# 2019 年度物性実験 V: 計算機物理

担当: 横山知大、居室: E210、内線: 6504
連絡先: tomohiro.yokoyama@mp.es.osaka-u.ac.jp

## 1 本演習の目的

この演習は、

1. Unix 系 OS の基本操作
2. Fortran による数値計算プログラムの初歩
3. グラフィック ツールや TeX を用いたレポート作成技術

の習得を目的とする。最初の数回 (2, 3 回) で作業の進め方とプログラムの書き方を練習してもらい、いくつかの数値計算アルゴリズム の解説を行なう。それ以降は、各自課題 (別資料) を進めていく。物理的な方程式を数値計算する際には、無次元化について解説したい。

### 1.1 計算機を利用する上での心構え

計算機を使う目的は、計算機を道具として (ここでは物理学に) 役立てるということです。どのような道具 (計算機、ソフトウェア、プログラミング言語など) を使っているかが重要なのではなく、その道具をいかに使いこなして (物理の理解等に) 役立てることができたかが重要なことです。

この科目では、計算物理学の基本的方法の習得のための基礎として、現在物理学の研究環境で一般的と思われる計算機やプログラミング言語などの使い方を学びます。しかし、ここで習った計算機やプログラミング言語が今後もずっと物理学の研究環境で一般的であり続けるとは限りません。また、今後各人が利用できる計算機の環境も、この科目で用いた環境とは異なっているでしょう。急速に進歩し変化していく計算機という道具に柔軟に対応し、利用できる計算機環境に柔軟に適応して、計算機を使う本来の目的を達するように心がけましょう。

### 1.2 プログラミングにおける検算の重要性

手計算 (解析的な計算) でもそうですが、数値計算では検算が非常に重要です。なぜなら、プログラムにミスがあっても何らかの数値が「答え」として返ってきてしまうからです。そして検算がおろそかだと、全く正しくない「答え」が一人歩きして、物理の理解にとって害になることさえあります。数値計算では手計算よりもミスが起きやすく、起き方も異なります。通例、実際 にプログラミングを行うことよりミスを発見して修正することの方がはるかに時間がかかります。

この実習では、自分でプログラムのミスを見つけて直す習慣・能力をつけることも重要な目標です。プログラムが少しでも書けるようになったら、「プログラムを検証・修正することは、それを書いた人の責任である」ということをよく理解してもらいたいと思います。

### 1.3 参考文献

冨田 博之・齋藤 泰洋 「Fortran 90/95 プログラミング」(培風館)

# 2　ローカルな情報

## 2.1　端末と計算機サーバの環境

- 場所: D234
- 端末 PC の OS: windows
- ターミナル: インストール済の「Tera Term」
- 計算機サーバ名: mpjknsrv3
- プライベート IP: 192.168.1.233
- アカウント名: mpjikken
- パスワード: 授業以外では非公開、授業で教えます
- ファイル転送 (SFTP): 「WinSCP」をインストール

## 2.2　計算機サーバへのリモートログイン

リモートログインとは、手元にある端末から離れた端末へネットワーク経由でログインして、様々な操作をすることである。その際に通信の暗号ルールを指定する ssh や、ログインする相手先の住所である IP（グローバルとプライベートの 2 種類）もしくはサーバ名、自分のアカウント名、いくつかのオプションを指定する。リモートログインではターミナル（端末）を利用することが多い。

実際に Tera term からリモートログインしよう。まずは Tera term を立ち上げる。立ち上がったウィンドウに

ホスト (T): 192.168.1.233

と表示されていることを確認する。表示されていない場合はその IP を入力する。OK ボタンを押すとユーザ名とパスフレーズ（パスワードのこと）を入力するウィンドウが立ち上がるので、入力して再び OK ボタンを押す。しばらく待ってターミナル画面に

mpjikken@mpjknsrv3:~$

と表示されたらログイン成功である。

他のターミナルの場合、上記の環境に関する情報から IP やアカウント名などを把握し、

ssh -X “アカウント名”@“プライベート IP”

と入力すればリモートログインできる。もしマシン名がネットワークに登録されている場合は@マーク後を IP ではなくマシン名にしてもログインできる。

## 2.3　作業ディレクトリ

まずはホーム（またはルート）ディレクトリにいることを確認する。コマンドラインに pwd と入力して現在の自分のディレクトリが

/home/mpjikken

であると確認する。また、ls と入力して現在のディレクトリにどの様なファイルと下部ディレクトリが存在するのかを調べる。

次に、cd コマンドを利用してディレクトリ間を移動する。まずは

cd ./jikken2019

と入力して下部ディレクトリの「jikken2019」へ移動。pwd や ls でカレントディレクトリの情報を確認する。同様の手順で、さらに下部ディレクトリの「subclass3」へ移動する。

ここで、自分の名前のディレクトリを作成する。コマンドラインに

mkdir “あなたの名前”

と入力する。ls コマンドで自分の名前のディレクトリが作成されたことを確認する。各々の名前のディレクトリ

を作業ディレクトリとする。**注意！自分の作業は自分の作業ディレクトリ内でのみにすること。また他人の作業ディレクトリを勝手にのぞいたり、編集しないこと。**

### 2.4 ログアウト

exit、または logout と入力。どこのディレクトリで行ってもよい。

## 3 計算機の構造と操作

各々の具体的な作業に入るために、計算機の構造と操作を把握してもらう。

### 3.1 よく使うコマンド

- cd: ディレクトリの変更 (change directory)
- pwd: カレントディレクトリ名の出力 (print working directory)
- ls: ファイル、ディレクトリに関する情報出力 (list)
- cat: ファイル内容の出力 (concatenate)
- mkdir: ディレクトリを作る (make directory)
- rmdir: ファイルの消去 (remove directory)
- vi: エディタ vim の起動とファイル編集
- ifort, gfortran: Fortran のコンパイル
- cp: ファイルの複写 (copy)
- mv: ファイルの移動・名前変更 (move)
- rm: ファイルの削除 (remove)
- man: 書くコマンドのマニュアルを表示 (manual)
- xterm &: 新規ターミナルを起動
- gnuplot: 描画ソフト gnuplot を起動

その他にも色々とコマンドは存在するが、列挙しているときりがないので自分で調べて欲しい。特に job 管理関係のコマンドは後々使うことが多いと思う。しかし、計算機サーバでは**自分だけが利用しているわけではない。** job 管理コマンドは他ユーザの迷惑になりかねないので、注意して使って欲しい。具体的にどの様なコマンドがあるのか、どういったことが出来るのかは質問しても良いし、自分で調べても良い。

コマンドラインで↑、↓を押すと過去のコマンド履歴を辿ってくれる。また、tab キーはファイル名の補完をしてくれるため、小指で押す癖をつけると操作が早くなるだろう。

コマンドラインでの操作は enter キーを押すことで入力される。それまでは誤っても back space キーや delete キーでやり直しができる。

### 3.2 ファイルとディレクトリ

#### 3.2.1 ファイル

様々なデータやプログラムコードを保存したものをファイルという。ファイルはその種類にあわせて拡張子を持つ。Fortran の演習で関係する拡張子をいくつか列挙しておく。

- .f90, .f95: Fortran のプログラムコードのファイル。90, 95 は形式番号
- .out: コンパイル後に生成される実行可能ファイル
- .dat, .d: 計算結果を格納するデータファイル
- .mod: コンパイルによるモジュールファイル。場合によって生成される
- .eps, .ps: gnuplot で描画した図の eps ファイル

### 3.2.2 ディレクトリ

UNIX のファイルは Windows と同様、ツリー状 (枝分かれした状態) で管理されている。ファイルを格納しておく「部屋」をディレクトリという。Windows や MacOS ではフォルダと呼ばれる。各ディレクトリは自分自身と属している「子」、「孫」、... のディレクトリに保管されているファイルを管理している。ディレクトリの下にディレクトリを作ることができるのでツリー状になる。ディレクトリにはいくつか特別なものがある。列挙しておく。

- root ディレクトリ: 一番上に存在するディレクトリ。`/`と表示される。
- current ディレクトリ: 現在作業の対象としているディレクトリ。working ディレクトリとも言う
- home ディレクトリ: ログインした時の current ディレクトリ。home ディレクトリはログインユーザごとに異なるが、本授業では全員で`mpjikken`を共有して利用する。ログイン直後に`pwd`コマンドでディレクトリ名を確認すると、`/home/“ユーザ名”/` となっている。

## 3.3 ファイル・ディレクトリの指定: 絶対パスと相対パス

ファイルを作成・消去したり、その内容を表示、あるいはディレクトリを操作するには、ファイルやディレクトリを指定する必要がある。ディレクトリが異なれば同名のファイル (またはディレクトリ) を置くことが許されているため、それらを一意的に指定する方法が必要となる。その方法には絶対パスと相対パスという 2 つの方法がある。

### 3.3.1 絶対パス

ルートディレクトリからたどってファイルやディレクトリを指定する名前を絶対パス名という。例えば、
`/home/mpjikken/subclass1/yokoyama/test.f95`
という名前で指定することができる。最初の`/`はルートディレクトリを表し、2 番目以降の`/`は “の下の” という意味のディレクトリとディレクトリの区切り、または、ディレクトリとファイルの区切りを表す。絶対パス名は必ず / で始まる。

### 3.3.2 相対パス

カレントディレクトリからたどってファイルやディレクトリを指定する名前を相対パス名という。例えば、ログイン直後は`/home/mpjikken/`にいるので、そこから先ほどと同じファイルを指定する場合は
`subclass1/yokoyama/test.f95`
または
`./subclass1/yokoyama/test.f95`
と指定する。また、自分直上、親ディレクトリを指定する場合は
`../`
とする。自分自身は`./`、姉妹ディレクトリは`../subclass2/`の様に指定する。

### 3.3.3 システム設定によるパス

指定したディレクトリを常に参照する状態を「パス (PATH) が通っている」と表現する。これは UNIX 系に限らずに使う表現で、後々使う LaTeX においても同様の設定がある。この設定にはシステムに変更を加えなければならないが、興味があれば自分で調べて欲しい。

# 4　エディタの操作

エディタとは、ファイルを編集するソフトウェアのことで、普段の PC で利用している word やメモ帳もエディタの 1 つである。Fortran や C++をはじめとした数値計算向けの言語でファイルを記述する際には vi（または vim）や emacs などがよく用いられる。阪大の授業では秀丸も利用されている。この授業では vi での編集を教える。その他には、個人的には windows で Sakura エディタも利用している。word やメモ帳に比べてこれらの利点は編集中の関数などに色を付けて見やすくできる点である。

## 4.1　vim について

vim とは vi エディタ (Visual Interface editor) を上位互換させた高性能エディタで、Vi IMproved（vi の改良）とされている。現在では、vi とコマンド入力することで大抵は vim エディタが起動される。word などのユーザーインターフェイスが良いエディタに慣れていると使いづらく感じるかもしれないが色分けを自動で行ってくれることや、文字コードの互換性のないファイルを作成することがないなど、数値計算のプログラムを書く上でとても便利である。そして書き終わって即座にコンパイルできるため、パラメータをいじくる時に使いやすさを実感してもらえるだろう。

## 4.2　vi の起動とファイルの作成

まずは vi エディタを使ってみよう。自分の作業ディレクトリに移動し、vi test.f95 と入力して enter キーを押す。vi エディタが起動され、test.f95 ファイルが開かれる。ここで、test.f95 ファイルはもともと存在していなくてもよい。ない場合は作成して空のファイルが開かれることになる。

細かい説明は後にして、まずは書き込みと保存を説明する。i キーを 1 度押す。この状態は編集モードになっている。この状態で、

Hello world !!

と打ち込んだら、ESC キーを押して編集モードを終了する。編集モードが終了したら: wq と打ち、enter キーを押すと vi エディタが終了する。ls コマンドと cat コマンドを使ってテキストが保存された test.f95 が作成されていることを確認できると思う。

## 4.3　vi のコマンド

- ←, h: 左へ移動
- ↓, j: 下へ移動
- ↑, k: 上へ移動
- →, l, space: 右へ移動
- i: 編集モードに移行し、カーソルの左側に文字列を挿入 (insert)
- a: 編集モードに移行し、カーソルの右側に文字列を追加 (append)
- o: 編集モードに移行し、カーソルのある行の下に新しい行を作り、文字列を書き込み
- 0: 編集モードに移行し、カーソルのある行の上に新しい行を作り、文字列を書き込み
- ESC: 編集モードを終了し、ノーマルモードに移行
- x: ノーマルモードのまま 1 文字削除
- dd: ノーマルモードのまま 1 行削除
- yy: ノーマルモードのまま 1 行コピー
- p: ノーマルモードのまま 1 行ペースト
- r: ノーマルモードのまま直後に押した文字に書き換え (replace)
- u: 直前の作業を取り消し (undo)
- :wq, ZZ: ノーマルモードで使用。ファイル保存して vi を終了

- :q!: ノーマルモードで使用。ファイル保存せずに vi を終了
- /“文字列”: ノーマルモードで使用。文字列の検索
- :%s/“文字列 A”/“文字列 B”/gc: ノーマルモードで使用。文字列 A を文字列 B に置換

## 4.4 vi のモード切替

vi はテキストをカーソル間の移動や文字列検索などを行うノーマルモードと具体的に編集する編集モード（または挿入モード）がある。他にもモードは存在するが経験上、全く使わないので割愛する。このモード切り替えは word で例えるとマウスでのページ間移動とキーボードでの入力に対応するが、これらをキーボードのみで行うため、初学者が最も混乱しやすい場所である。しかし、これに慣れるとマウスを持つことが煩わしく感じる。是非、体験して欲しい。

### 4.4.1 ノーマルモード

vi を起動した直後の状態。十字キーやマウスのホイールでファイル内を移動したり、コマンドで行ごとのコピー& ペースト、削除を行える。各モードから ESC キーでノーマルモードに戻ることができる。ファイルの保存・終了は常にノーマルモードで行う。

### 4.4.2 編集モード

ファイルの編集を行う。ノーマルモードから i, a キーなどで編集モードへ移行する。編集モードでの入力は基本的に word などと同じ。マウスを使ったコピー& ペーストも可能。

# 5 Fortran のプログラムコード

本演習ではプログラミング言語として Fortran を使用する。Fortran は科学研究向け数値計算に最も適した高級言語である。ここで言う「高級」とは、プログラミング言語のうち、より自然語に近く、人間にとって理解しやすい構文や概念を持った言語を意味する。BASIC や Java も高級言語に属する。逆に低級言語はコンピュータが理解しやすい言語でアセンブリなどがそれに当たる。優劣のことを指す言葉ではない。

Fortran の利点は計算速度と多様なパッケージ、組み込み関数の存在である。その他にも C++ や最近では Python や Ruby なども多くのパッケージ、組み込み関数があり、また mathematica や MathLAB もユーザーインターフェイスが高くよく用いられてる。しかし、どの様な計算が実際には行われているのか、と言う点を理解してプログラムコードを書くために Fortran を勉強してもらう。興味があれば、それぞれの言語について勉強して欲しい。最近では良質な教科書だけではなく、インターネット上で無料の学習コースや動画が公開されている。 1 つのプログラミング言語を習得すれば他を学ぶことは容易にできるはずである。実際、Fortran と C++ は似通っているどころか、パッケージ中で混在させて使用する場合すらある。

## 5.1 Fortran の構造

Fortran のプログラム単位には以下の 4 つがある。

- 主プログラム (メインプログラム、program から end program まで)
- 外部副プログラム (関数もしくはサブルーチンでどこにも属さない (contains されない) もの)
- モジュール
- 初期値設定 (BLOCK DATA)

1 つのプログラムには必ず 1 つの主プログラムからなり、必要に応じて副プログラムを伴う。プログラム単位はファイル分割が許されている最小の単位である。例えば、主プログラムと外部サブルーチン 2 つで構成されるプログラムがあった場合に、すべてを 1 つのファイルに記述することも可能である。

### 5.1.1 主プログラム

プログラムコードの内、program 文から end program 文まで。プログラム名の宣言は冒頭で行う。プログラム名は名前は何でも良いが、空白文字は使えない。Fortran の文には実行文 (write 文など) と非実行文 (program 文、end 文) などがある。宣言部、実行部、副プログラム部の順番で記述される必要があり、実行部に入ると宣言文を書くことはできない。以下に例を示す。␣ はスペース 1 つ分を意味する。

```
PROGRAM main                    !主プログラム名、字下げしないで OK
␣␣␣␣ IMPLICIT NONE
␣␣␣␣ INTEGER :: i              !宣言部
␣␣␣␣ i = 99                    !実行部
␣␣␣␣ call mysub(i)             !内部副プログラムを呼び出している
␣␣␣␣ stop                      !stop 文で計算フローを止める
CONTAINS                        !contains 文で仕切る
␣ SUBROUTINE mysub(a)           !内部副プログラム
␣␣␣␣     integer :: a
␣␣␣␣     write(*,*) a
␣ END SUBROUTINE mysub
END PROGRAM main
```

### 5.1.2 外部副プログラム

外部副プログラムは主プログラムとは別のファイルで用意し、call 文で呼び出して使用する。主プログラム中で同じ計算を複数回行う場合に毎回書くのではなく外部副プログラムを利用するとすっきりとした見やすいコードになる。しかし、他のファイルを呼び出すため、使いするぎると計算速度が遅くなる。

サブルーチンの場合

subroutine 文で始まり、宣言部、実行部、end 文で終わる。例を示す。

```
␣ SUBROUTINE outmysub(x,y,z)    !サブルーチン名
␣␣␣␣ IMPLICIT NONE
␣␣␣␣ REAL :: x,y,z             !宣言部
␣␣␣␣ z = x + y                 !実行部
␣ END SUBROUTINE outmysub
```

外部関数の場合

function 文で始まり、宣言部、実行部、end 文で終わる。例を示す。

```
␣ FUNCTION outmyfunc(a,b,c)     !関数名
␣␣␣␣ IMPLICIT NONE
␣␣␣␣ REAL :: a,b,c             !宣言部
␣␣␣␣ c = a*b                   !実行部
␣ END FUNCTION outmyfunc
```

#### 5.1.3 モジュール

モジュールは変数や定数、配列の宣言をまとめて書いておいたり、変数宣言と一緒にサブルーチンの定義を行ったりできる。サブルーチンと同じく独立したプログラム単位として使用し、主プログラムなどの宣言部にて use 文で使用宣言する。例を示す。

␣ MODULE parameter !モジュール名
␣␣␣␣ IMPLICIT NONE
␣␣␣␣ REAL :: alpha, beta, gamma
␣␣␣␣ REAL, PARAMETER :: pi = acos(-1.0d0)
␣␣␣␣ INTEGER, PARAMETER :: N = 32
␣ END MODULE parameter

## 5.2 Fortran で使う変数、関数、演算、制御文

Fortran では、使用する変数や関数には「型」が存在する。具体的には整数型 (integer)、実数型 (real)、倍精度実数 (double precision)、複素数 (complex)、倍精度複素数 (complex(kind(0d0)))、文字型 (character)、論理型 (logical) である。研究では単精度を用いることはまずない。本演習では倍精度で教える。

制御文とはプログラミングの中で計算の流れを決定する構文である。プログラムの中で用いる do 文、if 文、call 文などである。広義的には write 文や read 文も制御文と言える。「制御文」は他の教科書などで共通して用いたれている言葉ではないが、本演習では教育的に用いる。

#### 5.2.1 変数の宣言

本演習では整数、倍精度実数、倍精度複素数を用いる。文字型と論理型を使いたい人は教科書を参考にして欲しい。Fortran のプログラムコード中で用いたい変数はコードの冒頭でその変数と型を指定しておく必要がある。宣言部の例を以下に示す。

```
IMPLICIT NONE
INTEGER :: i,j,n,m
DOUBLE PRECISION :: x,y,z,E
COMPLEX(KIND(0d0)) :: cx,cy,cz
```

ここで、implicit none は自分の意図しない変数を使える状態にしないための宣言である。マナーとして必ず！書くこと。変数の宣言では、物理的に共通認識のある（常識的な）変数のルールを適用するとよい。例えば、上で挙げたような i, j, n, m は整数、x, y, z は実数変数、E は（無次元化された）エネルギーを指す実数、cx, cy, cz は複素数変数、などである。また、プログラム中で変動しないパラメータとして定義したい場合は、例えば

```
DOUBLE PRECISION, PARAMETER :: pi=dacos(-1.0d0)
```

とする。

倍精度実数（または複素数）であることを指定するために、数字に “d0” を付けている。これがないと単精度とプログラムが認識するので注意して欲しい。この “d” の後の数字は桁を意味する。実数の 100 を指定する変数 x に代入する場合、

```
x=100.0d0
```

としても

```
x=1.0d2
```

としても良い。また、複素数 $e^{i\pi}$ を変数 cx に代入したい場合は

```
ci=dcmplx( 0.0d0, 1.0d0 )
cx=dexp( ci*pi )
```

としても良いし、

```
cx=dcmplx( dcos(pi), dsin(pi) )
```

としても良い。個人的には組み込み関数 dexp( ) が誤作動しないように後者の書き方を推奨する。

### 5.2.2 関数

Fortran は便利な組み込み関数を多数持っている。列挙しきれないので教科書を参照して欲しい。ただし、関数にも整数、単精度、倍精度が存在する。例えば、

```
b=sin(a)
```

は単精度実数 a の入力から単精度実数 b を返す。倍精度は

```
y=dsin(x)
```

という関数で、x, y は倍精度実数である。

以下に Fortran で良く使用する（倍精度）組み込み関数を挙げておく。

- dsin(x): sin 関数の計算
- dcos(x): cos 関数の計算
- dtan(x): tan 関数の計算
- dacos(x): arccos の計算
- dexp(x): 倍精度実数 x の指数を計算、$e^x$。x には複素数も適用可
- dlog(x): 倍精度実数 x の対数を計算
- dsqrt(x): 平方根 $\sqrt{x}$ の計算
- dble(i): 整数 i や単精度実数 a を倍精度実数化。複素数の場合は実部を倍精度実数として取り出す
- dabs(x): 倍精度実数の絶対値を計算
- dcmplx(x,y): 2 つの倍精度実数 x,y を用いて複素数 x + iy を作り出す
- int(x) : （倍精度）実数 x を整数化。小数点以下は切り捨てる
- mod(i,j): 整数 i に対して整数 j で割った際の余りを計算
- dconjg(cx): 倍精度複素数 cx の複素共役を計算
- dimag(cx): 倍精度複素数 cx の虚数部分を取り出し、倍精度実数を作る
- dsign(x), isign(x): 符号を計算。dsign は倍精度実数に対して、isign は整数に対して

### 5.2.3 演算

演算に用いる文字は $=, +, -, *, /, **$ (べき乗) とそれを補う括弧 $(\cdots)$ である。Fortran のプログラムコード中では通常の四則演算をそのまま適用できる。例えば、

```
x=2.5d0
y=x+1.0d0
z=x*y
```

とすれば x, y, z にはそれぞれ 2.5, 3.5, 8.75 が収納されることになる。ここで、“=” は右辺の数値を左辺の変数に代入する演算を意味する。その際、“=” の右辺で用いた変数を左辺に置くこともできる。例えば、上の続きに

```
x=x*x+1.0d1
```

とすると、x には 16.25 が収納される。倍精度実数で例を示したが、整数、倍精度複素数でも同様に計算できる。

演算において、“=” の左右で変数の型が共通している必要がある。注意して欲しいことが、変数の型が合っていなくても **Fortran の最近のコンパイラはエラーを返さずに単精度複素数 > 倍精度複素数 > 単精度実数 > 倍精度実数 > 整数の優先順位で自動的に型を変更してしまう**ことである。変数の型宣言は丁寧にチェックして欲しい。

### 5.2.4 配列

Fortran に限らず、プログラミング言語の重要な機能の 1 つに配列がある。これは 1 つの定義変数が 1 つ、あるいは複数の引数を持つことでベクトルや行列、テンソルとして演算に組み込むことができる機能である。配列の引数の数とその次元は変数の宣言部で定義しておく。具体的には、

```
DOUBLE PRECISION :: a(1:10,0:3)
```

といった記述となる。これは倍精度実数変数 a は 2 つの引数を持ち、その引数は 1 つ目が 1 から１０、２つ目が０から３となる。行列としては $10 \times 4$ と見ることもできるが、０から始まる場合は行列とは捉えずに単に２つの引数を持つ配列と認識した方が良い。当然だが、配列の引数は整数のみである。多くのプログラムでは

```
COMPLEX(KIND(0d0)) :: c(4,4,2)
```

といった宣言をしている。この場合は全て 1 からの引数である。

配列の次元は宣言時点で定義すると述べたが、実際にはプログラム中で定義することもできる。前者を静的配列割付け、後者を動的配列割付けと呼ぶ。どちらにしろ引数は宣言部で定義しなければならない。動的配列割付けの場合、

```
INTEGER, DIMENSION(:), ALLOCATABLE :: m
```

または

```
INTEGER, ALLOCATABLE :: m(:)
```

と宣言する。そして、プログラム中の必要な部分で

```
allocate( m(1:3) )
m(1)=0 ; m(2)=1 ; m(3)=4  !実行部
⋮
deallocate( m )
```

のような使い方をする。これは計算機のメモリを使用する際にできる限りサイズの大きい配列を計算するための工夫である。しかし、２つの欠点がある。１つはメモリの確保と解放の過程で計算速度が遅くなること、もう１つはバグが生じやすいことである。特に奇妙なメモリ確保をした結果、意図しない初期値が格納されている場合である。そうすると、一見何も間違っていないコードなのにバグが発生する。まずは**静的配列割付けで記述することを推奨する。**

### 5.2.5 制御文

制御文はプログラムの要である。人に理解しやすいコードはデバッグが楽にできるが早いとは限らない。特に if 文の使用は便利な反面、計算スピードが遅くなる欠点がある。まずはよく利用する制御文を以下に列挙する。

- do: 指定した回数同じ計算を繰り返す
- if: 条件付けをして計算の選択肢を増やす
- write: 計算結果をディスプレイやデータファイルに書き込む
- read: データファイルやディスプレイから数値を読み込む
- open: 書き込み、読み込み予定のデータファイルを開く
- close: ファイルを閉じる
- stop: プログラムの計算フローを終了する (end 文とは別物)
- call: パッケージやサブルーチンを呼び出して使用する
- use: モジュールを使用すると宣言する

**do 文**

整数変数を 1 つ用いて繰り返し回数を指定する。下の例だと整数変数 i を 1 から１０まで 2 刻みで繰り返す（つまり i=1,3,5,7,9 で計算する）。

```
do i=1,10,2
   write(*,*) i   !実行部
end do
```

繰り返しが 1 刻みの場合は省略可能。実行部は複数行でも良い。**ただし、実行部で繰り返しの整数変数 i を等号 = の左辺に置くなど、内部で値を変えるとエラーが生じる。その状態でコンパイルできてしまうと重大なバグが発生する。処方箋として、do 文に用いる整数変数はあらかじめ決めて置くとよい。**

**if 文**

論理式を用いて条件付けを行う。例を示す。

```
if(論理式 A) then
  x=0.0d0
elseif(論理式 B) then
  x=1.0d0
elseif(論理式 C) then
  x=2.0d0
else
  x=3.0d0
end if
```

例の様に複数回の条件付け、分岐が可能。実行部は複数行でも良い。論理式 B は**自動的に A を満たさない範囲**での条件付けとなる。論理式 C は B、C を満たさない範囲である。また、else は全ての論理式を満たさない条件となる。従って、else は 2 つ以上は作れない。論理式の例を挙げる。

- x.lt.y または x<y
- x.le.y または x<=y
- x.gt.y または x>y
- x.ge.y または x>=y
- x.eq.y または x==y
- x.ne.y または x≠y
- (...).and.(...)
- (...).or.(...)

また、 1 文で完結する単純 if 文もある。この場合、実行部も 1 文のみとなる。例を示す。

```
if( (i.eq.0).and.(j.ne.0) ) write(*,*) x
```

単純 if 文には then は不要である。

**open 文、close 文**

open でファイルを指定し番号付を行う。新規ファイルへの出力や既存のデータファイルからの読み込みに用いる。ファイルが不要になったら close で閉じる。使用例を示す。

```
open(10, file='parameter.dat', status='old')
open(20, file='result.dat', status='new')
⋮
close(10,20)
```

close 文はなくてもプログラムは終了するが、マナーとして書くこと。

**read 文（入力）**

キーボードやプログラム中で指定したファイルから数値を入力する。上の例でファイルを定義している場合は

```
read(10,*) a, b
```

として parameter.dat ファイルから変数 a と b を読み込む。キーボードから入力する場合、番号を 5 または * に指定する。

**write 文（出力）**

ディスプレイやプログラム中で指定したファイルへ数値を出力する。上の例でファイルを定義している場合は

```
write(20,*) x, y
```

として result.dat ファイルに変数 x と y を出力する。ディスプレイに出力する場合、番号を 6 または * に指定

する。

**FORMAT 文**

read 文、write 文において、２つ目の * は数値の形式、整数か実数か、小数点をどこまで記述するか、を指定する。特に read 文を用いる場合は FORMAT 文を用いて記述形式を指定しておくべきである。例を示すと、

```
100 FORMAT(2I,1X,2E12.4)
    write(20,100) i, j, x, y
```

の様に記述する。ただし、FORMAT 文はプログラムのどこに記述してもよい。詳しくは教科書を参照して欲しい。

#### 5.2.6 文関数

ある決まった計算をさせる際にサブルーチンなどの外部服プログラムを call 文で呼び出して使用する。しかし、1 行で記述できる簡単な計算の場合には文関数を用いると便利である。文関数の定義文は宣言文でも実行文でもないので、以下の様に宣言部の後、実行部の前に記述する。

```
PROGRAM example
␣␣␣␣ IMPLICIT NONE
␣␣␣␣ INTEGER :: i,j,N
␣␣␣␣ DOUBLE PRECISION :: x,y,z,f
␣␣␣␣ DOUBLE PRECISION, PARAMETER :: pi=dacos(-1.0d0)
␣␣␣␣ f(x,y) = dcos( pi * dsqrt(x) )*dcos( pi * x ) + dexp( - y*y )
␣␣␣␣ N = 200
␣␣␣␣ open(10,file='example.dat')
␣␣␣␣ do j=0,N
␣␣␣␣     y = dble(j)*1.0d-2
␣␣␣␣     do i=0,N
␣␣␣␣         x = dble(i)*1.0d-2
␣␣␣␣         z = f(x,y)
␣␣␣␣         write(10,*) x,y,z
␣␣␣␣     end do ;! (i)
␣␣␣␣     write(10,*) ""              ;! 空行を入力、おまじないの様なもの
␣␣␣␣ end do ;! (j)
␣␣␣␣ close(10)
␣␣␣␣ stop
END PROGRAM main
```

### 5.3 サンプルコードの作成

練習問題としてサンプルコードを載せる。これまでの説明を基にプログラムコードを書いて欲しい。

**hello.f95** のプログラムコード

```
PROGRAM hello
␣␣␣␣ IMPLICIT NONE
␣␣␣␣ write(*,*) 'Hello World !!'
␣␣␣␣ stop
END PROGRAM hello
```

**add.f95** のプログラムコード

```
PROGRAM add
␣␣␣␣ IMPLICIT NONE
```

```
␣␣␣␣ DOUBLE PRECISION :: x,y
␣␣␣␣ write(*,*) 'First number is'
␣␣␣␣ read(*,*) x
␣␣␣␣ write(*,*) 'Second number is'
␣␣␣␣ read(*,*) y
␣␣␣␣ write(*,*) 'The sum is',x+y
␣␣␣␣ stop
END PROGRAM add
```

### 5.3.1 READ 文、WRITE 文の書式の徹底

学生のプログラムコードを見ると、read 文、 write 文の書き方が非常に汚い。そのために生じるエラーがしばしば見受けられる。以下に、FORMAT 文を用いて丁寧に記述した書き方を示すので、参考にしてほしい。

```
PROGRAM readwrite
    IMPLICIT NONE
    INTEGER :: i,j
    DOUBLE PRECISION :: x,y,tmp1,tmp2,tmp3,tmp4
    COMPLEX(kind(0d0)) :: cx,cy
100 FORMAT(2I,5E12.5)              ! Format 文の番号指定は左詰で
110 FORMAT(1E12.5)
    open(10,file='result.dat',status='new')
    open(20,file='input.dat',status='old')
    ⋮
    read(20,110) x
    do i=1,10
      do j=1,20
    ⋮
        tmp1=dble(cx) ; tmp2=dimag(cx) ; tmp3=dble(cy) ; tmp4=dimag(cy)
        write(10,100) i,j,y,tmp1,tmp2,tmp3,tmp4
    end do ; end do ;!(i,j)        ! どの do 文の終わりなのかを引数で明示しておくとよい
    close(10,20)                   ! 必ずファイルは close 文で終了させる
    stop
END PROGRAM readwrite
```

## 5.4 コンパイル

プログラムコードを完成させると、それをコンパイルすることでそのプログラムの実行可能ファイルが作成される。コマンドラインで

```
ifort xxxxx.f95
```

と入力すると xxxxx.f95 がコンパイルされる。ifot がない場合は

```
gfortran xxxxx.f95
```

とする。コンパイルが成功すれば実行可能ファイル a.out が作成されている。このファイルは

```
./a.out
```

で実行される。また、mv コマンドで名前を変更しても実行可能ファイルのままである。

### 5.5 コメントアウト

Fortran だけではなく C++ などの他のプログラミング言語や LaTeX などの機能としてコメントアウトというものがある。これはコードの途中でコードに無関係な文章を挿入する、または一時的に無効化する機能である。Fortran の場合はコードの各行の左の 1 文字目に「!」を書き加えるとコメントアウトされる。学生の皆さんには是非、コメントアウトを活用して欲しい。特にプログラムコードの説明文を書き加えることで、他人（教授など）に説明する場合や各研究室の資産として残していくときに無用な混乱を避けることができる。これは「正しいマナー」でプログラムを書いていくことと同じぐらい大切である。

### 5.6 デバック

プログラムコードのデバックは最も時間のかかる作業である。何故なら、コンパイルできないだけならエラー部分を調べれば良いが、コンパイルできてしまったが明らかに計算結果がおかしい場合、計算結果がなんとなく正しいように見えるが実際には間違っている場合があるからである。

**デバックの正しいやり方というものはない。**各自で少しずつ自分なりの方法を身に付けていくしかない。デバック作業をやりやすくするためにもコメントアウトや空白を上手に使って見やすいプログラムコードを書いて欲しい。

## 6 グラフィックソフト gunplot

Gnuplot は数字のデータファイルからグラフを画面に描いたり、印刷用のファイルを作成するソフトウェアである。ここでは Gnuplot の基本的な使い方を解説する。グラフィックソフトは他にも多数あるので、自分に合わせて調べて欲しい。

### 6.1 gnuplot のコマンド

- plot, p: 関数やデータファイルを表示する
- replot, rep: 2 本目以降のデータを表示する
- quit, q: gnuplot を終了する
- set xrange [0:10]: x 軸の範囲を 0 から 1 0 に固定する。数字は任意。y 軸も可能
- set xlabel ”文字列”: ”文字列”を軸の名前として表示する。y 軸も可能
- set title ”文字列”: ”文字列”を図の表題として表示する。
- unset key, unset k: プロットしたデータの説明を非表示にする。

やはり機能を説明仕切ることは不可能なので、各自で勉強して欲しい。以下で一連の画像データ作成の流れを説明するが、直感的な試行錯誤で十分に把握できると思う。

### 6.2 gnuplot で関数を描画

例題として、三角関数をプロットして eps ファイルに出力させよう。gnuplot を起動する。

`p sin(x*pi) with lines`

と入力すると別ウィンドウが立ち上がり、sin 関数が表示される。さらに

`rep cos(x*pi) w l lw 2`

と入力すると、先の sin 関数と一緒に cos 関数が新たに表示される。ただし、cos 関数の方が太く表示する様に指定した。デフォルトのデータ表示ではプロットの点数が少ないので

`set samples 200;rep`

などとして細かいプロットに調整する。最後の`rep`は表示の更新を意味する。x 軸が長いので

`set xrange [0:4];rep`

などの様に表示範囲を調節する。

表示した状態を eps ファイルに保存しよう。

set term postscript eps enhanced color 20

と入力すると、gnuplot の入力モードが postscript に変わる。この状態で

set output "sincos.eps";rep;unset output;set term windows

と入力する。細かい説明は省くが、sincos.eps という eps ファイルが作成される。最後の set term windows で再び gnuplot 上で描画できるモードに変更される。gnuplot を他のターミナル上で利用する場合はそのターミナルに応じて set term X11 などとする。

q

を入力して gnuplot を終了し、ls コマンドでファイルが作成されていることを確認する。

gnuplot は windows 上でもインストール不要で利用できる。ダウンロードしてきて windows 上で画像ファイルを作成しても良い。また、数値計算結果を可視化して解析したりバグの有無を検討することもある。3 次元プロットをするとネットワーク間での情報量が多くなり、スムーズな描画ができないことがある。そういった場合にも、計算結果のデータファイルを手元に持ってきて、手元の端末で可視化すると良い。

### 6.3 端末間でのファイルの送信・取得

作成したファイルを手元の端末 (windows) に移動させるには WinSCP を用いる。計算機と端末間でのファイルの送信・取得は WinSCP のような SFTP ソフトを用いても良いし、ターミナル上でコマンドによって行っても良い。大概の SFTP ソフトは直感的なドラッグ&ドロップ操作なので、ソフトがあればそれを使えば良い。しかし、端末によってはターミナルしかない場合もある。

sftp “ユーザ名”@“マシン名か IP”

でログインし、put コマンドと get コマンドさえ使えれば大抵なんとかなる。

## 7 課題

プログラミングの進め方は step by step が基本である。以下では課題を与えているが、まず例題として少々難易度の高いものを分解して少しずつ進めていこう。【例題】入力した日の曜日を出力するプログラムを作成しなさい。

この

【課題 1 】 2 つの関数 $f(x) = 1/(1 + x^n)$ と $g(x) = e^{-x}$ を $x$ に対して数値計算させてどちらが先に小さくなるか示しなさい。

【課題 2 】整数を 1 から任意の N まで和をとるプログラムコードを書きなさい。N はキーボードから入力し、結果をディスプレイに出力させなさい。

【課題 3 】入力した整数 N に対して、偶数なら'even'、奇数なら'odd' と出力するプログラムコードを書きなさい。組み込み関数の mod 関数を使うと便利だが、使わない方法を探してみても良い。

【課題 4 】入力した 3 つの整数を 3 次元ベクトルの要素として、配列を利用してベクトルの長さと単位ベクトルを計算するプログラムを書きなさい。また、余力があれば計算された単位ベクトルに垂直な 2 つのベクトルを計算するプログラムが書きなさい。

# 8　無次元化

物理的な問題を数値的に計算して解を求める際に必ず行うべき処方が無次元化である。これは何らかの関数を冪的に展開して近似式から計算するときに、冪の次数ごとに物理の次元が異なる、などという間違いを起こしてはいけないことと同じである。また、調べている物理現象がどのようなスケールの世界で起きていることなのかを理解する手助けにもなるし、一見異なる物理現象との共通した発現機構を理解することにもつながる。また、数値計算のパラメータを掃引する際にどのような範囲で行えば良いかの指針を与える。

## 8.1　例 1：運動方程式の無次元化

バネにつながれた質量 $m$ の質点に正弦波で外力 $A\sin(\omega t)$ が加わっているとする。運動方程式は

$$m\frac{d^2}{dt^2}x(t) = -kx(t) + A\sin(\omega t) \tag{1}$$

で与えられる。ここで、各文字の次元を確認しよう。$m$ は質点の質量なので [M]、$t$ は時刻なので [T]、$x$ は変位なので [L] である。両辺は力の次元 [$\mathrm{M^1\ L^1\ T^{-2}}$] で記述されているので、ばね定数 $k$ の次元は [$\mathrm{M^1\ T^{-2}}$]、力の振幅 $A$ は [$\mathrm{M^1\ L^1\ T^{-2}}$] である。また、sin 関数の中身である $\omega t$ についても、角振動数である $\omega$ は当然 [$\mathrm{T^{-1}}$] と考えてもよいし、関数の中身は全体で無次元なので時間の逆数の次元 [$\mathrm{T^{-1}}$] となると理解してもよい。

この運動方程式を無次元化する。その際に、物理現象に対して特徴的なスケールを決める必要がある。質量については $m$ しかないので $m_0 = m$ をスケールとすれば十分である。質量の異なる複数の質点を考える場合はその平均質量を特徴的なスケールとするのが妥当である。時間については、外力の振動周期が $T_0 = 2\pi/\omega$ で $\omega$ は問題において定数なので、これをスケールにしよう。一方、長さについては何が特徴的なスケールになるかがまだ明確ではないので、一旦置いておく。運動方程式 (1) の左辺に着目すると、$m$ と $t$ が裸で現れている。両辺に $T_0^2/m_0$ をかけると、

$$\left(\frac{m}{m_0}\right)\frac{d^2}{d(t/T_0)^2}x(t) = -\frac{T_0^2 k}{m_0}x(t) + \frac{T_0^2 A}{m_0}\sin\left(2\pi\frac{t}{T_0}\right) \tag{2}$$

ついでに sin 関数の中身を $T_0$ で書き換えている。ここで、右辺第 2 項を見ると $T_0^2 A/m_0$ が次元 [L] だと分かる。これを長さのパラメータ $L_0 \equiv T_0^2 A/m_0$ として、式 (2) の両辺を $L_0$ で割る。さらに、無次元化したパラメータ $\tilde{x} = x/L_0$、$\tilde{t} = t/T_0$ を用いると

$$\frac{d^2}{d\tilde{t}^2}\tilde{x}(\tilde{t}) = -\tilde{k}\tilde{x}(\tilde{t}) + \sin\left(2\pi\tilde{t}\right) \tag{3}$$

と無次元化した運動方程式が求まる。ただし、変位 $x$ が時刻 $t$ の関数から無次元パラメータ $\tilde{t}$ の関数になっている。また、ばね定数も $\tilde{k} \equiv T_0^2 k/m_0$ で無次元化されている。

無次元化を行うことで変数の範囲を 1 程度にできる。ただ、無次元化は一意的ではなく、例の通りでなくともよい。例えば、時間スケールの取り方を $T_0 = 1/\omega$ とすれば、sin 関数の中身に $2\pi$ が現れない。個人的には、例の様に無次元化した方が時間の周期を $\tilde{t} = 1$ とできるため、時間に対するシミュレーションの刻み幅を、例えば $\Delta\tilde{t} = 0.01$ とする、などのシンプルで美しい形にできる方が好ましいと考える。

## 8.2　例 2：調和振動子の比熱

ハミルトニアンが

$$H = \frac{p^2}{2m} + \frac{1}{2}m\omega^2 x^2 \tag{4}$$

で与えられる $N$ 粒子系の分配関数は

$$\begin{aligned} Z_N &= \frac{1}{h^N}\int_{-\infty}^{\infty} d^N p_\nu \int_{-\infty}^{\infty} d^N x_\nu \exp\left(-\frac{H(x_\nu, p_\nu)}{k_\mathrm{B}T}\right) \\ &= \frac{1}{h^N}\prod_\nu \int_{-\infty}^{\infty} dp \exp\left(-\frac{\beta}{2m}p^2\right) \times \int_{-\infty}^{\infty} dx \exp\left(-\beta a x^2\right) \end{aligned} \tag{5}$$

である（$a = m\omega^2/2$）。それぞれの積分部分について $\tilde{p}^2 = \beta p^2/2m$、$\tilde{x}^2 = \beta a x^2$

$$Z_N = \left[\frac{1}{h}\left(\sqrt{\frac{2m}{\beta}}\right)\int_{-\infty}^{\infty} d\tilde{p}\exp\left(-\tilde{p}^2\right) \times \left(\frac{1}{\sqrt{\beta a}}\right)\int_{-\infty}^{\infty} d\tilde{x}\exp\left(-\tilde{x}^2\right)\right]^N \tag{6}$$

として、積分を無次元のパラメータ $\tilde{x}, \tilde{p}$ のみにする。

ちなみにこの場合は数値計算せずともガウス積分を実行できる。

$$Z_N = \left[\frac{1}{h}\left(\sqrt{\frac{2m\pi}{\beta}}\right)\left(\frac{\pi}{\sqrt{\beta a}}\right)\right]^N = \left(\frac{2\pi}{h\beta\omega}\right)^N = \left(\frac{k_{\mathrm{B}}T}{\hbar\omega}\right)^N \tag{7}$$

調和振動子からずれる場合はどうなるか、どう無次元化すれば良いかを考えてみること。

比熱を求めよう。自由エネルギー $F$、エントロピー $S$、内部エネルギー $U$ はそれぞれ

$$F = -k_{\mathrm{B}}T\log(Z_N) \tag{8}$$

$$S = -\frac{\partial F}{\partial T} = k_{\mathrm{B}}\log(Z_N) + k_{\mathrm{B}}T\frac{\partial \log(Z_N)}{\partial T} \tag{9}$$

$$U = F + TS = k_{\mathrm{B}}T^2\frac{1}{Z_N}\frac{\partial Z_N}{\partial T} \tag{10}$$

内部エネルギーの $C = \partial U/\partial T$ なので、(5) 式の分配関数から

$$C = 2k_{\mathrm{B}}T\frac{Z_N'}{Z_N} + k_{\mathrm{B}}T^2\frac{Z_N'' Z_N - {Z_N'}^2}{{Z_N}^2} \tag{11}$$

となる。ここで、

$$Z_N' = \frac{\partial Z_N}{\partial T} = \left[\frac{1}{h}\int_{-\infty}^{\infty} dp\int_{-\infty}^{\infty} dx\left(\frac{H(x,p)}{k_{\mathrm{B}}T^2}\right)\exp\left(-\frac{H(x,p)}{k_{\mathrm{B}}T}\right)\right]^N \tag{12}$$

$$Z_N'' = \frac{\partial^2 Z_N}{\partial T^2} = \left[\frac{1}{h}\int_{-\infty}^{\infty} dp\int_{-\infty}^{\infty} dx\left\{\left(\frac{H(x,p)}{k_{\mathrm{B}}T^2}\right)^2 - \frac{2H(x,p)}{k_{\mathrm{B}}T^3}\right\}\exp\left(-\frac{H(x,p)}{k_{\mathrm{B}}T}\right)\right]^N \tag{13}$$

これらの積分を数値計算で計算すれば良い。解析的に求められる調和振動子の場合などは直接 $Z_N(T)$ を微分すれば良いが、そうでない非調和振動子 $V = ax^2 + bx^4$ の場合、(5) 式中の $p$ については先に積分して、$x$ については数値計算すれば良い。その際には $\tilde{x}$ で無次元化する。

# 9　方程式の解法

これまで、数学や物理学で多くの方程式を解いてきたと思う。高校数学の問題は、必ず解が存在し、技巧的な方法で解けた。しかし、実際の問題では解析的に（手計算で）解を求めることができることはむしろ例外で、多くの場合は数値計算に頼ることになる。例えば、次の解を見つける問題は解析的には解けない)。

$$-\exp\left[-(x-1)^2\right] + \log x + \sqrt{x} = 0 \tag{14}$$

これは関数 $f(x) = -\exp\left[-(x-1)^2\right] + \log x + \sqrt{x}$ に対して、$f(x) = 0$ の方程式の解を求める問題と同じである。この場合は偶然に $x = 1$ という解が分かっている。試しにこの関数を gunplot で表示して見てほしい。

以下では一般的な方程式 $f(x) = 0$ の解法のうちニュートン法と二分法を説明する。

## 9.1　ニュートン法

方程式 $f(x) = 0$ の解を数値計算によって求める方法の一つ。探索点近傍を 1 次式で近似して次の探索点へ移動する手法である。1 元方程式の場合について説明する。まず適当に、真の値に近いと思われる値 $x_0$ を一つ決める。あとは、漸化式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{15}$$

に従って $x_{n+1}$ を決める。これは関数 $f(x)$ の符号変化によって漸近的な $x_{n+1}$ 決定を変化させる。適当な数 $\epsilon$ を設定し、$|x_{n+1} - x_n| < \epsilon$ を満たすまでこのサイクルを繰り返す事で、$\epsilon$ に応じた精度で近似解が求まる。手順にすると以下になる。

1. 探索点 $x_0$ を定め、探索点における接線の傾き $f'(x_0)$ を求める。
2. $\alpha_n = f'(x_n)$、$\beta_n = f(x_n)$ とすると、接線は $\alpha_n(x - x_n) + \beta_n$ である。
3. この接線と $x$ 軸の交点を次の探索点 $x_{n+1} = x_n - (\beta_n/\alpha_n)$ とする。
4. 探索点の距離が $|x_{n+1} - x_n| < \epsilon$ を満たすまで 2. と 3. の過程を繰り返す。($\epsilon$ は計算精度を決める定数)

例えば、(14) 式の場合の関数 $f(x)$ は $x$ の増加に対して単調増加なので、1 階微分は $f'(x) > 0$ である。一方、$f(x)$ は解となる $x = 1$ で（当然だが）符号を変える。従って、$x = 1$ の辺りで数列 $\{x_n\}$ は行きつ戻りつしながら $x = 1$ に近づいていく。

なお、この方法では数列 $\{x_n\}$ の解への収束性は必ずしも保証されないので注意が必要である。また、微分がゼロになる場合は (15) 式の右辺第 2 項が発散するため、関数の極大・極小点近傍に解がある場合は収束性が悪くなる。収束する場合、2 次収束で解に収束する。また、ニュートン法の場合、調べる関数の探索点における微分 $f'(x)$ を知る必要がある。もしも解析的には導関数 $f'(x)$ がもとまらない場合は、ある小さな値 $\Delta x(< \epsilon)$ に対して

$$f'(x_n) \simeq \frac{f(x_n + \Delta x) - f(x_n)}{\Delta x} \tag{16}$$

と近似する過程を手順に組み込む必要がある。

以下にニュートン法を用いたプログラムを載せるので参考にしてほしい。

```
 REAL(8) FUNCTION f(x)
     IMPLICIT NONE
     DOUBLE PRECISION :: x
     f = -dexp( -(x-1.0d0)*(x-1.0d0) ) + dlog(x) + dsqrt(x)
     return
 END FUNCTION f
!==============================
 REAL(8) FUNCTION g(x)
     IMPLICIT NONE
     DOUBLE PRECISION :: x
     g = 2.0d0*(x-1.0d0)*dexp( -(x-1.0d0)*(x-1.0d0) ) + 1.0d0/x + 0.5d0/dsqrt(x)
     return
 END FUNCTION g
!==============================
 PROGRAM NewtonMethod
     IMPLICIT NONE
     INTEGER :: i,icheck
     DOUBLE PRECISION :: x,f,g,alpha,beta,tmp1,tmp2
     DOUBLE PRECISION, PARAMETER :: eps = 1.0d-4
     x=2.0d0 ; icheck=1
     do i=0,1000
        beta=f(x) ; alpha=g(x)
        tmp1=x-beta/alpha
        if( icheck.ne.0 ) then

        if( dabs(tmp1-x).gt.eps ) then
```

```
            x=tmp1
         else
            tmp2=f(tmp1)
            x=x-beta*(tmp1-x)/(tmp2-beta)
            tmp2=f(x)
            write(*,*) x,tmp2
            icheck=0
         end if ;! (eps)

         end if ;! (icheck)
      end do ;! (i)
      if( icheck.eq.1 ) then
         write(*,*) 'not found within 1000 iteration'
      end if ;! (icheck)
      stop
! CONTAINS
END PROGRAM NewtonMethod
```

### 9.2 二分法

二分法はニュー トン法より収束は遅いが、方程式が連続で以下で述べる初期値を与える事ができれば必ず収束するという特徴を持つ。手順は以下の通りである。

1. $f(x=a)$ と $f(x=b)$ とで符号が異なるような $a$ と $b$ を定める。
2. $a$ と $b$ の中点 $c$ を定める。
3. $f(c)$ の符号が $f(a)$ と等しければ $a$ を $c$ で置き換える。逆に $f(b)$ と等しければ $b$ を $c$ で置き換える。
4. 新しい $a$ または $b$ に対して $|b-a| < \epsilon$ を満たすまで 2. と 3. の過程を繰り返す。($\epsilon$ は計算精度を決める定数)

つまり、少しずつ判定する範囲を絞っていく手法である。

このように、ある区間 $(a,b)$ に解が1つだけ存在することが分かっている場合に、上のアルゴリズムで確実に（近似）解を求めることが出来る。また、1 階微分が分かっていない数値的な関数の場合にも適用できる手法である。ただし、非常に近い範囲に解が複数存在するかもしれない場合、例えば縮退に対する摂動論などの問題では、全ての解をちゃんと見つけられるかを慎重に検討すべきである。

## 10 微分方程式の解法

物理学において、微分方程式を解くことは日常茶飯事である。しかし、微分方程式は少し複雑になっただけで途端に計算難度が上がり、厳密解を求めることができなくなる。そのため、コンピュータによる数値計算において微分は重要な課題である。にも関わらず、数値微分は計算精度が非常に悪いことが知られている。皆さんには以下に示す数値微分の手法を身につけ、解析解と比較することでその現状を楽しんでもらいたい。

### 10.1 単純な差分近似とオイラー法

非常にシンプルな数値微分の手法。シンプルすぎるので精度が低いが、簡単なのでラフに微分の振る舞いを調べるときに役に立つ。

数値計算では無限小を扱うことができないので、小さいが有限の値 $h$ を用いて

$$f'(x) \simeq \frac{f(x+h)-f(x)}{h} \tag{17}$$

と差分近似を適用する。本質的にはこれだけである。

物理で微分を取り扱う代表的な例は運動方程式である。つまり、時間に対する常微分方程式を数値的に解くことが必要になる。それに差分法を適用したものがオイラー (Euler) 法である。1 次元で 1 変数の 1 階常微分方程式

$$\dot{x}(t) = \frac{dx(t)}{dt} = f(t, x(t)) \tag{18}$$

を考えよう。運動方程式が意味することを言葉にすると「時刻 $t$、位置 $x$ における変化率 (速度) は現在の時刻 $t$ と位置 $x$ によって定まる」ということである。初期条件を $x(t=0) = x_0$ とする。数値的には解く場合、時間軸を離散化することになる。整数 $j$ を用いて

$$t_j \equiv jh \tag{19}$$

とする。その際の座標を $x_j \equiv x(t = t_j)$ と表記する。ここで注意してもらいたいことは、$h$ は物理的な運動に対して小さな値である、ということと、「小さい」という表現から当然だが、時刻 $t$ は無次元化された時間である、という 2 点である。

$x(t+h)$ をテイラー展開しよう。

$$\begin{aligned} x(t+h) &= x(t) + h\dot{x}(t) + O(h^2) \\ &= x(t) + hf(t, x(t)) + O(h^2) \end{aligned} \tag{20}$$

ここで、右辺第 2 項に運動方程式 (18) を用いている。(20) 式において $h$ の高次項 $O(h^2)$ を無視する。時刻が $t = t_j = jh$ の時に

$$x_{j+1} \simeq x_j + hf(t_j, x_j) \tag{21}$$

この式の解釈を運動方程式の解釈と対比させて述べると「時間に対するステップ毎に位置 $x_j$ は 1 ステップ前の位置から $hf(t_j, x_j)$ ずつ変化していく」と言える。数値計算では計算機に逐次 $f(t_j, x_j)$ を計算させ、これまでの位置 $x_j$ に加えていけば良い。

上記の仮定から、オイラー法では数値計算の 1 ステップ毎に $O(h^2)$ の誤差が生じる。ある時刻 $T = Nh$ における座標 $x_N$ をオイラー法で求めるためには (21) 式を $N$ 回用いる。従って、$t = 0$ から出発して $T = Nh$ における座標 $x_N$ を求めたときには、$h^2 \times N = Th$ に比例した誤差が生じる。1 ステップ毎の誤差を $O(h^3)$ に抑える修正オイラー法もある。これは

$$\begin{aligned} x_{j+1} &= x_j + k_1 \\ k_0 &= hf(t_j, x_j) \\ k_1 &= hf(t_j + h/2, x_j + k_0/2) \end{aligned}$$

とする。つまり「オイラー法で半ステップだけ進んで、そこの $\dot{x}$ で元の位置から 1 ステップ進む」というものである。

## 10.2 ルンゲ・クッタ法

微分方程式の初期値問題に対して近似解を与える制度の高い手法である。つまり、ルンゲ･クッタ (Runge-Kutta) 法では、2 次以降の項を考慮した補正項を導入して近似精度を上げる。最もよく知られた方法は、$h$ の 4 次の項まで一致させる方法である（つまり 5 次の誤差を含む)。$t_j, x_j$ を用いて、$t_{j+1} = t_j + h$ における $x_{j+1}$ を

$$x_{j+1} = x_j + \frac{k_0 + 2k_1 + 2k_2 + k_3}{6} \tag{22}$$

ただし、右辺第 2 項の各 $k_j$ は

$$k_0 \equiv hf(t_j, x_j) \tag{23}$$

$$k_1 \equiv hf(t_j + h/2, x_j + k_0/2) \tag{24}$$

$$k_2 \equiv hf(t_j + h/2, x_j + k_1/2) \tag{25}$$

$$k_3 \equiv hf(t_j + h, x_j + k_2) \tag{26}$$

ルンゲ・クッタ法の導出は少々時間がかかるので、とりあえず使えるという程度で十分である。詳細については各自で勉強して欲しい。この程度の修正でオイラー法より格段に精度が上がることは特筆すべきことであり、実際の研究ではオイラー法を使わない理由でもある。

課題として、$f(t,x) = x$ の場合の厳密解、オイラー法、ルンゲ・クッタ法の比較をしてみてほしい。その際に、時刻のステップ幅（刻み幅）を変えた場合を比較する図を作成してほしい。

以下にルンゲ・クッタ法のプログラム例を載せるので参考にしてほしい。

```
PROGRAM RungeKutta
    IMPLICIT NONE
    INTEGER :: i,N
    DOUBLE PRECISION :: x,t,f,k0,k1,k2,k3,tmp1
    DOUBLE PRECISION, PARAMETER :: pi=dacos(-1.0d0)

    f(t,x) = dsin( pi*t ) + ( x*x )*dexp( -x*x )

    open(10,file='RK.dat')
    N = 1000 ;! number of steps for integral
    h = 1.0d-2 ;! distance of steps
    x = 0.0d0 ;! initial position
    do i=0,N
       t = h*dble(i)
       k0 = h*f( t             , x             )
       k1 = h*f( t + 0.5d0*h , x + 0.5d0*k0 )
       k2 = h*f( t + 0.5d0*h , x + 0.5d0*k1 )
       k3 = h*f( t + h         , x + k2        )

       tmp1 = (k0 + 2.0d0*k1 + 2.0d0*k2 + k3)/6.0d0
       x = x + tmp1

       write(10,*) t,x
    end do ;! (i)

    close(10)
    stop
END PROGRAM RungeKutta
```

## 10.3 連立微分方程式

前節では変数は 1 つだけであり、 1 階の微分方程式の場合を述べた。しかし、実際の問題としては、多変数・高次階の連立微分方程式を扱うことの方がはるかに多い。以下では多変数・高次階の連立微分方程式を扱うレシピを述べる。

まず、高次の微分についてだた、具体例としてばねによる質点の振動の運動方程式を考える。（無次元化された）運動方程式は (3) で与えられる。簡単のために表記を簡単にして

$$\ddot{x}(t) + kx(t) = \sin(2\pi t) \tag{27}$$

ここで、座標 $x(t)$ を $x_1(t)$ に変更し、新しい変数 $x_2(t) = \dot{x}_1(t)$ を導入する。すると運動方程式は

$$\dot{x}_1 = x_2(t) \tag{28}$$
$$\dot{x}_2 = -kx_1(t) + \sin(2\pi t) \tag{29}$$

と 1 階微分のみの連立微分方程式に帰着される。3 階以上の微分に対しても同様に新しい変数を導入して 1 階の連立微分方程式に帰着させることができる。ここで重要な点は**左辺に変数の 1 階微分のみ、右辺はその時刻の変数で決まる関数のみとなっていることである。**

次に、多変数の場合を考える。簡単のために 2 変数 $x, y$ の連立微分方程式とする。

$$\dot{x} = f(t, x, y) \tag{30}$$
$$\dot{y} = g(t, x, y) \tag{31}$$

これに対して、ルンゲ・クッタ法は

$$x_{j+1} = x_j + \frac{k_0 + 2k_1 + 2k_2 + k_3}{6} \tag{32}$$
$$y_{j+1} = y_j + \frac{l_0 + 2l_1 + 2l_2 + l_3}{6} \tag{33}$$

と拡張される。ただし、

$$k_0 = hf(t_j, x_j, y_j) \tag{34}$$
$$l_0 = hg(t_j, x_j, y_j) \tag{35}$$
$$k_1 = hf(t_j + h/2, x_j + k_0/2, y_j + l_0/2) \tag{36}$$
$$l_1 = hg(t_j + h/2, x_j + k_0/2, y_j + l_0/2) \tag{37}$$
$$k_2 = hf(t_j + h/2, x_j + k_1/2, y_j + l_1/2) \tag{38}$$
$$l_2 = hg(t_j + h/2, x_j + k_1/2, y_j + l_1/2) \tag{39}$$
$$k_3 = hf(t_j + h, x_j + k_2, y_j + l_2) \tag{40}$$
$$l_3 = hg(t_j + h, x_j + k_2, y_j + l_2) \tag{41}$$

ここで注意してもらいたいことは $k_{0,1,2,3}$ と $l_{0,1,2,3}$ を計算させる順番である。$k_{1,2,3}$ と $l_{1,2,3}$ にはそれぞれ $k_{0,1,2}$、$l_{0,1,2}$ が必要なので、先に添え字が若いものから順番に数値計算する必要がある。つまり、$k_{0,1,2,3}$ を先に計算してから $l_{0,1,2,3}$ を求めるという手続きは間違いである。先に述べた様に、高次の微分は新しい変数の導入で連立微分方程式に帰着されるので、最終的には 1 階の多変数連立微分方程式をルンゲ・クッタ法で解けば良い。また、線形方程式に帰着された場合は行列・ベクトルにまとめると良い。

# 11 数値積分

数値計算における定積分は言ってしまえばただの足し算である。高校の数学で積分が棒グラフの足し算の様になっていたと思うが、それを数値的に実行するだけである。ただし、その際に誤差が小さくなる工夫を行う。最も単純な手法は今述べた長方形の和をとる手法で、関数 $f(x)$ を $x = a$ から $b$ まで積分するのに、区間 $[a, b]$ を $N$ 等分し、$N$ 個の長方形の面積の和 $S$ で近似する。

$$S = \Delta \sum_{j=0}^{N-1} f(x_j) \tag{42}$$

ここで、$\Delta = (b - a)/N$ は積分の刻み幅である。以下では、この最も単純な近似から少しずつ精度の高い手法へ進んで行こう。

### 11.1 台形公式

数値積分の手法の 1 つで、積分範囲を等間隔に刻んで関数を台形に近似する。1 次関数による近似なので、次にあげるシンプソン法よりも精度が低いが、激しく変動しない関数ならば十分な精度を与える。

長方形での近似と同様に、刻み幅を $\Delta = (b-a)/N$、その位置を $x_j = a + j\Delta$ とすると、

$$S = \frac{\Delta}{2}\sum_{j=0}^{N-1}(f(x_j) + f(x_{j+1})) = \Delta\left[\frac{f(x_0)+f(x_N)}{2} + \sum_{j=1}^{N-1} f(x_j)\right] \tag{43}$$

となる。(42) 式と比較すると、ほとんど計算的な手間は変わらないことがわかる。にも関わらず、台形公式は精度を高めてくれる。これは実際に積分結果を比較して確かめてほしい。

### 11.2 シンプソン公式

Thomas Simpson によって開発された積分の近似値を求める手法である。被積分関数を 2 次関数で近似するため、台形公式より精度が高い。

シンプソン公式では $[a,b]$ の積分区間を $2N$ 等分する。偶数で等分する点が重要である。刻み幅は $\Delta = (b-a)/2N$、その位置は $x_j = a + j\Delta$ となる。数値積分を実行するにあたって得ているデータは $j = 0 \sim 2N$ に対する $(x_j, y_j \equiv f(x_j))$ の組である。そこから 3 つの連続したデータ点 $(x_{2k}, y_{2k})$、$(x_{2k+1}, y_{2k+1})$、$(x_{2k+2}, y_{2k+2})$ に対して、区間 $[x_{2k}, x_{2k+2}]$ での被積分関数を 2 次関数 $g(x) = a(x-x_{2k+1})^2 + b(x-x_{2k+1}) + c$ で近似する。この微小区間での積分を $\Delta S_k$ とすると、

$$\begin{aligned}\Delta S_k &\simeq \int_{x_{2k}}^{x_{2k+2}} dx g(x) = \int_{-\Delta}^{\Delta} dt(at^2+bt+c) = \left[\frac{a}{3}t^3 + \frac{b}{2}t^2 + ct\right]_{-\Delta}^{\Delta} \\ &= \frac{2}{3}a\Delta^3 + 2c\Delta = \frac{\Delta}{3}(2a\Delta^2 + 6c)\end{aligned} \tag{44}$$

とできる。途中で積分変数を $t = x - x_{2k+1}$ に変数変換を行い、積分の範囲を $t = -\Delta \sim \Delta$ に取り直した。一方、近似に用いた 2 次関数の係数 $a, b, c$ は以下の関係式を満たす。

$$y_{2k} = g(x_{2k}) = a\Delta^2 - b\Delta + c \tag{45}$$

$$y_{2k+1} = g(x_{2k+1}) = c \tag{46}$$

$$y_{2k+2} = g(x_{2k+2}) = a\Delta^2 + b\Delta + c \tag{47}$$

3 式を組み合わせれば、先ほどの積分の最右辺に対して $(2a\Delta^2 + 6c) = y_{2k} + 4y_{2k+1} + y_{2k+2}$ となることが分かる。つまり、

$$\Delta S_k \simeq \frac{\Delta}{3}(y_{2k} + 4y_{2k+1} + y_{2k+2}) = \frac{\Delta}{3}\left(f(x_{2k}) + 4f(x_{2k+1}) + f(x_{2k+2})\right) \tag{48}$$

この $\Delta S_k$ を $k = 0$ から $N-1$ まで足しあげると、

$$S = \frac{\Delta}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{2N-1}) + f(x_{2N})\right) \tag{49}$$

というシンプソン公式が求まる。

実はこのシンプソン公式はルンゲ・クッタ法と関係している。興味がある人は調べてみてほしい。

## 12 ランダムな数値の利用

擬似的な乱数を発生させると、ノイズや乱流、ブラウン運動など、実験では予測しきれない物理現象を数値的に考察することができる。ただし、注意しなければならないことはそれが「どの程度の乱数なのか？」である。例えば、ランダムな数値を使って銀河に分布する星の位置を与え、シミュレーションを行いたいとする。しかし、適切でない乱数を用いてしまうと本来存在しない縞状の分布が現れることもある。一方で、数値的な「擬似」乱数だからこその利点もある。つまり、「毎回同じ乱数」であることを使ってバグチェックや系統的な振る舞いを調べることができる。

## 12.1 擬似乱数の発生

擬似乱数の発生には組み込み関数を用いることが一般的である。その際、最初に乱数発生の種（整数）を与える。この種を使い続けることで「同じ乱数」が得られる。また、意図的に現在時刻を乱数の種に指定することで絶対に同じ乱数が発生しない様にすることも可能である。

まずは do ループで擬似乱数を発生させ、write 文でデータファイルに数列として出力しよう。また、どの様なランダム数列が発生したのかを視覚的に確認するために gnuplot で描画しよう。サンプルコードを示す。

```
PROGRAM Random
    IMPLICIT NONE
    INTEGER :: i,iseed
    DOUBLE PRECISION :: x,f,S,k0,k1,k2,k3,tmp1,tmp2
    INTEGER, PARAMETER :: N = 100
    DOUBLE PRECISION, PARAMETER :: h = 1.0d-2
    open(10,file='random.dat')
    iseed=19                    ;! put odd integer to generate random numerics
    do i=0,N
       call rand()
       x(i)=rand
       write(10,*) i,x(i)
    end do ;! (i)
    close(10)
    stop
END PROGRAM Random
```

同じ計算機を使用する限り、同じ乱数の種 (iseed) に対して同じ擬似乱数を得ることができる。これはデメリットの様に感じるが、実際にはメリットである。というのも乱数を使いながらバグチェックを行う際に、「完全に同じ」乱数を使うことでどこにバグが存在するのかの判定や高速化を施した際にバグが生じていないか、といったチェックに用いることが出来るからである。ただし、注意して欲しいことはこれが「擬似」乱数という点である。例えば、銀河（の腕や中心部）に散らばる恒星の分布をランダムなものと仮定して擬似乱数を用いて描画させると、縞状の銀河が出来上がった、という「バグ」が生じたりする。

## 12.2 モンテカルロ法による $\pi$ の計算

モンテカルロとはモナコ公国の 1 行政区画の名前で、世界的に有名なカジノ区画である。ランダムな数列を利用して正解を導く手法から、Ulam が中性子の運動を記述するために開発し、von Neumann によって命名されたとされている。

最近の物性研究、特に強相関系で量子モンテカルロシミュレーションが重要な研究手法とされている。筆者は詳しくないので他の先生に聞いてほしい。

$[0,1)$ 上の一様乱数を 2 つ作り $x, y$ とする。この $(x_j, y_j)$ を正方形 $[0,1)\times[0,1)$ 内の 1 点に対応させる。$x^2+y^22 \leq 1$ ならばその点は 4 分円の内部に入るが、その確率は $\pi/4$ である。多数回このことを反復して 4 分円の中に点が位置する比率を計算し、それを 4 倍すれば $\pi$ の近似値が求められる。10,000 回程度の試行を 行い、1,000 回ごとにそれまでに得られた近似値を出力しなさい。さらに、ランダムな点の座標データを 4 分円の内部のものと外部のものに分けて別の 2 つのファイルに書き出し、Gnuplot を用いて図示しなさい。

# 13 パッケージの利用

Fortran には様々な計算パッケージ（ライブラリと呼ばれる）が存在する。最も有名なのが以下でサンプルコードを示す LAPACK だが、他にも様々なパッケージが存在する。

### 13.1 LAPACK

物理学の問題、特に量子力学の問題は大抵が行列で記述される。(量子力学の前身は行列力学) そして行列計算といえば LAPACK である。本演習では深入りしないが、固有値と固有ベクトルを求める例題コードを載せておく。インターネットで検索すれば当該パッケージのマニュアルも手に入る。是非、自分で調べて身に付ける、というプロセスを行って欲しい。

## 14 数値誤差について

Fortran において、実数を用いたときに生じる誤差について紹介する。

### 14.1 丸め誤差

仮数部が有限桁であるために、計算機内の実数値は切捨てられ、あるいは丸められる。これによって生じる誤差を丸め誤差といいます。上記の例では、仮数部は 2 進 5 3 桁なので、切捨てを行なったときに生じる相対誤差の上限は $2^{-52}$ となる。この値は浮動小数点の表し方で決る値で、計算機イプシロンと呼ばれる。

### 14.2 情報落ち

大きさの異なる 2 つの実数の加算を行なう場合、小さい方の指数部を大きい方の指数部に合わせて仮数部の桁をずらしてから加算が実行される。これによって、小さい方の仮数部の下の方の桁が失われる。これを情報落ちという。足しこむときには小さい方から足していく。

### 14.3 桁落ち

大きさのほとんど同じ 2 つの実数の減算を行なう場合、指数部が等しく、n 桁の仮数部の上位 k 桁が等しいとすると、減算の結果仮数部の桁数は n − k になってしまう。これを桁落ちという。

## 本資料の参考元

本資料は鶴田先生、生田先生の授業資料と慶應義塾大学の光武先生、古池先生から頂いた資料を基に作成しています。また、奈良教育大学の薮哲郎氏「数値計算と fortran の基礎」(2012) を参考させて頂いた。