルロ法が考案され一定の成功を収めている。

熱平均の式 (8.1) を積分形式で書くと、

$$\int A\left(\boldsymbol{S}\right) P\left(\boldsymbol{S}\right) d\boldsymbol{S} \tag{8.2}$$

となる。つまり物理量の期待値を求めるということは、積分を計算していることに他ならない。以下では、物理シミュレーションとしてのモンテカルロ法の基本的なアイデアを理解し、前節で学習したモンテカルロ積分を使った物理量の計測を行う。

[3]当然スーパーコンピュータを 1 年間占有するのは現実的には不可能である。普通のコンピュータの場合は 30 スピン程度で計算待ち時間の限界となる。

8.2 メトロポリス法

強磁性イジング模型の 1 スピンあたりの磁化[4]は

$$m = \frac{1}{N}\left\langle \sum_{k=1}^{N} s_k \right\rangle = \frac{1}{N} \sum_{s_1=\pm 1} \sum_{s_2=\pm 1} \cdots \sum_{s_N=\pm 1} \frac{(\sum_{k=1}^{N} s_k) \exp\left(-H(\boldsymbol{S})/k_b T\right)}{Z}$$

で定義され、スピンがどれ位揃っているかを表す。ここで N はスピン数である。これを積分と見なし前節のようにモンテカルロ積分で物理量を求めることを考える場合、式 (8.2) でいうところの $A(\boldsymbol{S}) = \sum_{k=1}^{N} s_k/N$, $P(\boldsymbol{S}) = \exp\left(-H(\boldsymbol{S})/k_b T\right)/Z$ に対応し積分の次元は 2^N になる。磁化を測定する場合、$A(\boldsymbol{S})$ はあるスピンの組み合わせが与えられたときの S_i の平均値である。観測したい物理量により $A(\boldsymbol{S})$ が変わるだけで、例えばエネルギーであれば $A(\boldsymbol{S}) = H(\boldsymbol{S})$ となる。単純にモンテカルロ積分を考えると、N 個のスピンに対しランダムに $+1$ か -1 を割り当て、$A(\boldsymbol{S})P(\boldsymbol{S})$ の平均を取ればいいように思うが、これは大変非効率である。なぜなら $P(\boldsymbol{S})$ という関数はギブス・ボルツマン分布において殆どの組み合わせで値が小さくなるからである。それは有意な物理現象が期待される低温において $P(\boldsymbol{S})$ の T 依存性から明からである。極低温におけるほぼ全てのスピンがそろった $(+1 \text{ or } -1)$ 強磁性状態は、ランダムにスピンを決めた場合、サイズとともに指数関数的に低い確率でしか起こらない。そのために分散が非常に大きくなり、結局莫大な数のサンプルを生成しなければならなくなる。

この困難を解決したのが、今日物理シミュレーションとして利用される重点サンプリングモンテカルロ法である。この方法では積分変数をランダムに生成するのではなく、$P(\boldsymbol{S})$ の値が大きい変数を重点的に生成するのである。低温であればスピンがそろった状態が優先的に生成され、高温であれば無秩序に生成されるという具合である。それではどのようにしてサンプルを生成すればよいだろうか。もし $P(\boldsymbol{S})$ が分かっていればそれに従う乱数を生成すればよいが、多くの場合非自明である[5]。重点サンプリング法では確率過程により逐次的にスピン配置を生成する。現在の適当なスピン配置から次の時刻におけるスピン配置を確率的に決定するのである。ここで次の時刻の状態は現在の時刻の状態にのみ依存し、この

[4]磁化は同時にモデルの秩序パラメータである。

[5]分布関数 (分配関数) が分かると言うことは、解析的に解けているかもしくは全てのスピン配置について調べ上げることと同じである。

ような確率過程はマルコフ過程と呼ばれる。この時、次の時刻 $t+\Delta t$ においてある状態の実現する確率 $P_i(t+\Delta t)$ は次のマスター方程式によって記述される。

$$P_i(t+\Delta t) = P_i(t) - \sum_{j\neq i} P_i(t)\, w(i\to j) + \sum_{j\neq i} P_j(t)\, w(j\to i)$$

ここで $w(i\to j)$ は状態 i から j に遷移する確率である。右辺第 2 項は確率の流出、第 3 項は流入を表している。熱平衡状態においては左辺と右辺第 1 項が等しいと考えられるので、次の等式が成り立つ。

$$\sum_{j\neq i} P_i w(i\to j) = \sum_{j\neq i} P_j w(j\to i)$$

前述のように平衡状態において P はギブス・ボルツマン分布である。ここで十分条件として詳細釣り合い条件

$$\frac{w(i\to j)}{w(j\to i)} = \frac{P_j}{P_i} = \exp\left((E_i - E_j)/k_bT\right)$$

を選ぶことにする。ここで E_i は系の状態 i におけるエネルギー $E_i = H(\boldsymbol{S}_i)$ である。最後の問題は遷移確率 w だが、メトロポリス法ではこれを

$$w(i\to j) = \min\left(1, e^{(E_i-E_j)/k_bT}\right)$$

として逐次状態遷移を行う。

課題 16(参考)
メトロポリス法が詳細釣り合い条件を満たすをことを確かめよ。
さらに熱浴法 (ギブスサンプラー)

$$w(i\to j) = \frac{e^{-E_j/k_bT}}{e^{-E_i/k_bT} + e^{-E_j/k_bT}}$$

ではどうか[1]。

[1]簡単のためイジング模型の 1 スピン更新による式である。連続スピン系では分母は積分になる。

具体的な状態更新の方法として、1 回の試行でランダムに選んだ 1 つのスピンを反転するかを決めよう[6]。適当な状態から始めた場合、詳細釣り

[6]詳細釣り合い条件を満たす限り同時に複数のスピンを反転しても良い。

合い条件を満たすように 1 つずつスピンを更新していけば、系は平衡状態に近づく[7]。遷移確率は反転前と反転後のエネルギーの差と、系の温度 k_bT によって計算される。物理量を測定するには、熱平均は時間平均で置き換えられ、

$$\langle A \rangle = \frac{1}{M} \sum_{t=t_{eq}}^{t_{eq}+M} A(t)$$

を計算すればよい[8]。ここで t_{eq} は系が熱平衡状態に達した時刻であり、時間の定義はモンテカルロステップ (MCS) に取ることが多い。モンテカルロステップとは系全体に渡る更新の試行を 1 時刻と見なす単位である。例えばイジング模型の 1 スピン更新であれば、全てのスピンの更新と見なせる N 回の試行を 1MCS とする。

8.3 二次元イジング模型

メトロポリス法による二次元イジング模型のシミュレーションをコンピュータで実現する準備段階として、スピン状態の配列変数への格納とその可視化、スピン間相互作用と系のエネルギーの計算手順の概要について述べる。

8.3.1 準備 1 : 可視化

計算の過程を可視化する事は計算の進み方を把握し、現象を理解するために重要である。ここでは、系の状態をリアルタイムに把握するために、PGPlot というライブラリ[9]を使った可視化法を例示する。

プログラム中で PGPlot ライブラリを使う場合の最も簡単な例を見ていこう。

---

[7]全ての状態へ有限ステップで到達可能という条件 (エルゴード条件) も満たさなければならないが、ランダムに選ばれたスピン更新による状態遷移においてこれは満たされる。

[8]より正確には、各測定は相関時間より十分長い時間間隔をおくべきである。毎ステップごとに物理量を測定すると、各状態は独立と見なすことは出来ない。

[9]例えば `http://www.pol.geophys.tohoku.ac.jp/~kizu/pgplot/tutorial.htm` のようなサイトのリファレンスマニュアルが役立つ。C 言語から PGPlot を使う時の関数の名前は小文字で、FORTRAN の場合の関数名の先頭に小文字の c を付け加えたものとなる。

サンプルコード 8.1: ising.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <cpgplot.h>
5  #include <mtwister.h>
6  #define NPANEL 4
7
8  int main()
9  {
10   int i,j;
11
12   cpgopen("/xwindow"); // 描画ウインドウを開く
13   cpgscr(10, 0.7, 0.0, 0.0); // 赤色の定義:色番号10番とする
14   cpgscr(20, 0.1, 0.4, 0.1); // 緑色の定義:色番号20番とする
15   cpgenv(0, NPANEL, 0, NPANEL, 1, -2); // 座標を設定。
16
17   cpgbbuf(); // 描画のバッファリング開始
18   cpgeras(); // 描画面の消去
19   for(i=0;i<NPANEL;i++){
20     for(j=0;j<NPANEL;j++){
21       cpgsci(10); // 赤色を選択
22       cpgrect(i+0.1,i+0.9,j+0.1,j+0.9); // パネルを描画
23     }
24   }
25   cpgebuf(); // 描画を描画
26   usleep(10000); // 10000 マイクロ秒の休止
27
28   cpgclos(); // 最後に画面を消去して終了
29 }
```

各行の意味は、コメントに書いた通りであるが、おおまかな手順としては、12 から 15 行目が初期化、17 から 25 行目が描画、27 行目で終了の手順となる。各関数の役割を表 8.1 にまとめた。

このプログラムをコンパイルするには、以下のようにする。

```
tsubame% gcc -I$EDULIB -c ising.c
tsubame% gcc -o ising ising.o -L$EDULIB -lmtwister -lcpgplot -lpgplot
-L/usr/lib64 -lX11 -lgfortran -lm
```

二行目は、長いので折り返してあるが、一行で入力する。

またいつものように、Makefile を用意する事をお奨めする。Makefile を用意することで作業の効率化が図られると同時に、実行形式を作成するの方法を記録する事にもなり、時に大変役立つ事がある。矢印部分で

行	関数名	関数の役割
12	cpgopen()	図形描画のためのウインドウを開く。
13, 14	cpgscr()	使用する色に番号をつけて準備する。
15	cpgenv()	描画域に xy 座標を設定する。
17	cpgbbuf()	描画用記憶域に書き始めるよう設定する。
18	cpgeras()	描画域を消去する。
21	cpgsci()	(先に定義した) 色番号を選択する。
22	cpgrect()	長方形を描画する。
25	cpgebuf()	描画用記憶域の内容を画面に描く。
28	cpgclos()	ウインドウを閉じる。

表 8.1: PGPlot の関数群の一部

はページ巾の関係で折り返しているが、実際には 1 行で記述する。いつもの事だが、行頭のスペースはタブキーで入力する必要がある。

サンプルコード 8.2: Makefile

```
1 ising: ising.o
2 	gcc −o ising ising.o −L$(EDULIB) −lmtwister
3 → −lcpgplot −lpgplot −L/usr/lib64 −lX11 −lgfortran −lm
4 ising.o: ising.c
5 	gcc −I$(EDULIB) −c ising.c
```

→の所は、紙幅の都合で折り返しているが実際は折り返さずに記述する。例のごとく gcc の前の空白は、タブキーで入力する必要がある。

課題 17

本章で取り扱うイジング模型では、格子上のスピンが上下の二値を持つ。そこで、上下を ± 1 で表現する事とし、二次元格子上のスピンを二次元配列[1]で取り扱う事にしよう。

まず最初にスピンの持つ値は、乱数で生成する事とする。乱数については、7 章で最初に取り扱った。区間 [0:1) の一様乱数を発生し、[0:0.5) の場合下向き、[0.5:1.0) の場合上向きとする事にしよう。

サンプルコードから出発して、4×4 の二次元配列の内容に応じたカラーパネルを描画する関数を書いてみよう。プログラムでは、スピンの向きを ± 1 の値として二次元配列に保持し、スピンの向きに応じ、上向きの場合赤色で、下向きの場合緑色でパネルを表示する。うまく準備出来ていれば、表示されたパネルは赤と緑が入り乱れているはずである。

さらに、上記のプログラムをベースにして、配列の内容を乱数で初期化しパネルを描画する操作を 100 回連続して行うように改変せよ[2]。

[1]例えば int spins[NPANEL][NPANEL]; のような配列。

[2]描画と描画の間には、usleep(10000); のような行を挿入して、一定時間処理を休止すると良い

8.3.2 準備２：スピン間相互作用とエネルギーの計算

準備 1 で用意した各々のパネルは、左右と上下 4 本の「腕」で他のパネルと接続されていると見なすことができる。これがスピン間の相互作

用を表す。ここでは、周期境界条件を設定し、左端は右端に、下端は上端に接続されていると考えよう。すると、図 8.1 のように

系の全エネルギー $E = -\sum_{\langle k,l \rangle} J s_k s_l$ を計算することを考えよう。一つの相互作用のエネルギーは隣接するスピンペアの積によって計算される。全ての相互作用についての和を求めるには、各々のスピンの右側と上側の相互作用だけ、全スピンに対して計算するというルールを適用すればちょうど良いので、`-spins[k][l]*(spins[k+1][l] + spins[k][l+1])` を、全部のスピンについて足せば良い。ここで $J = 1$ とした。

一つ気をつけるべきなのは、右端と上端の場合の取り扱いである。周期境界条件は、右端のパネルから右に伸びる相互作用は左端と、上端のパネルから上へ伸びる相互作用は下端とつながっている事を要求する。

この条件を実現するには複数のテクニカルな選択肢がある。

- 上端と右端だけ特別に取り扱う方法
  足し算をループする時、端の一歩手前までのループとし、端の列は別にループする方法。この方法は平易であるが、端を例外として取り扱わねばならず、特に右上角の取り扱いに注意が必要となる。

- 隣のパネル番号を関数で扱う方法（推奨）
  右側あるいは上側に相当するパネルの番号を返す関数を記述すれば、例外的な取り扱いを関数内に封じ込めることが出来る。

  少し考察すると分かるが、実はそのような目的に手頃な関数が標準で存在する。すなわち、除算の余りを求める関数 % を用いれば上記の目的は容易に達成可能で、配列の番号を `(k+1)%NPANEL` のようにするだけで良い。

8.4 熱平衡系のシミュレーション

それではメトロポリス法によるスピンの状態遷移を行おう。ここで、現在のスピン状態から、適当に選んだ格子座標＝ `(k,l)` 番目のスピンだけが反転した状態を考える。反転された状態が採択される確率はメトロポリス法により

$$w\left(S \to -S\right) = \min\left(1, e^{-(E_{-S} - E_S)/k_b T}\right)$$

となる。つまり反転前と反転後の全エネルギーの差によって遷移確率が決まる。具体的には、あるサイトのスピンを反転した場合、エネルギーが

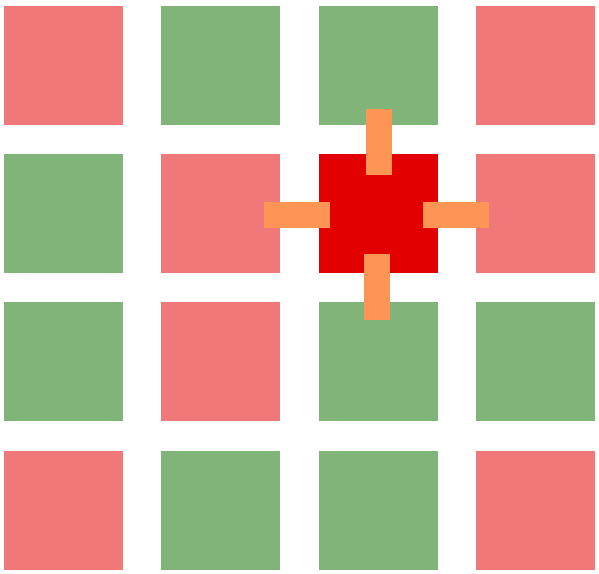


図 8.1: 一つのスピンは 4 本の相互作用で連結されている。

下がれば確率 1 でスピンを反転し、そうでなければ確率 $\exp(-\Delta E/k_bT)$ で反転するかが決まる。採択されなければスピンはそのままの状態を維持する。エネルギー差 ΔE は前課題で作成した関数を用いて計算しても良いが、専用の関数を新たに作成することを奨める。なぜなら、エネルギー差への寄与は選んだスピンの周りの相互作用のエネルギーにのみ由来するからである。あるサイトのスピン S_i が反転した後と前のエネルギー差は、

$$\begin{aligned}\Delta E = 2 * \mathtt{spins[k][l]} * (&\mathtt{spins[k+1][l]} + \mathtt{spins[k-1][l]} \\ &+\mathtt{spins[k][l+1]} + \mathtt{spins[k][l-1]})\end{aligned}$$

と計算され全エネルギーから求めるより効率的である[10]。

実際の手順としては次の通りである。以下 $k_b = 1$ の単位を採用することにして、スピン相互作用定数 J は 1 とする。

[10]実際には、先ほどと同様に端部の処理を考慮し配列の要素番号を与えれば良い。その際、さらに負数の剰余が機種依存であることを考慮し、左と下側のパネルについては、要素番号に NPANEL を加算して、(k+NPANEL-1)%NPANEL を配列の要素番号とすれば統一的な扱いが可能である。

手順 A

(1) 最初スピンの初期状態はランダムであるとする。
↓
(2) 乱数を二つ生成して、反転するスピンの候補 (k,l) を決める
↓
(3) エネルギー差 ΔE を (k,l) の周りのスピンとの相互作用から求める↓
(4) 確率 $p = \exp(-\Delta E/k_bT)$ で 2 で決めたスピンを反転させる[1]。
($1-p$ の確率で、現在の状態を維持する。)
↓
(5) 2 に戻る[2]。

[1] `spins[k][l] = -1 * spins[k][l]` とする事で、スピンを反転出来る。

[2] この 2–4 の過程は、例えば `onestep` のような名前の関数に纏(まと)めた方が良い。

課題 18
課題 17 作成したプログラムを改造して、上記の手順A 1–4 までの過程を模したシミュレーションプログラムを作成せよ。
温度 T は後で変更するとして最初は 1.5 を選択し、
i) スピンの向きが変化して行く様子を観察せよ。
ii) パネルの数を 4×4 から増やすと変化の様子は変わるか？
iii) 変化の様子には、温度の依存性が見られるか？

8.5 物理量の測定

強磁性イジング模型を特徴付ける物理量を測定する事を試みよう。系の状態を表す物理量として磁化 m・磁化率 χ・エネルギー E・比熱 C を取り上げるとそれぞれ、

$$\begin{aligned} m &= \left\langle \frac{1}{N}\sum_k s_k \right\rangle_{\mathrm{MC}}, \\ \frac{\chi T}{N} &= \left\langle \left(\frac{1}{N}\sum_k s_k\right)^2 \right\rangle_{\mathrm{MC}} - \left\langle \frac{1}{N}\sum_k s_k \right\rangle_{\mathrm{MC}}^2, \end{aligned}$$

$$
\begin{aligned}
E/N &= \left\langle -\frac{1}{N}\sum_{\langle k,l\rangle} J s_k s_l \right\rangle_{\mathrm{MC}}, \\
CT^2/N^2 &= \left\langle \left(-\frac{1}{N}\sum_{\langle k,l\rangle} J s_k s_l \right)^2 \right\rangle_{\mathrm{MC}} - \left\langle -\frac{1}{N}\sum_{\langle k,l\rangle} J s_k s_l \right\rangle_{\mathrm{MC}}^2
\end{aligned}
$$

と計算される[11]。添え字の MC は手順 B の計測を複数行った場合の平均＝モンテカルロ平均を表す。エネルギーなどは系のサイズ $N \equiv \mathrm{NPANEL}^2$ に比例するので、事前に割っておくとサイズ比較の際便利である。

物理量の測定の手順は、次のようになる。8.4 節の手順A 1–4 の過程のうち 2–4 の繰り返し (MCS) 部分を用い、

手順 B

(1) 最初に系が平衡状態に達するまで $10000 \times N$ 回以上 MCS する。(N はスピンの個数)

↓

(2) 自乗磁化 $\left(\sum_k s_k/N\right)^2$ と 1 スピンあたりのエネルギー $-J\sum_{<k,l>} s_k s_l/N$、をサンプリングし、適当な配列に格納する[1]。

↓

(3) 2 の計測から系が十分に時間が経過し、計測が独立なものと見なせるまで、MCS する。回数としては、$100 \times N$ 以上とする。

↓

(4) 2 に戻る

[1]後で平均値と分散が計算出来れば良いだけなので、配列に格納せず計測ごとに計測値とその自乗を足し上げて置くだけでも良い。

のような手順となる。最後に、(2) の操作で求めたサンプル列の平均を求め、そこからそれぞれの物理量を計算する。比熱はエネルギーの分散から計算してもよいし、(2) の操作でエネルギーの自乗を格納してもよい。この操作を温度を数点変えながら繰り返す事で物理量の温度依存性を調べる事ができる。また、上記の過程を、異なる初期状態を生成して繰り返す事で、初期状態に対する依存を減らす事が出来る。

[11]それぞれの物理量の定義から導いて確認せよ。磁化率は磁場の入ったイジング模型のハミルトニアンから出発すると良い。

課題 19
課題18で作成したプログラムを改造して、手順Bの過程に従い、物理量を測定するプログラムを作成せよ。
最初、例えば物理量がエネルギー ($E = -\sum_{<i,j>} J s_i s_j$) であれば、隣り合うスピン (i,j) 全ての組み合わせについての和、全磁化であれば全スピンの和 $M = \sum_i s_i$ を計算し、ターミナルに表示する関数を作成することから始めよ。
計算は時間がかかることが多いので、可視化は省くと良い。

1. 物理量の温度依存性を求め図示せよ。サンプリング回数は例えば 10,000 回とする。
2. 次節「発展」を参照しながらイジング模型により、磁性体の電子スピンによる物性を評価せよ。

8.6 発展課題

余力のある人は是非次のような課題について挑戦してほしい。研究レベルに達するデータ・考察を期待する。

相転移点 サイズを大きくしながら物理量を測定せよ。転移点とはどのように定義されるか考え、無限系の相転移点を予想し Onsager による厳密解と比較せよ。無限系の厳密なエネルギー・比熱の式を調べ、シミュレーションの結果と厳密解を同時にプロットするとよい。サイズは大きければ大きいほどよいが、待ち時間を十分考慮する必要がある。

3 次元 2次元イジング模型を拡張し3次元イジングモデルに対してシミュレーションを行え。転移点はおよそいくらくらいになるだろうか。

反強磁性模型 相互作用定数 J が -1 の場合はどうなるだろうか。まずは低温で視覚表示し、何を秩序パラメータにするのが尤もらしいか考えよ。物理量は強磁性模型と比較してどうだろうか。

最適化 以上の計算は非常に計算量が多く完了するまでに時間を要する。プログラムを高速化するために、どの部分の計算に時間を取られて

いるか考え最適化し、どの程度高速化されたかを調べよ。ヒント: 1. 隣のスピンを調べるために毎回 `if` 文や % 演算子を用いるのは無駄である。事前にテーブルを作って参照すると良い。2. エネルギー差を求める際の exp の引数の取り得る値は限られていないだろうか。これもテーブルで事前に計算し格納すると良い。3. やや技巧的だが、`if` 文 (条件分岐) を用いずにスピンの反転を行う方法を考えよ。

第9章　数値処理

コンピューターによるデータ取得が一般化している現代の物理学実験において、測定とは何らかの物理量や付随した量を数値として計測する事を意味する。得られた数値自体が最終的な結果を意味している場合もあるが、多くの場合数値間の相関や時間変化などの考察を通してより本質的な情報を抽出する必要があり、その過程をデータ解析と呼ぶ。

本章では、そのようなデータ解析一般において役立つデータ処理の方法を、統計学的な手法に基づき、スペクトルのフィッティングを中心として取り扱う。

9.1　中心極限定理

同一の実験条件で複数回の計測を繰り返すと、得られたデータは「真値」の周りに統計的な分布をつくる。真値は通常知られていないので、得られたデータから最も信頼できる値を抽出し、その信頼の度合いに応じた誤差を評価しなくてはならない。

このような過程において最も重要な定理の一つが「中心極限定理」である。

> 平均 μ 標準偏差が σ の**任意**の確率分布に従うデータを N 回計測したデータの和は、平均 $N \times \mu$ 標準偏差が σ の正規分布に従う。

また、特に “データの和” の代わりに “N 回計測の平均値” を考えると、平均値の分布は 平均 μ 標準偏差が $\sigma/\sqrt{N}$ の正規分布に従う。

この定理のポイントは、**任意**の確率分布に従うデータの和が、**正規分布**に従う点である。例えば 7 章で扱った、区間 [0:1) の一様分布に従う乱数の場合を考えてみよう。元の分布は 区間 [0:1) の一様分布であるから、平均は 0.5、分散は 1/12、標準偏差 ~ 0.29 である。

図 9.1 は、上記乱数をそれぞれ 100 回、1000 回、10000 回発生しその平均値を求める試行を 10000 回繰り返した結果得られた分布である。結果のヒストグラムを正規分布でフィットして得られた曲線を実線で、右上のインセットに結果を表示している。

得られた分布の平均値は ほぼ 0.5 である。また、標準偏差は 0.028, 0.01, 0.0030 となった。中心極限定理から期待される平均は 0.5、標準偏差は $\sigma/\sqrt{N}$ であるから 0.029, 0.01, 0.0029 であるから良く一致している事がわかる。これらの曲線は、平均値の確率分布関数そのものであるから、このことが示しているのは、平均値の決定精度は、観測回数の平方根に比例して良くなるという事であり、そのばらつきは、$\sigma/\sqrt{N}$ 程度であるという事である。

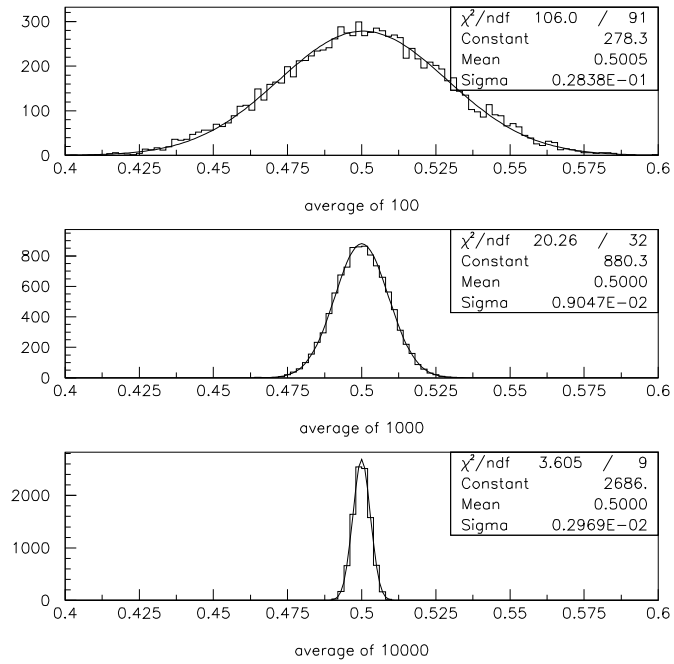


図 9.1: 区間 [0:1) の一様分布乱数を、上から順にそれぞれ観測回数 100 回、1000 回、10000 回発生し平均を取る試行を、それぞれ 10000 回繰り返した結果の分布。実際の観測の際には普通一度しか試行しないのでこの分布の “どこか” に観測値は来る事になるため、発生数に応じ確率分布の幅、すなわち決定精度は異なる事を示している。

上記の例を、物理量の観測に置き換えてみよう。ある物理量を数値として観測する場合、検出される数値は入力された物理量に応じた確率分布に従う。この確率分布は、検出器の応答関数と呼ばれる。

もちろん、応答関数自体、実験や解析により与えられる条件に依存して物理量の「どの」部分の応答であるのかは変わる。単純で理想的な検出器の応答関数は入力された物理量の大きさそのものであり、入力に対して単調な関数を中心とし、検出器の分解能を幅とするような正規分布となる場合が多いだろう。

中心極限定理は、このような検出器を用いた観測の際に得られる精度を与える。すなわち、検出器の応答関数の形によらず、N 回の観測による平均値の確率分布の標準偏差は $\sigma/\sqrt{N}$ となるから、平均値の決定精度は $\sigma/\sqrt{N}$ (1σ) となるのである。

9.2 観測データを支配する統計

特にヒストグラムのようなカウント数を扱う場合、各ビンの高さ則ち事象の出現頻度はポワッソン分布を形成すると期待されるので、応答関数の上下にポワッソン分布から予想されるばらつきでデータが分布する。

例えば、NaI シンチレータと光電子増倍管で ^{60}Co から放射されるガンマ線を検出した場合を考えよう。この場合、NaI 結晶に入射した 1333 keV と 1172 keV のガンマ線は光電効果またはコンプトン散乱によって電子に運動エネルギーを与え、この電子が検出機内の電子を励起した結果発生する可視光領域の光子が光電子増倍管に入射して計測される。

上記の過程を支配している統計は、ポワッソン統計である。走行電子により励起される電子の数は、走行電子のエネルギー損失を電子を励起する仕事関数で割った数になり、励起電子が脱励起して放出される光子が光電子増倍管に入る確率は脱励起の起こる場所や波長に応じたポワッソン分布に従う。

ここで例えば、1333 keV のガンマ線の全エネルギーが検出器に吸収された場合、平均で 10,000 個の光子が観測されるとしよう[1]。すると応答関数は平均値 10,000 のポワッソン分布により与えられるが、これは平均値が十分大きいので[2]平均値 10,000、標準偏差 100 の正規分布で良く近似

[1]実際にはエネルギーの一部は検出器外部に逃げ出すので、より複雑な応答関数となる。

[2]ポワッソン分布の平均値 μ が おおよそ 10 以上の場合、ポワッソン分布は分散が $\sqrt{\mu}$ 平均値が μ の正規分布により良く近似される。

される。従って、この検出器の分解能は、1333 keV に対して 1 % の $\sigma =$ 13.3 keV 程度であると言うことが出来る[3]。

課題 20
以下の思考実験を行い、推定される観測値と誤差を評価せよ。ある物理量を 10,000 回観測したら、その分布スペクトルは 99.5 から 100.5 のヒストグラム 1 ビンに全データが入った。このスペクトルから推計される物理量の値と誤差はいくらか。

9.3 パラメータ・フィッティング

例えば、実験で得られたデータをヒストグラムの形で整理し、なんらかのスペクトルを形成したとする。スペクトルを予想曲線でフィットする事は、何らかの物理的な情報を、その統計的な信頼性と共に抽出する為に有効な手段である。以下では、二次元データ列 $\{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ を、θ をパラメータとする関数 $y = f_\theta(x)$ でフィットする場合を考え、二つの異なるフィッティング手法を学ぶ。

最尤法

パラメーターフィッティングのうち一つ目の方法は最尤法(Maximum Likelihood Method) と言われる。

最尤法では、θ をパラメータとする関数を用いて、各データ点 (x_i) が得られる確率密度関数を $p_\theta(x_i)$ と記述した場合、尤度関数 (likelihood function) $\mathcal{L}(\theta)$ を、$\mathcal{L}(\theta) \equiv \Pi_{i=1}^{m} p_\theta(x_i)$ で定義する。この尤度関数を最大化するパラメータセットをもってフィット結果とするのが、最尤法である。通常は積を和に変換するために $\log \mathcal{L}(\theta)$ を最大化するパラメーターを探すので、log likelihood とも呼ばれる。

特にデータがカウント数である場合、各データ点での確率密度関数は仮定したフィッティング曲線を平均とするポワッソン分布で与えられる。図9.2

[3] この分解能は、検出器の設計精度や較正とは無関係である点に注意してほしい。ここで述べているのは統計誤差であり、観測値には例えば較正精度のような観測条件に起因する系統誤差が別に付随する。また、NaI 検出器の分解能は通常 $\sqrt{w + E}$ で推移する。

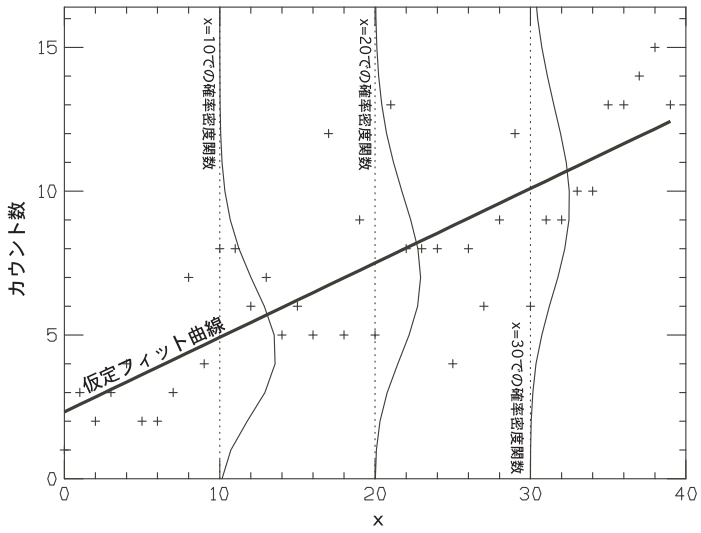


図 9.2: 最尤法の概念図。仮定フィット曲線の周囲には確率密度関数が定義される。各データ点の位置での確率密度関数の大きさの積が最大になるようなフィット曲線をもってフィットを行う手法である。

には、最尤法の概念を示した。予想曲線を平均とする確率密度関数が定義出来るので、各点について観測されたデータの “予想出現確率” $p_\theta(x_i)$ が計算できる。ある観測データセットが観測される確率は、$\mathcal{L}(\theta) \equiv \Pi_{i=1}^{m} p_\theta(x_i)$ で与えられるので、これを最大化するというのが主旨である。この手法は、任意形状の確率密度関数に対して適用可能であり、汎用性がある。例として示した図 9.3 は、数十程度のパラメータ空間上で尤度関数最大を求めたフィットの例である。

最小自乗法

最尤法は強力な手段であるが、スペクトルのフィットで、ヒストグラムの各ビンのカウント数が 10 以上と十分大きい場合には、確率密度関数が正規分布で近似され、さらに簡便な方法として最小自乗法を用いることができる。

まず、予想曲線全体にわたる予想曲線とデータ点のズレの大きさを表す量として以下の様な量を定義しよう。ここでは、カウント数の統計誤

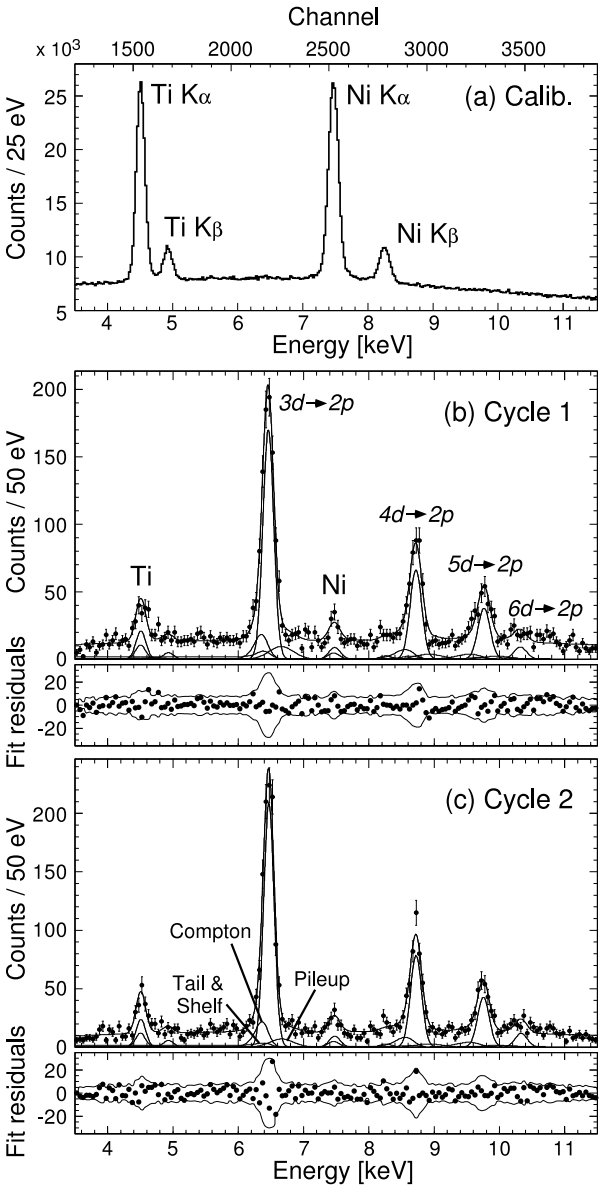


図 9.3: パラメーターフィッティングの例：KEK-PS E570 実験スペクトル。K 中間子原子 X 線のエネルギースペクトルを数十個のパラメーターで記述される関数でフィットして、中心エネルギーと幅、遷移強度の情報を抽出している。(S. Okada *et al.*, Phys. Lett. **B653**, 387 (2007).)

差 $\Delta_i \equiv \sqrt{f_\theta(x_i)} \sim \sqrt{y}_i$ である。

$$\chi^2 \equiv \sum_i \frac{(f_\theta(x_i) - y_i)^2}{\Delta_i^2} \tag{9.1}$$

関数 $f_\theta(x)$ は予想曲線を表す関数、(x_i, y_i) は、横軸が計測値、縦軸がカウント数である。

このズレの大きさ χ^2 を最小 $\chi^2_{\rm min}$ にするような関数 $f_\theta(x)$ を決定する。この事を最小自乗法によるフィットを行うと言う。特に y_i が正規分布に従う場合 χ^2 はカイ二乗分布に従う。データ点数が m で、パラメーター数が k の場合、分布は自由度 $n \equiv m - k$ のカイ二乗分布となり、$\chi^2 = n$ で確率分布がほぼ最大となる。

関数 $f_\theta(x)$ は例えば $y = a \times x + b$ のようにいくつかのパラメータ（この場合は a と b）から記述される。フィットの際には、これらのパラメータを変えながら χ^2 が最小となるパラメータセットを探す事がフィットの実質的な部分である。

さて、このようにして得られた χ^2 の最小値 $\chi^2_{\rm min}$ を与えるパラメータセットとフィット関数が、もっともらしいかどうかは調べてみる必要がある。予想曲線が真値をうまく記述し、カイ二乗分布から予想されるズレの程度が適切であるかどうか。自由度 n のカイ二乗分布は $\chi^2 = n$ で確率分布がほぼ最大になるので、得られた $\chi^2_{\rm min}$ は n から大きくはずれなさそうである。

図 9.4 は、$\chi^2_{\rm min}$ を 自由度 n で割った数値[4]縦軸に表示された値よりも大きくなる“確率” を、横軸=自由度に対してプロットした図である。10% あるいは 90 % の意味は、フィットすると 10 回に 1 回しか、10 % の線より $\chi^2_{\rm min}/n$ が大きくなったり 90 % の線より小さくなったりする事はない事を意味する。つまり、フィットの結果 $\chi^2_{\rm min}/n$ が 10 % の線より大きくなった場合、統計的に見てそのフィットは“なかなか起きない事が起きた” という事を意味し、予想曲線が適切かどうか、誤差の評価は正しいか、見直さなくてはならない。また、$\chi^2_{\rm min}$ が自由度程度に収まったからと言って、予想曲線が真の物理現象を記述しているとは“限らない” 点にも注意を払う必要があろう。

それでは、以上で得られた最小自乗法によるフィットに対してどのような誤差を設定すれば良いだろう。答えは、$\chi^2 = \chi^2_{\rm min} + 1$ となる空間を包含する曲面が $1\sigma = 68.3$ % 信頼区間となる。

[4]通常 Reduced χ^2 と呼ばれる。

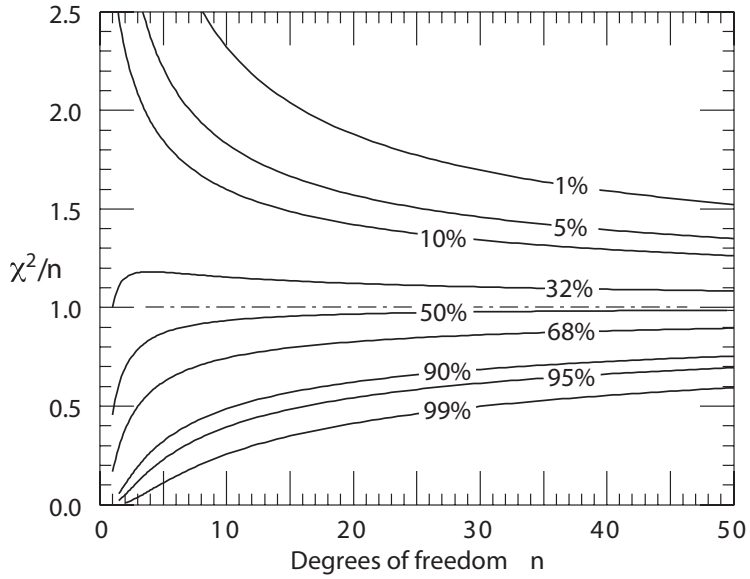


図 9.4: 自由度 n あたりの $\chi^2_{\rm min}$ が、縦軸の数字より大きくなる「確率」を自由度に対してプロットした図。(Review of Particle Physics 2006 より)

9.4 計算機を用いたフィッティング

それでは実際にフィットを実行してみよう。データ点のフィットには、ライブラリやツールを用いる場合が多い。原子核・素粒子の業界では、欧州合同原子核研究機関 (CERN) で開発されたMINUITと呼ばれる[5]ライブラリが有名である。MINUIT については付録に使い方を紹介した。ここでは、より簡便な gnuplot を用いたフィットをためしてみよう。

[5]最近は、C++で書き直された MINUIT2 が使われている。

課題 21
説明する手順に従い、模擬データを生成し解析せよ。
全体の流れは以下の通りになる。

1. フィットに使う模擬データを生成する。
2. 模擬データを加工して、gnuplot で読めるデータ形式にする。
3. gnuplot でスペクトルを表示し、フィットを実行する。
4. フィットの良さを確認・検証する。

それでは細かい内容を以下で説明しよう。

1. 模擬データ生成

まず、解析に用いるデータを生成しよう。自分の学籍番号の下 5 桁を用意し (例えば 12345 とすると)、ターミナルから、
`tsubame% gendata 12345`
のようにしてみよう。

```
115
126
119
121
125
113
....
```

のような、数列が出現したはずである。この数列が、解析データである。
生成した数列は 200 個の数値からなりその意味は、横軸変数を x として、x=0 から x=199 に対応するヒストグラムの高さである。

2. データ加工

それでは、上記の解析データをファイルに保存しよう。例えばこの “生データ” を rawdata.dat というファイルに保存するなら、

```
tsubame% gendata 12345 > rawdata.dat
```
とすれば良い。

このまま縦軸の数値が並んだだけのデータ形式では解析しにくいので、縦軸、横軸と統計誤差が並んだデータ形式、つまり

```
0 115 10.723805
1 126 11.224972
2 119 10.908712
3 121 11.000000
4 125 11.180340
5 113 10.630146
....
```

のような形式にして、gnuplot が読み込める形に加工してみよう。

加工するためにはいろいろな手段が考えられる[6]が、授業では C 言語のプログラムを用意する事にしよう。

サンプルコード 9.1: convdata.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7    int x,y;
8    FILE *fp;
9    int i;
10
11   fp = fopen("rawdata.dat","r");
12   if (NULL == fp){
13     fprintf(stderr,"file␣open␣error\n");
14     exit(1);
15   }
16
17   x=0;
18   while(EOF!=fscanf(fp,"%d",&y)){
19     printf("%d␣%d␣%f\n",x,y,sqrt(y));
20     x++;
21   }
22   fclose(fp);
23 }
```

[6]この程度の加工であればawk（オーク） で `tsubame% gendata 12345 | awk '{print NR-1,$1, sqrt($1) }'` とでもするのが簡単だろう。

このサンプルコードでは、関数 fopen を用いて、“rawdata.dat” ファイルを読む準備を行いポインタ変数 fp に関連づける (この事をファイルを開くと言う)。ファイルが正しく開けたことを fp のアドレスが NULL (ヌル) (空) でないことにより確認した後、fscanf 関数でファイルの中身を一行ずつ読み込む。while(EOF!=fscanf(fp... の部分の意味は、ファイルの最後 (EOF) まで変数 y に整数値 (%d で変換なので正確には 10 進数符号付き整数、表 B.1 を参照。) を一個ずつ読んでくるという意味である。printf 文で、x,y,sqrt(y) を並べた形で出力し、最後に fclose(fp) によりファイルを閉じる。

上記を実行した結果をリダイレクトして、

```
0 115 10.723805
1 126 11.224972
2 119 10.908712
3 121 11.000000
4 125 11.180340
5 113 10.630146
....
```

のような形式の fitdata.dat という名前のファイルを作る事にしよう。

3. gnuplot によるフィット

まずは、fitdata.dat をプロットしてみよう。

```
gnuplot> plot "fitdata.dat" using 1:2
```

データは id 番号ごとに異なるので全く同じ図にはならないが、図 9.5 のようなスペクトルが得られたはずである。

図を見て分かるように、スペクトルは中心付近にピーク構造を持つ。S/N 比は良くないらしく、ピークは傾いたバックグラウンドに乗っている形をしている。ピークの形は正規分布であり、バックグラウンドは一次関数であると仮定してフィットを行ってみよう。すなわち

$$y = f_\theta(x) = a + bx + c\exp\left(-\frac{(x-d)^2}{e^2}\right) \tag{9.2}$$

のような、a 〜 e までの 5 パラメータにより記述される関数でのフィットである。

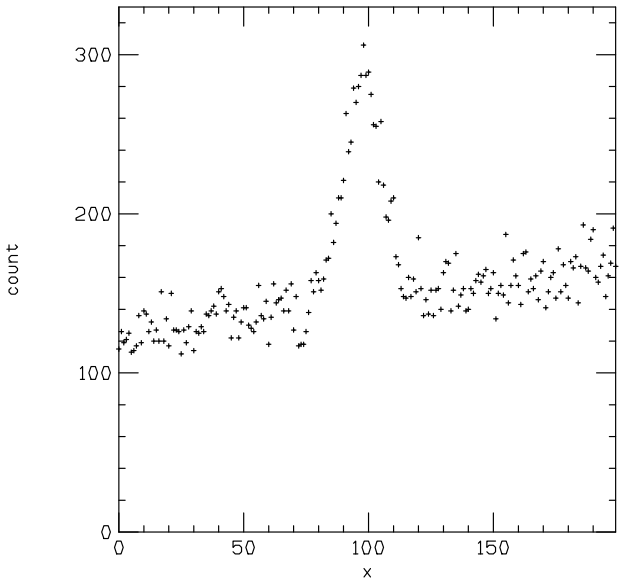


図 9.5: サンプルデータのプロット

まずは、先ほどの図をよく見て、パラメータ a,b,c,d,e のフィット結果を**予想**して初期値を設定しよう。パラメータ a は y 切片であるから、図の場合だいたい 120 である、b は傾きであるから... のように図から読み取って予想値を準備する。

その上で、gnuplot を起動し、以下のように入力してフィットを実行する。gnuplot では、x の y 乗は `x**y` と記述する点に注意して欲しい。

```
gnuplot>  plot "fitdata.dat" using 1:2:3 with yerrorbars
gnuplot> a = 120
gnuplot> b = 0.2
gnuplot> c = 150
gnuplot> d = 100
gnuplot> e = 10
gnuplot> f(x) = a + b*x + c*exp(-(x-d)**2/e**2)
gnuplot> fit f(x) "fitdata.dat" using 1:2:3 via a,b,c,d,e
gnuplot> plot f(x),"fitdata.dat" using 1:2:3 with yerrorbars
```

上記の a = 120, b = 0.2 ... の部分がフィットの初期値を与える部分で、f(x) = ... の部分がフィット関数を与える部分。fit f(x) ... via a,b,..,e

の部分が、a,b,...,e をパラメータとしてフィットを行うコマンド、最後に結果をプロットしている。

上記を毎度入力するのは面倒なので、適当な名前のファイル fit.gnuplot を用意して、中身を

```
plot "fitdata.dat" using 1:2:3 with yerrorbars
a = 120
b = 0.2
c = 160
d = 100
e = 5
f(x) = a + b*x + c*exp(-(x-d)**2/e**2)
fit f(x) "fitdata.dat" using 1:2:3 via a,b,c,d,e
plot f(x),"fitdata.dat" using 1:2:3 with yerrorbars
```

のようにし、

```
gnuplot> load "fit.gnuplot"
```

としても良い。

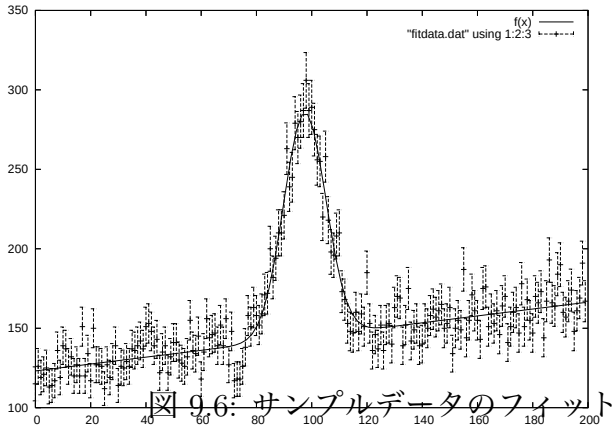


図 9.6: サンプルデータのフィット

フィットの結果は、図 9.6 のように表示されると思う。上記のフィットにより、ターミナルには長い出力が流れていくと思うが、大事なのは以下の部分である。

フィットの出力

```
....
....
After 5 iterations the fit converged.
final sum of squares of residuals : 169.02
rel. change during last iteration : -5.06785e-06

degrees of freedom (ndf) : 195
rms of residuals      (stdfit) = sqrt(WSSR/ndf)      : 0.931005
variance of residuals (reduced chisquare) = WSSR/ndf : 0.86677

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 123.126          +/- 1.561        (1.268%)
b               = 0.216355         +/- 0.01372      (6.34%)
c               = 141.436          +/- 5.064        (3.581%)
d               = 97.4537          +/- 0.2839       (0.2913%)
e               = 10.7799          +/- 0.3984       (3.696%)
```

まず、フィットが収束 (converge) している事：After 5 iterations the fit **converged**. 次に、自由度 (ndf) が 195 であり、カイ二乗を自由度で除した値 (reduced chisquare) が 0.86677 である事が分かる。最後に、フィット結果となるパラメーターの値が誤差付きで表示されている。

4. フィット結果検証

上記で得られたフィット結果は、どれくらい正しいのだろうか。自由度 195 で reduced chisquare が 0.867 は、図 9.4 では右側に切れているが目の子で延長してみると、68 % 線の上くらいだろうか。それほどひどいフィットではなさそうである。

フィットの結果の正しさは、より多面的にみる必要がある。一つの有効な手段は、フィットの残差をプロットする事 (図 9.7 参照) である。残差プロットに構造が残っていないか、2 σ を越える残差の数が 4.5 % 程

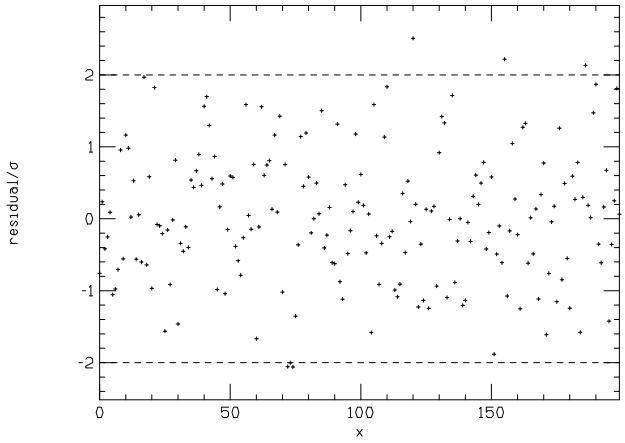


図 9.7: サンプルデータのフィット残差。残差に構造がみられない事、2 σ を越える点が5、まんべんなく分布している事から、フィットとしては”正しそう” である。

度になっているか、まんべんなく分布しているかどうかなど、総合的に判断する事ができる。

フィットの誤差は、既に結果の一部として得られているが、特にパラメータ間の相関が強い場合、設定された誤差が過大評価されている場合もあるので、$\chi^2 = \chi^2_{\rm min} + 1$ を与える領域をパラメータ空間上でプロットして見ることは大切である。領域を等高線の形で描くことで、パラメータ間の相関や、フィット状態の遷移が一目瞭然に分かる場合がある。

以上が、フィットの手続きになるが、フィットが正しそうだからといって、仮定したフィット関数が正しいことを証明出来る訳ではない。フィットが正しいことは必要条件にはなるが、十分条件となるわけではない事に注意して欲しい。

上記は、おそらく “一発” でフィットが出来る比較的簡単な例である。一般のパラメータフィッティングでは、例えば初期値の設定が悪かったりすると、パラメータ空間中でパラメータが極小点（ローカルミニマム）に落ちこんでしまい、最小点に到達出来ない場合や、フィットが収束せずに発散してしまう場合がある。このようなフィットを行う場合には、パラメータの挙動に特に注意する必要がある。

また、パラメータの取り方にも注意が必要である。上記例では、区間 [0:199] を定義域としていたが、これが [1000:1199] であったらどうであろうか。これを　`y = a * x + b` 型の関数でフィットをすると、パラメー

タ a と b の間の相関が非常に強くなってしまう。このような場合フィットがうまく収束しない場合もある。例えばデータの重心位置を見て、y = a（x-1100）+ b というような関数でフィットする事で、パラメータ間の相関を弱めることが出来る。

付 録A　UNIX

A.1　コマンド

以下の表 A.1 では、UNIX を対話的に使うために用いるコマンドの一部を紹介する。UNIX には、夥しい数のコマンドが備えられている。使っているうちになれるので、様々なコマンドを試してみると良い。

UNIX のコマンドは、ほとんどが 5 文字以内でタイプする回数が少なくて済むようになっている。しかしそれでもものぐさなもので、タイプする回数は少なければ少ないほど良いと思うのも人情だろう。学術情報センターの標準設定では、ターミナルでユーザーと対話的な処理を行っているプロセスは bash (Bourne Again SHell) と呼ばれるプログラムである。bash は プロンプトを表示したり、コマンドを解釈したりするが、ユーザーと対話的な処理を行う際に、様々な方法で入力をサポートしてくれる。例えば、ホームディレクトリで、`cd buturi` とする際、`cd b` までタイプしたところで、[TAB] キーを打って欲しい。自動で `cd buturi/` と補完したはずだ[1]。ここでリターンキーで確定しても構わないし、さらに [TAB] キーを打つことで次に補完出来る選択肢を表示させる事も出来る。コマンド名についても同様に補完が出来るので、`rmdir` と打つ代わりに `rmd` とタイプした後 [TAB] キーを打てば良い。

よく言われることだが、UNIX は “小さなプログラムを組み合わせて機能を実現する” という設計思想を持つ。例えば、表 A.1 で紹介している全プロセスを表示するコマンド `ps auxw` をすると、数十個のプロセスが表示される。ターミナルがスクロールしてしまい全プロセスを眺めるにはスクロールバーを動かして見なくてはならない。ここで `ps auxw | less` とするとどうだろう。画面左下に見覚えのある “:” が出現したはずだ。ps コマンドの出力を less コマンドで見ることが出来るのである。プロセスの数を数えたいなら、`ps auxw | wc -l` として行の数を数えるコマンド wc を、文字列を検索したいなら、`ps auxw | fgrep Library` の

[1] ホームディレクトリに、他に b から始まるファイルやフォルダがあれば別だが...

ように fgrep コマンドを “|” の後ろに書けばよい。この “|” はパイプと呼ばれ、パイプの前のコマンドの標準出力を、パイプの後ろのコマンドの標準入力に連結する[2]。慣れてくると `tar cpvf - . | (cd /tmp; tar xpvf -)` のような連結によって、ディレクトリの再帰的なコピーをしたりするようなより複雑な機能を実現出来るようになる。

これまで対話的に利用する場合を示してきたが、バッチ処理させるのは容易である。コマンドを並べたファイルを作成して、それをバックグラウンドで実行するだけで良い[3]。例えば、

```
#!/bin/sh

cd buturi/chapter4
pwd
ls
./mcpi
```

のようなファイル、シェルスクリプト tmp.sh を作って、

```
tsubame% sh tmp.sh &
```

とすればバックグラウンド実行が開始される。この & 記号がバックグラウンドでの実行を意味する。このような単純な例だけでなく、emacs を起動する時にバックグラウンドで起動する事もできるし、シェルスクリプトを使えば、繰り返し実行や条件分岐実行も可能になるので、興味が有れば調べて欲しい。

A.2 エディター

UNIX は標準では vi というエディターと、Emacs という高機能な作業環境が提供されている (場合が多い)[4]。授業の環境では、Emacs の使用を推奨する。Emacs は、

```
tsubame% emacs filename.c &
```

[2]つまり、UNIX で標準出力はターミナルに表示する事を主目的としたものではない。ターミナルに表示したいものは、標準エラー出力に出力すべきである。

[3]本当に長時間走るジョブなどは、システムの推奨する方法でジョブを流す必要がある。

[4]慣れれば使いやすいという意味も含めて Emacs を推薦しておく事にする。ただ、日本語の入力は困難な場合もある。

`ls`	ファイルやディレクトリ (Windows で言うところのフォルダ) のリストを表示する。`ls -laF` とすることで、より詳細な情報を表示する。
`pwd`	現在の作業ディレクトリを表示する。
`mkdir`	新しいディレクトリを作成する。
`rmdir`	空のディレクトリを消去する。
`cd`	作業ディレクトリを変更する。`cd` *new_dir* のようにして変更する。`cd` とだけうつとホームディレクトリに、`cd ..` とうつと、一つ上のディレクトリに移動する。
`cp`	ファイルをコピーする。`cp ../chapter2/example1.c .` のように使用すると、一つ上のディレクトリ下にある chapter2 ディレクトリから example1.c というファイルを、カレントディレクトリ (ピリオドで表す) にコピーする。
`mv`	ファイルやディレクトリを移動したり名前を変更する。
`rm`	ファイルを消去する。
`less`	ファイルの内容を対話的に表示する。ファイルの内容はテキストなど可読でなくてはならない。less が起動したら、中で使うコマンドは、`q` で終了、`j` で一行進む、`k` で一行前へ、[SPC] で次のページへ移動、`b` で前のページへ移動、`/` で下方へ検索、`?` で上方へ検索、などである。
`cat`	ファイルの内容を表示する。
`head`	ファイルの先頭を表示する。
`tail`	ファイルの終端を表示する。`tail -f` で起動する事でファイルの末尾が更新されると自動で表示を更新する事も出来る。
`jobs`	実行中のジョブを表示する。
`ps`	実行中のプロセスを表示する。`ps aux` とする事で全プロセスを表示することもできる。
`kill`	シグナルを送る。例えば、何かのプログラムを実行中に応答がなくなってしまった場合、そのプロセスやジョブに KILL シグナルを送ることで終了出来る場合がある。これをプロセスを殺すと言うが、自分が殺せる全てのプロセスを殺す場合には、`kill -9 -1` とすると良い。
`find`	ファイルを検索する。このコマンドは高機能すぎるので説明は省略する。

表 A.1: UNIX でよく使うコマンド一覧

のように、編集したいファイルの名前を emacs というコマンド名に続けてタイプして起動する。

emacs の基本的な操作は、メニューから行えば良い。emacs にはモードという概念があり、時におかしなモードに入ってしまう事がある。この場合、基本の編集モードに戻るためには、Ctrl-g をタイプすれば良い。

付 録B　C 言語講座

B.1　何をどこに記述すべきか

C 言語でのプログラミングは、文法規則に則った記述をしなくてはならないのは当然であるが、文法に従うだけでは書き方の自由度が大きい。以下では文法に加えて文法に従った上での「流儀」について触れておこう。

一般的な C 言語のプログラムは以下のようなスタイルである。

```
 1 #include <stdio.h>  /* ヘッダーファイルの読み込み */
 2 #include <stdlib.h> /* ヘッダーファイルの読み込み */
 3 ...
 4 #include <math.h>
 5 #include "myhead.h" /* ユーザー作成ヘッダーの読み込み */
 6
 7 static int localVar; /* 大域変数の宣言 */
 8 int Var;             /* 大域変数の宣言 */
 9
10 static double myLocalFunc() /* 関数 myLocalFunc */
11 {
12     ....
13 }
14
15 double globalFunc(double x) /* 関数 myLocalFunc */
16 {
17    ...
18 }
19
20
21 int main()
22 {
```

```
23      double x;
24      ...
25      ...
26      myfunc(x);
27 }
28
```

1 から 6 行目の#define #include #ifdef のような # から始まる行はコンパイラの前過程：プリプロセッサへの命令で、C 言語として、プログラムを解釈する前の段階で処理される。特に #include は、その位置にファイルを読み込む。#define は、続く文字列の定義[1]を表す。#include の後に <...> で囲まれた場合は、システムのヘッダーファイルを読み込み、"" で囲まれた場合は、ユーザー作成ヘッダーファイルを読み込む。システムのヘッダーファイルは、通常 /usr/include 以下に配置され、ユーザー作成ファイルはカレントディレクトリに配置される。

7, 8 行目では、整数の大域変数 Var と localVar を宣言している。10 から 18 行目は関数の定義で、倍精度実数を返す関数 myLocalFunc() と globalFunc() を定義している。21 行目から 27 行目は main 関数の定義である。

実は上記のサンプルコードの 7,8 行目、10, 15 行目を良く眺めてみると、次の事に気づく。7 行目と 10 行目には static という修飾子がついているが 8, 15 行目にはついていない。関数の内部においては static 修飾子は、メモリ上に静的に配置するという意味であったが、関数の外での static 修飾子は、異なる意味合いを持つ。関数外では、続く大域変数や関数が、**他のファイルに書かれたプログラムからは見えない**という事を意味する。分割コンパイルした場合、他のファイルのプログラムで extern int Var; と記述する事で、8 行目の大域変数 Var は共有されるが、同様にしても localVar は共有できない。であれば、static を付ける理由はあるのか？

大域変数を extern 宣言により、共有する事は大変便利である。しかし同時に、深く考えずに用いると極めて危険でもある。分割コンパイルした時、他のファイルに書かれたプログラムから全く予期しない形で変数の値が変更される可能性があるからである。そのため、大域変数は static 宣言するのが普通であり、static を付けないのは、何らかの理由がある

[1] さらに、#ifdef や #ifndef を使って文字列の定義による分岐が可能で、本文中でも #define DEBUG のような形で例示した。

場合だけである[2]。また、extern 宣言を用いてファイル間で変数の共有を行う事は、事情がない限り**全く推奨しない。**

それでは、どのようにして static 宣言された変数にアクセスすれば良い。それには、関数を用いればよい。上記の例であれば、例えば以下のようなアクセス専用の関数を用意してやれば良い。このようなアクセス専用の関数を用意する事で、予期しない形で変数を書き換えられるのを防ぐ事が出来るのである。

```
int getLocalVar()
{
   return(localVar);
}

int setLocalVar(int v)
{
   if (0!=v)localVar = v;
}
```

同様に 10 行目のように、他のファイルのプログラムから呼ぶ必要のない関数には static 宣言を付ける。こうする事で一つのファイル内部でのみ有効な関数を定義する事ができるのである。

上記のような方針に従いプログラムする事で、変数や関数の “セキュリティ” に配慮したプログラムを記述する事が出来る。この事はオブジェクト指向プログラミングへの第一歩ともなる。

B.2　ヘッダーファイルの書き方

前節の例での、ユーザー作成ヘッダーファイル myhead.h の中はこのようなプログラムである。

```
1 #ifndef _MYHEAD_H
2 #define _MYHEAD_H
3
4 int setLocalVar(int);
5 int getLocalVar();
```

[2]例えば、FORTRAN の COMMON ブロック宣言と共有化したい場合など。

```
6 double globalFunc();
7
8 #endif
```

一行目と二行目そして最後の行は、同じヘッダーファイルが二度以上読み込まれないようにするために必要である。4 から 7 行目で、他のファイルから使っても構わない関数の型だけを宣言している。

このような宣言は関数のプロトタイプ宣言と呼ばれ、プロトタイプ宣言が書かれたヘッダーファイルを読み込む事により、あたかもファイルの先頭付近でその関数が定義されているかのようにして関数を使用する事が出来るようになる。注意して欲しいのは、C 言語では、関数は使用される前 (つまりプログラム上の方の行) に宣言されていなくてはならない点である。プロトタイプ宣言する事により、実際の定義を後回しにしたり、他のファイルで定義したり出来るようになる。プロトタイプ宣言は、分割コンパイルの場合に特に有効で、正しく関数の型を定義したヘッダーファイルを用意する事で他のファイルで定義された関数をスムーズに使用する事が出来る。

`$SAMPLE/chapter7/MC-Model` に、この章の内容に即した形で記述した７章のモンテカルロ・シミュレーションのプログラム例を配置した。興味が有れば参照してみて欲しい。

B.3 条件文・制御文

B.3.1 if 文

本文中で、登場する条件文の代表例は if 文である。if 文は、

if (条件式 1) 文 1 ; else if (条件式 2) 文 2 ; else 文 3;

のように記述し、条件式 1 が真であれば文 1 を、条件式 1 が偽で条件式 2 が真であれば文 2 を、どちらも偽であれば文 3 を実行する。else if () の部分は、複数記述出来る。

if 文は、以下のような方法で使用する。

```
...
if (0==i){
  y = cos(x);
```

```
  }else if (1==i) {
    y = sin(x);
  }else if (2==i) {
    y = tan(x);
  }else{
    y = x*x;
  }
  fprintf(stderr,"x,y-> %f,%f\n",x,y);
  ...
```

この例では、i の値に応じて y に代入する数を変えている。これは、三種類の条件を使い分けたい場合などに便利で、上記の例は、一つの関数で同じような形式の四つの関数を表現する場合などに使えるだろう[3]。

条件式は比較演算子で記述される。== は等価演算子で、左辺と右辺の値が同じであれば真となり、そうで無いと偽となる。== と反対なのが != で左右が同じ場合に偽となり、異なる場合真となる。

比較演算子には他に、左辺が右辺より小さい場合真となる < 同様に>があり、さらに左辺が右辺以下の場合真となる<= 同様に>= がある。

条件式を組み合わせて、複雑な条件式を構成する事もできて、条件式１ && 条件式２とすれば条件式１と条件式２の双方が真である場合のみ、真となり他は偽となる。条件式１ || 条件式２とすれば、条件式１と条件式２の双方が偽である場合のみ、偽となり他は真となる。また、!条件式１とすれば、条件式１が真の場合偽となり、偽の場合真となる。

実は、注意が必要なのが文字列の比較の場合である。文字列を比較する場合、例えば

```
{
   ...
   // ダメな例
   char retu[]="mojimoji";
   if ("mojimoji" == retu){
     ...
   }
   ...
}
```

[3]上記の例で、わざわざ “定数 == i” のようにしている点には注意して欲しい。こうしておけば、= を一つ書き忘れてもコンパイラが警告してくれる。

のようにしても期待した通りの動作はしない。この場合期待した動作をさせるには、

```
#include <string.h>
...
{
   ...
   char retu[]="mojimoji";
   if ( strcmp("mojimoji",retu) ){
     ...
   }
   ...
}
```

のように string.h をインクルードした上で、strcmp() 関数を用いると良い。

B.3.2 switch 文

if 文で条件が増えてくると、else if が沢山並ぶ事になり、単純な条件でも視覚的に把握するのが難しくなる場合がある。switch 文は、比較的単純な条件をシンプルに表現するのに向いていて次のように記述する。

```
...
switch(i){
case 0:
  文1;
  break;
case 1:
  文2;
  break;
default:
  文3;
}
...
```

case の後に書いてあるのが、条件式に相当するラベルと呼ばれる部分で、i が 0 だと文 1 から、1 だと文 2 から、それ以外だと文 3 から実行を始める。注意して欲しいのは、文 1 や文 2 の直後に break 文が挿入されている事である。これがないと、例えば文 1 を実行した後続いて文 2 を実行し、さらに文 3 を実行してしまう。switch 文は、あくまで case xxx: で示されたラベルへのジャンプを意味するだけなので、break 文がないと、続いて次の文を実行してしまうのである。

B.3.3 三項演算子

最後の条件文の表現方法が三項演算子である。この演算子は、以下のようにして使用する。

条件 1 ? 式 1 : 式 2

条件 1 が真であれば式 1 が、偽であれば式 2 が評価される。式は文である必要は無いので、

```
...
i==0 ? y = cos(x) : y = sin(x);
y = (i==0 ? cos(x) : sin(x));
...
```

のようなどちらの表現も可能である。i が 0 であれば、y には cos(x) がそうでなければ sin(x) が代入される。

B.3.4 for 文

制御文のうち、最も多く使われるのが for ループでおなじみの for 文である。for 文は本文中に沢山登場しているので説明は省略する。

B.3.5 while 文

while 文は、for 文と同じくループを構成するためのものだが、ループを続ける条件を式で表し、以下のように記述する。

```
...
while(式 1 ){
  文 1 ;
}
...
```

文 1 は、式 1 が真である限り繰り返し実行される。最初の一回で式 1 が偽である場合、文 1 は一度も実行されない。

似ているが動作が少し異なるのが、do 〜 while 文である。

```
...
do {
  文 1 ;
}while(式 1 );
...
```

この場合、文 1 を実行してから、式 1 を評価してループを継続するかどうかの決定を行うため、最初の一回で式 1 が偽である場合でも、文 1 は一度だけ実行されるのである。

B.4 書式指定子

printf や scanf 関数で書式を指定する際に使用する書式指定子についてまとめておく。

B.5 ポインタ

B.5.1 ポインタとは

まずポインタとは何なのかを定義しておこう。ポインタとは、アドレスを格納するための領域である。因みに変数は値を格納するための領域である。アドレスとはメモリ上の位置の事で、実行形式を実行する際には変数であれ関数であれ、メモリ上[4]に展開されている[5] ので、その実際の番地の事である。

[4]太古のコンピューターでは物理的なメモリを指したが、現在はプロセスごとにわりあてられた仮想的なメモリを指す。

[5]メモリ内では、テキスト、データ、スタックの三領域に分類される。機械語の命令はテキスト、静的変数、malloc で割り当てられた変数はデータ、自動変数はスタックに

fprintf() 関数と printf() 関数の書式指定子	書式の意味
%c	1 文字として出力する
%d	符号付き 10 進数として出力する
%u	符号無し 10 進数として出力する
%x	符号無し 16 進数として出力する
%i	符号付き 8 進数として出力する
%o	符号無し 8 進数として出力する
%f	実数として xx.xxx の形式で出力する
%e	実数として指数形式で出力する
%g	実数として短い書式で出力する
%s	文字列として出力する
fscanf() 関数と scanf() 関数の書式指定子	**書式の意味**
%c	1 文字として入力する
%d	4 byte 分の 10 進数として入力
%x	4 byte 分の 16 進数として入力する
%o	4 byte 分の 8 進数として入力する
%ld	8 byte 分の 10 進数として入力
%f	4 byte 分の実数として入力する
%lf	8 byte 分の実数として入力
%s	文字列として入力する

表 B.1: 主な書式指定子

では、アドレスを格納すると何がうれしいのか。ポインタを使う利点が発揮される機会は多いとは言えない[6]が、以下のような場合があろう。

例えば、１００個の要素を持つ配列を関数に引き渡したいとき、１００個の要素の持つ値をいちいち関数に引き渡すのは面倒であるし、時間もかかる。そこで、１００個の要素が連続したメモリ領域に確保されているなら、先頭のアドレスと、１００個引き渡すという情報だけ関数に引き渡せば良い。

実際、これまで既に、

```
...

#define N 100

int func(double xy[N])
{
   ...
   xy[i] = 10.0;
   ...
}

int main()
{
   double xy[N];
   ...
   func(xy);
   ...
}
```

のようなプログラムを書いたことがあると思う。実は、倍精度実数配列 xy[N] を要素番号を書かずに、名前だけで使うと、配列の先頭アドレスを表すのである。従って、main 関数内で、func(xy) としている部分は、配列 xy の先頭番地を関数に渡している。そして、関数 func() の引数 xy はポインタに他ならない。int func(double xy[N]) の部分は double *xy と

配置される。データセグメントはさらに、データ領域とヒープ領域に分けられる。詳細についてはプロセス　スタック　セグメントをキーワードに検索せよ。

[6]ポインタを使う方が高速であるという記述は散見されるが、現在の進化したコンパイラでは、配列の添え字を使って記述した場合との差は無い。

書いても良い。誤解しないで欲しいのだが、配列とポインタは全く別物であるが、特に関数の引数として宣言された配列は例外的にポインタとして *xy のように記述された物と解釈される[7]。

普通のポインタの記法は以下の通りである。ポインタを宣言する場合には、double *p のように記述して、宣言する。(double *) が型名で、p が倍精度実数型のポインタである事を宣言する。

宣言しただけではポインタにアドレスが入っていないので、実際に使うには、ポインタにアドレスを代入しなくてはならない。配列の先頭アドレスを代入するには

```
...
double xy[N];
double *p;
...
p = xy;
...
```

のようにすれば良い。

配列でない変数のアドレスを代入するには、

```
double *p;
double x;
...
p = &x;
...
```

のようにして、アドレス演算子 & を用いる。アドレス演算子は、右辺の変数が確保しているメモリ上の番地を表す。

このようにしてポインタが実際のアドレスを保持すれば、そのアドレスに対して様々な操作を行えるようになる。例えば、倍精度実数の変数のためのアドレスであれば、そこに値を入れることが出来るし、関数の先頭番地であればその関数を実行する事が出来る。

それらの操作を行うためにポインタの保持するアドレスを参照するには間接参照演算子 * を用いる[8]。上記の例であれば、

---

[7]関数の引数の場合に限り double xy[] のようにする事ができ、要素数を書く必要もない。

[8]ポインタ宣言と同じ記号なのがまたややこしいのかもしれない...

```
   double *p;
   double x;
   ...
   p = &x;
   *p = 2.0;
   printf("%f\n",x);
   ...
```

のようにすれば、倍精度実数変数 x のアドレスをポインタ p に代入した上で、ポインタ p の保持するアドレスに 2.0 という値をセットする。結果として x に 2.0 が代入されたのと同じで画面には 2.0 と表示される。これを見ても分かるように “普通” ポインタを使う意味など無い。

ただ、関数で行われた演算結果を、呼び出し元に返す場合には便利な場合がある。例えば以下のような場合である。

```
#include <stdio.h>

int func(double *q)
{
   *q = 2.0;
   return(0);
}

int main()
{
   double x;
   double *p;

   p = &x;
   func(p);
   printf("%f\n",x);
}
```

この例では、倍精度実数変数 x のアドレスをポインタ p に代入して、ポインタ p の保持する番地を関数 func() に渡している。関数 func() の中では、渡されたアドレスを q という名前のポインタで保持し、演算結果である 2.0 という値をポインタ q が保持するアドレス、つまり変数 x の

アドレスにセットしている。このようにして、関数内部での計算結果を、呼び出し元に反映する事ができる。

但し、このようにする事によって、呼び出し元と、関数は強く結合してしまう事にも注意して欲しい。則ち関数 func() の中で、呼び出し元の変数 x の値はいかようにも変更する事が出来る。知らない内に呼び出し元の想像の範疇を超えた値がセットされる危険性があるのである[9]。

B.5.2 ポインタの使い方

それでは、ポインタはどのようにして使うのが推奨されるのか。ポインタを使うべき場合の例としては、上記のように関数に配列 (や構造体) を渡す場合の他に、メモリを動的に確保する場合があろう。

例えば、プログラムを実行して初めて、必要な配列の要素数が分かる場合、スマートなプログラムを書くにはメモリの動的な確保は避けては通れない。

例えば、何らかの計算を行った結果要素数 num の倍精度実数配列が必要になったとしよう。メモリ領域を確保するには malloc() という関数を用いるが、malloc() は確保した領域の先頭の番地を返すので、ポインタを一つ用意して先頭番地を格納すれば良い。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6   int num;
 7   double *p;
 8   num = 10000;
 9   p = (double *)malloc(num * sizeof(double));
10   if (NULL == p ){
11     fprintf(stderr,"Memory allocation error\n");
12     exit(1);
13   }
14   p[200] = 10.0;
```

[9] このような危険を避けるために、オブジェクト指向の考え方の一つ変数のセキュリティというものを意識しながらプログラムする必要がある。

```
15   printf("%f\n",p[200]);
16   free(p);
17 }
```

この例では、7 行目で倍精度実数のポインタ p　を宣言している。8 行目で決定した num 個の要素を持つ倍精度実数配列を 9 行目で確保する。10 から 13 行目で、配列が正しく確保されなかった場合のエラー処理を行う。14 行目で確保された配列の 200 番目の要素に値 10.0 を代入している。16 行目で確保された配列の領域を破棄している。malloc() 関数の前についている (double *) は倍精度実数型ポインタへのキャストである。

ポインタとして露わに登場するのは、7,9,10 行目と 16 行目だけで他は配列としての記法を用いているので、違和感は少ないだろう。実は、上記の 14 行目は、以下のようにも書けるし、このような記法を用いる例は多い。

```
*(p+200) = 10.0;
```

この表記で一見して謎なのは、ポインタに 200 を加算している点である。ポインタはアドレスを保持するが、アドレスに数字を足すとは？倍精度実数のポインタ p は、動的に確保された配列の先頭番地を保持している。これに 200 足すとは、先頭から見て 200 番目のアドレスを指す事に他ならない。従って p[200] と*(p+200) は全く等価である。ポインタに 1 足すと、ポインタの型に応じた分先のアドレスを意味する。倍精度実数は 8 byte なので、8 byte 先のアドレスを、整数なら 4 byte 先のアドレスを指すのである。

B.6　ファイル入出力

これまで、UNIX の機能である リダイレクトを用いてファイルの読み書きをしてきた。当然 C 言語にも指定したファイルに情報を書き込むための関数群は備わっている。以下で、実際にファイルの読み書きを行ってみよう。

```
1 #include <stdio.h>
2
3 int main()
```

```
 4 {
 5    FILE *fp;
 6    ...
 7    if (NULL==(fp=fopen("output.dat","w"))){
 8      fprintf(stderr,"file open error\n");
 9      exit(1);
10    }
11    fprintf(fp,"%f %f\n",x,y);
12    fprintf(fp,"join 1\n");
13    ...
14    fclose(fp);
15 }
```

これは、データを書き出す場合の例である。5 行目で、ファイル型へのポインタを宣言している。このポインタを使って、以降ファイルを開き、ファイルにデータを書き出し、最後にファイルを閉じる操作を行う。ファイルを開くとは、普通ディスク上のファイルと入出力するための準備を行う事である。7 行目で、ファイル型ポインタ fp は “output.dat” というファイルと、書き出しモードで関連づけられる。fopen 関数は二つ引数を取り、一つ目がファイル名、二つ目がモードである。モードには、以下の表のような種類がある。

B.7 FORTRAN ライブラリの使用

FORTRAN は長い歴史を持つプログラミング言語であり、その言語的特性から特に数値計算ライブラリが充実している。ここでは、そのような FORTRAN ライブラリを C 言語から利用する場合の注意点を記述する。

FORTRAN で記述されたライブラリを C 言語から利用する場合いくつかの決まり事を守る必要がある。例えば、以下の FORTRAN のサブルーチン FSUB を C 言語から使う場合を見てみよう。

```
SUBROUTINE FSUB(I,X,ARR,FNAME)
INTEGER I
REAL*8 X
REAL*8 ARR(10)
CHARACTER*32 FNAME
```

r	テキストファイルを読み出すために開く。
r+	読み出しおよび書き込みするために開く。
w	ファイルを書き込みのために開く。ファイルが既に存在する場合には長さゼロに切り詰める。ファイルがなかった場合には新たに作成する。
w+	読み出しおよび書き込みのために開く。ファイルが存在していない場合には新たに作成する。存在している場合には長さゼロに切り詰められる。
a	追加 (ファイルの最後に書き込む) のために開く。ファイルが存在していない場合には新たに作成する。
a+	読み出しおよび追加 (ファイルの最後に書き込む) のために開く。ファイルが存在していない場合には新たに作成する。読み出しの初期ファイル位置はファイルの先頭であるが、書き込みは常にファイルの最後に追加される。

表 B.2: fopen 関数の取るモードの種類。

```
...
...
```

このサブルーチンは、四つの引数を持ち、それらはそれぞれ整数 (INTEGER) I、倍精度実数 (REAL*8)X、倍精度実数配列 (REAL*8 (10))MAT、文字列 FNAME である。

このサブルーチンを C 言語から利用する時は、以下のようにして使用する。

```
1 int main()
2 {
3    int i;
4    double x;
5    double arr[10];
6    char fname[32];
7
8    i=10;
9    x=5.0;
10   arr[0]=1.0; arr[1]=2.0; ....
```

```
11   fsub_(&i,&x,arr,fname,32);
12   ....
13 }
```

決まり事

- FORTRAN のサブルーチン、関数の名前を小文字にし、最後にアンダースコア (_) を付けた名前を用いる。
- 引数はアドレスを渡す[1]。つまり、変数の名前の前にアンポアサンドを付けてアドレス変換する。配列は、配列名を渡す。
- 文字列は、文字列長を引数の最後に列挙する。

[1]アドレスについては、必要に応じて付録 B.5.1 を参照せよ。

上記のプログラム例 11 行目を見てもみよう。まず、関数名は FORTRAN のサブルーチン名 FSUB を小文字に変換し、アンダースコアを付加して `fsub_`となる。次に、変数 `i`, `x` には、変数名にアンポアサンド & を前置させアドレスに変換して渡している。配列 arr[10] と文字列 fname[32] は、配列名と文字列名がアドレスを表すのでそのまま渡す。最後に文字列長 32 を引数として付加している。

以上の注意を守る事で、C 言語から FORTRAN のライブラリを利用出来る。

付 録C　グラフィカルプロット

本文中でも述べたとおり、UNIX は文字ベースのインターフェースを標準としているが、X11 というグラフィックシステムを起動する事で、画像の情報を取り扱う事も出来る。以下では X11 が起動している事を前提とする[1]。

C.1　GNUPLOT

本講義で利用する、学術情報センターの教育用コンピュータには、グラフ作成ソフトウェアである GNUPLOT が既にインストールされている。GNUPLOT は無償のソフトウェアであるにもかかわらず、関数の表示、数値データのプロット、非線形関数によるフィッティング、3 次元プロット、媒介変数表示といった、非常に多くの有用な機能を備えている。ここでは、本講義の課題を進めていく上で必要と思われる、基本的な利用方法について説明する。GNUPLOT は、物理学の研究においてよく利用されるソフトウェアであるので、本講義を通じてその使い方を身につけておけば、将来研究をする上で役に立つかもしれない。実際に自分の手を動かして、GNUPLOT の使い方を習得して欲しい。より詳しい使い方や、ここでは述べられていない、その他の GNUPLOT の機能について知りたい場合には、以下の資料を参考にすると良い。

「使いこなす GNUPLOT」大竹つよし著　テクノプレス

http://t16web.lanl.gov/Kawano/gnuplot/index.html

http://www.al.cs.kobe-u.ac.jp/~inamoto/unix-tools/useful/gnuplot/

http://www.ai.cs.scitec.kobe-u.ac.jp/~tmori/index.php?gnuplot

[1]本章は主に上野祐亮（2007 年度 TA、(株) リコー）によって記述された

C.1.1　起動および終了

まず、gnuplot を起動するには、ターミナル上で以下のように入力する。

tsubame% gnuplot

すると、ターミナル上に

gnuplot>

が表示される。
例えば、ここで

gnuplot> plot sin(x)

とコマンドを入力すると、図 C.1 のように、$y = \sin x$ の関数が画面に表示される。
また、

gnuplot> plot sin(x), cos(x)

と入力することで、$y = \sin x$ と $y = \cos x$ の２つの関数を同時に表示することもできる。
ここで述べた例の他にも、tan(x), exp(x), log(x) などの様々な数学ライブラリ関数が利用できる。
GNUPLOT を終了させるには、

gnuplot> quit

と入力する。

C.1.2　数値データの可視化

GNUPLOT を使うことで、計算結果を簡単にプロットすることができる。例えば、以下のような２列の数値データの書かれたファイル (sample1.dat) があるとする。

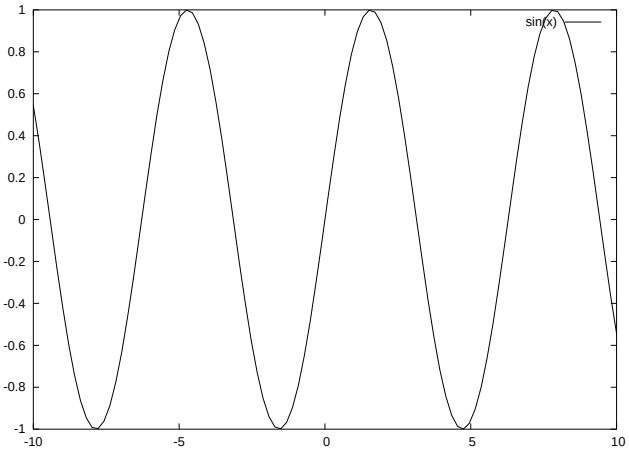


図 C.1: 関数 $y = \sin x$ の表示。

```
# 'sample1.dat'     (コメント行 [2])
# x       y
 1.0      0.6
 2.0      1.1
 3.0      1.4
 4.0      2.2
 5.0      2.3
 6.0      3.1
```

このファイルの 1 列目を x 座標、2 列目を y 座標として、プロットするには、

```
gnuplot> plot 'sample1.dat'
```

と入力すれば良い。出力結果は、図 C.2 のように表示される。
　さらに、

```
gnuplot> plot 'sample1.dat', x/2
```

[2]データファイルの中の、# のついた行はコメント行となり、GNUPLOT では読み込まれない。

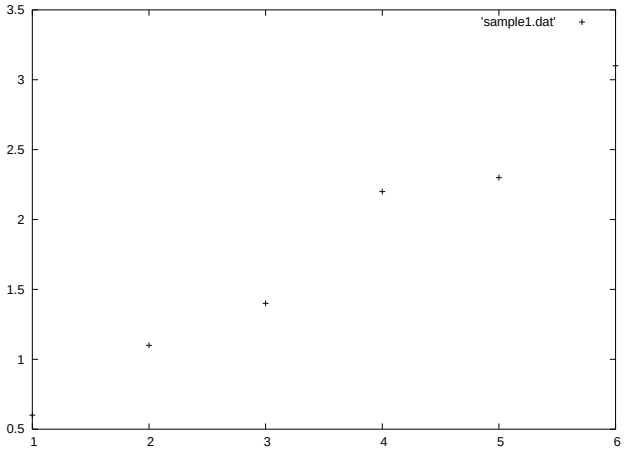


図 C.2: 'sample1.dat' のプロット。

のようにして、数値データのプロットだけでなく、関数を加えることもできる（図 C.3)。

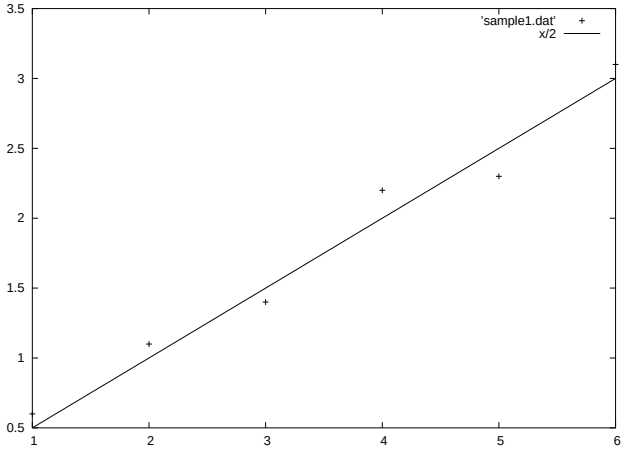


図 C.3: 'sample1.dat' のプロットおよび直線 $y = \frac{1}{2}x$。

次に、複数列の数値データをプロットする方法について説明する。以下のような、３列のデータを持つファイル (sample2.dat) があるとする。

```
# 'sample2.dat'
# x      y1      y2      y3
 1.0     1.0     1.0     1.0
 2.0     1.5     2.0     4.0
```

3.0	2.0	3.0	9.0
4.0	2.5	4.0	16.0
5.0	3.0	5.0	25.0
6.0	3.5	6.0	36.0

例えば、このファイルの 1 列目を x 座標、3 列目を y 座標にとって、プロットするには、

```
gnuplot> plot 'sample2.dat' usi 1:3
```

とする。4 列目を y 座標にとりたいときは、usi 1:4 とすればよい。

なお、同時に 2 列のデータをプロットする場合には、以下のようにすればよい。

```
gnuplot> plot 'sample2.dat' usi 1:3, 'sample2.dat' usi 1:4
```

同様の方法で、3 列以上のデータを同時にプロットすることもできる。

C.1.3 ファイルへの出力

gnuplot で作成したグラフは、PostScript(EPS) 形式のファイルとして保存することができ、紙に印刷したり、TEX の文書に図として貼付けることが可能である。以下に、グラフを PS ファイルとして保存するための、最も簡単なコマンドの例を示す。

```
gnuplot> set term postscript eps
gnuplot> set output 'sample.eps'
gnuplot> plot 'sample.dat' [3]
```

このようにして、作成したグラフが EPS ファイル sample.eps に出力され

[3] set term postscript を入力する以前に、plot コマンドでグラフを既に出力している場合には、単に replot と入力するだけで良い。この replot コマンドは、C.1.4 節で説明するような様々な設定を課してグラフを作成する場合に、改めて同じコマンドを入力する手間を省けるので、非常に役に立つ。

る。[4]

C.1.4　グラフの設定

レポートや論文などの文書にグラフを挿入する場合には、横軸と縦軸がそれぞれ何の物理量を表しているのかを、はっきりと書く必要がある。また、そのグラフが他者から見て理解しやすいものかどうかということも、それらの文書を作成する上で非常に重要なことである[5]。そこで、描いたグラフにタイトルや軸の名前を入れたり、グラフの表示範囲を制限するなど、グラフに条件を課すための基本的なコマンドをいくつか紹介したい。

gnuplot> set title '*title_name*' （グラフにタイトルを与える。）
gnuplot> set xlabel '*xaxis_name*' (横軸に名前を与える。)
gnuplot> set ylabel '*yaxis_name*' （縦軸に名前を与える。）
gnuplot> set xrange [*xmin* : *xmax*] （横軸の範囲を決める。*xmin*, *xmax* にはそれぞれ、最小値、最大値を数値[6]で入れる。）
gnuplot> set log x （x 軸をログスケールにする）
gnuplot> set yrange [*ymin* : *ymax*] （縦軸の範囲を決める。*ymin*, *ymax* にはそれぞれ、最小値、最大値を数値で入れる。）
gnuplot> set key *x*, *y*　（データのラベルを (*x*,*y*) の位置に表示する。*x*, *y* には数値を入れる。）
gnuplot> set nokey （データのラベルを表示しない。）
gnuplot> set pointsize *size*　（データ点のサイズを指定する。*size* には数値を入れる。デフォルトは 1.0。）

これらのうち、必要なコマンドを入力した後に、

gnuplot> plot '*data_file*'

---

[4]実は、付録 C.1 に示した図は全て、set term postscript のコマンドで作られた PS ファイルである。

[5]図 C.1 - C.3 は、グラフが画面上にどのように表示されるのかを示すため、敢えて設定はデフォルトのままにしてある。そのため、グラフを分かりやすくするための工夫が全く施されていないだけでなく、横軸と縦軸が何を表すのかすら書かれてない。本来、このようなグラフを、レポートや論文に載せるべきではない。

[6]円周率 π も利用できる。例えば、$-\frac{\pi}{2} \leq x \leq \frac{\pi}{2}$ としたい場合には、[-pi/2 : pi/2] とする。

と入力すれば、設定した条件に従って、グラフが表示される。

なお、plot コマンドには、オプションをつけることができる。例えば、以下のようなオプションがある。

gnuplot> plot '*data_file*' with lines　（データ点を折れ線でつなぎ、表示する。）
gnuplot> plot '*data_file*' with linespoints　（データ点と折れ線を同時に表示する。）
gnuplot> plot '*data_file*' ti '*data_label*'（データのラベルを自分で決める。）

ここで紹介したもの以外にも、GNUPLOT には多数のコマンドが存在し、さらにそのコマンドごとに複数のオプションが用意されているため、グラフにより細かい条件を課すことができる。それらの設定方法については、冒頭で挙げた文献に詳しく記述されているので、ぜひとも参考にして頂きたい。