Fortran 演習

天野孝伸

2023 年 05 月 07 日

目 次

注釈: この文書について

この文書は東京大学理学部地球惑星物理学科 3 年生向けの演習科目「地球惑星物理学演習」用に書かれたものです．姉妹版として Python 演習 があります．これ以外の用途に使用することを妨げるものではありませんが，必ずしも万人向けのものではありませんのでご注意下さい．特に演習課題は Unix 系のコマンドラインでの実行を前提としています．

なお，当然のことながらこの文書の利用は自己責任でお願いします．

第1章　はじめに

この演習では非常に短い期間で Fortran によるプログラミングの基礎を学ぶことになる．ここで学ぶ内容は今後の演習に必須のものとなるので，心して取り組んで頂きたい．

とは言っても，プログラミング言語を用いて自分のやりたいことを思うがままに実現できるようになるのはそれほど簡単なことでは無く，この演習だけでは不十分であると思われる．プログラミング言語の文法自体は英語や日本語などの自然言語に比べたら格段に簡単だが，それでもそれを使いこなすのにはある程度の訓練が必要なのである．特にプログラミング初心者の人がこの演習の内容を全て理解するのは難しいので，分からない内容があっても落ち込むことは無い．(興味を持ってやっていればそのうちに自然と身に付くものである．) 最低限の目標は，Fortran というプログラミング言語の基礎を覚え，自分で簡単なプログラムの作成および実行が出来るようになること (=この後の演習の内容についていけるようになること) である．各節のタイトルに「†」がついている項目がいくつかあるが，この内容は演習の課題をこなすためには必ずしも知らなくて良い内容である．ただし，知っておくと色々と便利なことが多いので余裕があればこの内容もマスターしておくと良い．

なお演習時には言語仕様を事細かに解説するようなことはしない．そんなことをしていては時間が足りないし，何より退屈なだけである．また言語仕様の詳細を理解したからと言ってプログラムを書けるようになるわけではなく，全く実用的では無い．その代わりに，各事項を理解する助けになるようサンプルプログラムを多数用意してある．大事なのは自分でサンプルプログラムを修正，実行し，その動作を自分の目で確認することである．また毎日の課題に取り組むことで，「習うより慣れよ」の精神で実践的に基礎を身につけて行って欲しい．この文書についても教科書的なものでは無く，サンプルプログラムの解説といった位置付けである．また，各章の最後に用意してある演習課題には各自で取り組んで欲しい．

なお最初に断っておくと，この文書は **未完成** である．用語が統一されていなかったり，間違いなどを含んでいる可能性も大いにあるので，その点については注意して欲しい．内容としてはどちらかと言うと教科書には書いていないようなことを随時盛り込んでいく予定である (あくまで予定である)．基本的にはこの html の Web ページを正式版とするが，オフラインで勉強したい場合には以下から PDF をダウンロードすることも出来るので適宜活用して欲しい．(ただし PDF 版は機械的に変換したもので，サンプルファイルへのリンクなどが上手く動作しないので注意して欲しい．)

PDF 版: https://amanotk.github.io/fortran-resume-public/fortran-resume.pdf

この演習では説明しないような内容 (細かい文法事項など) については必要であれば以下を適宜参考にすると良い[1]．高木 (2009) は非常に良くまとまっているので，これがあれば困ることは少ないであろう．紙の本が良いという人は富田・齋藤 (2011) がよくまとまっている．少し古いバージョンが Web で参照でき，多くの人にはこれで十分である．牛島 (2007) は少し初心者には分かりづらいかもしれないが，モジュールに関する記述が実践的である．数値計算についても少し触れられているが，本格的に取り組む場合は専門の文献も併せて参考にした方が良い．

[1] 基礎的には本演習で扱う事項さえ理解してしまえば，分からないことは本で探すよりインターネットで探した方が早いので，自分で必要性を感じなければ教科書などは特に購入する必要は無い．更に言うと巷には古い (参考にしないほうが良い)Fortran の本があふれているので注意して欲しい．大抵はどこかの大学の年配の先生が書いた年季の入ったテキストを基に教科書としたものであるが，お世辞にも薦められないような本がごく最近にも出版されている．偏見たっぷりに言うと，サンプルプログラムが全部大文字で記述されているような本はかなりの確率でこの部類に入ると思って良い．

1. 高木征弘 (2009) 『地球物理学演習テキスト Fortran90/95 入門』
2. 富田博之，齋藤泰洋 (2011) 『Fortran90/95 プログラミング』 (培風館)
3. 牛島省 (2007) 『数値計算のための Fortran 90/95 プログラミング入門』(森北出版)

なおサンプルプログラム等は各章のリンクからダウンロードすることが出来るようになっている．また，以下の Github リポジトリ

https://github.com/amanotk/fortran-resume-sample

からまとめてダウンロードしても良い．各章ごとにディレクトリに分類して (例えば 3 章なら `sample/chap03`) 置いてあるので適宜手元にコピーして参照して欲しい．

ちなみに， `git` コマンドが使える（Unix 系の）マシンであればコマンドラインで

```
```

を実行すればサンプルが一括ダウンロードができる．（この場合は `sample` というディレクトリに格納される．）

このページとこれらのサンプルプログラムがあれば自習も十分に可能であるので，興味のある人はどんどん進めてもらって構わない．自習用に個人マシンに Fortran の開発環境を用意することも出来る．演習課題はリダイレクトなどの Unix コマンドラインの使用を前提としている．従って，自分のマシンに Unix-like 環境をインストールするとよいだろう．これには Windows なら WSL2(Windows Subsystem for Linux)，Mac なら Homebrew または MacPorts などをインストールすることで使えるようになるだろう．または VMWare や VirtualBox などを使って，仮想マシンとして Linux をインストールすることも出来る．興味のある人は教員や TA や Google 先生に相談してみよう．

1.1 プログラミングを学ぶ意義

いま

$$x = \cos x$$

なる非線形方程式の解を知りたいとしよう．残念ながら解析的には解は得られない．しかし逐次近似など，繰り返し計算で近似解を求める方法は色々と知られている．だからと言って紙と鉛筆で計算するのは大変なので計算機に任せよう，というのが基本的な考え方である．とは言っても計算機は自分で考えることはできない．どのような処理を行うかを逐一丁寧に教えてやらなければならない．これがプログラミングというものである．計算機を使って処理の自動化をする際の作法と言ってもよい．

もちろん，上の方程式の解を求めるだけなら何もプログラミングを学ぶ必要は無い．そのようなことをやってくれるソフトウェアは探せばいくらでも見つかるだろう．ところが汎用のソフトウェアは決められた計算をするには十分であるが，ちょっとでも違うことをやろうと思うと途端に困ってしまう．誰もやったことのないような最先端の仕事をするには結局のところ必要なものは自分で作らなければいけない．そのためにもここでプログラミングを学んでおくことは非常に重要である (と思う).

さて，自分のやりたいことを計算機で実現するにはどのような手続を処理させるかをまず整理しなければならない．この手続のことをアルゴリズムと言う．例えば上の例で言えば，非線形方程式を数値的に解くための手法のことである．アルゴリズムが決まればそれをプログラミング言語で記述し，計算機に実行させることができる．本演習では Fortran というプログラミング言語の使い方を覚えることが主たる目的であ

るので，アルゴリズムについては詳しく説明しない．しかし，プログラミング言語の文法は世の中に数多ある言語で様々であるが，アルゴリズムは言語が何であろうと変わらないので，実際にはアルゴリズムの理解の方が重要であることを十分に認識しておいて欲しい．

1.2 プログラミング言語について

これから学ぶのは正確には Fortran 90/95 と呼ばれる規格の言語である．90 とか 95 というのは 1990 年とか 1995 年に規格が定められたという意味である．Fortran 90 と Fortran 95 ではあまり違いが無いのでこのように記述することが多い．2000 年前後までは Fortran 77 という規格が幅を効かせていたのだが，最近では open source のコンパイラ (後述) が普及したこともあって，今から勉強するなら Fortran 90/95 が良い選択肢である．(より新しい規格として Fortran 2003 や 2008 も存在し，その主な違いはオブジェクト指向プログラミングのサポートである．これは明らかに本演習の範囲を超えているので扱わないことにする．) ただし太古の昔から脈々と受け継がれてきたプログラムなどは今でも古めかしい Fortran 77 のままで現役で使われている．そのようなプログラムに出会ってしまった時には諦めて Fortran 77 も勉強しよう．(実はそういうことは結構あるのだが，いくつかの違いさえ理解してしまえば，それほど難しいことでは無い．) 本演習では単に Fortran と言った場合には Fortran 90/95 を指している．

ちなみにプログラミング言語というのは星の数ほどあり，よく知られたものだけでも C/C++，C#，Java，Javascript，Perl，Python，Ruby，Lisp などがある．正直に言えば今更 Fortran を学ぶのは時代遅れであると言っても良い．多くの言語の文法が C 言語に近くできているのに対して Fortran は仲間外れの部類である．また，C 言語を習得した人が Fortran を習得するのは比較的容易いが，その逆は必ずしも真ではない．しかし一応フォローしておくと Fortran にもメリットはあって，分野にも依るが時代遅れと言われながらもしぶとく今でも現役で使われ続けている言語であり，この傾向はまだしばらく続くであろう．また，Fortran は細かいことを考える必要があまり無い比較的簡単な言語でもあるので，初心者にはとっつきやすいと同時にプログラミングの専門家では無い科学者向きの言語であるとも言える．

ただし基本的な考え方は他の言語でもあまり変わらないので，興味のある人はぜひ他の言語にも挑戦してもらいたい．おすすめは (少しとっつきにくいところはあるものの) 比較的簡単な言語でありながら応用範囲の広い Python である[2]．Python は Python 演習 で扱うが，これは Fortran(に限らずプログラミング言語一般) の知識をある程度習得している人向けである．また C 言語は教養として知っておくと非常に役に立つ．

注釈: この文書を書き始めたのはおそらく 2013 年頃であったかと思うが，それから約 10 年が経過した 2023 年現在でも，Fortran は依然としていわゆる High-Performance Computing（HPC）に現役で用いられている言語であり続け，Fortran が駆逐される時代はまだしばらく訪れないように思われる．

しかし，もし筆者が独断と偏見で学生に教育する言語を選択するのであれば Fortran は選ばないであろう．各研究分野の都合にも大きく依存するであろうが，私見ではレガシーコードの維持という観点以外では，学生に Fortran を教える積極な理由が見いだせてないためである．実際に新規開発される欧米の HPC コードやライブラリでは C++，特にいわゆる Modern C++を採用している例が多く見受けられるようになっている（その理由を考え始めると長くなるのでここでは述べない）．しかし，最近の C++は Fortran に比べると非常に難しく，一般の研究者向きの言語とは呼べないシロモノになってしまったのが悩ましいところである．ただし，多くの研究者にとっては Python が使えるようになっていれば，他の言語が必要になっても

[2] 実は Python を使うと C や Fortran の半分以下の行数でやりたいことが実現出来る場合が多く, 最近の計算機能力を考えると大規模計算をするのでなければ実用上はほとんど Python で事足りてしまう．実際に世界的に科学や工学の多くの分野で Python の導入が進んでいる．

比較的容易に対処ができるのではないだろうか（ちなみに最近の C++は Python に近い書き方ができるようになってきている）.

第2章　プログラムの作成と実行

まずは計算機が実行するプログラムの作成および実行する方法を学び，Fortran プログラムの基本的な構造について理解しよう．

参考:

- sample1.f90 : Hello, world
- sample2.f90 : Fortran プログラムの基本構造
- dot.Xmodmap : CapsLock と左 Control の交換
- dot.emacs.el : emacs の設定

この章の内容

2.1 Hello, world !

プログラミング言語を学ぶ時には "Hello, world !" を表示するプログラムから始めるのが慣例になっている．以下に示したものがその最初のプログラムである．

リスト 1: sample1.f90

```
program sample
  write(*,*) 'Hello, world !'
  stop
end program sample
```

これはソースコードと呼ばれ，人間が読める形式で記述されるテキストファイルである．拡張子は Fortran 90/95 の場合は `.f90` とする[1]．プログラムを実行するには，これをコンパイラと呼ばれるプログラムを用いて実行形式 (計算機が読み込んで実行することができる形式) に変換してやる必要がある．これをコンパイルと言う．この演習では gfortran というコンパイラを用いることにしよう．カレントディレクトリに `sample1.f90` というファイルがあることを確認して，ターミナルで以下のコマンドを実行しよう．

[1] Fortran 77 では `.f` である．

```

```

1 行目で gfortran コマンドによってコンパイルを実行している．コンパイルが成功すると `a.out` という名前の実行形式のファイルが作成される．2 行目で作成された `a.out` を実行し，結果が 3 行目に表示されている．このように"Hello, world !"が表示されれば成功である．

ちなみに

```

```

のように `-o` オプションを使って作成される実行形式のファイル名を指定することが出来る．上の場合は `hello` というファイルが作成されるのでこれを実行すると先ほどと同じ結果が得られるはずである．なおコンパイラに渡すことの出来るオプションは他にも山ほど存在するので，興味がある人は

```

```

で調べて欲しい．まあよく使うようなオプションはせいぜい数個程度であろう．このように，どんなプログラムであっても基本的には emacs や vi 等のエディタでソースコードを編集し，コンパイル，最後に実行という流れになる[2]．

注釈: プログラミングを始めたばかりの時にはコンパイラの吐き出すエラーメッセージの意味が分からず困ってしまうことも多いだろう．そんなときには，まずは落ち着いて英語のメッセージを読んでみよう．何がどうおかしいのかが書いてあるハズである．

なお，コンパイラのエラーメッセージは長くなりがちだが，最初のエラーから解決していったほうがよいことが多い．なぜなら後のエラーは前のエラーに引きずられて発生したものであることが多く，その場合は前が解決されれば自動的に後も解決されるからである．

いずれにせよ，自分で解決できそうもない時には遠慮なく質問しよう．

2.2 ソースコードの基本

以下に示すサンプルを例にとって Fortran のソースコードの基本的な構造を説明しよう．

リスト 2: sample2.f90

```
! これ以降は「コメント」として無視される．
! ソースコードは人間が直感的に理解できないので，適宜コメントを入れてプログラムの
! 内容を理解しやすくすることを推奨．
! 日本語も入力できるが，環境によっては文字化けする可能性がある．


program sample ! ここにコメントを書いても良い
```

(次のページに続く)

[2] そうでない場合もあるのだが，とりあえず今は気にしないことにしよう．

(前のページからの続き)

```
  ! 空行は無視される

  ! 標準出力に文字列を表示 (シングルクオートもしくはダブルクオートで囲む)
  write(*,*) 'Hello, world 2 !'

  ! 複数の文字列や変数をカンマで区切って並べてもよい
  write(*,*) "This is ", "also ", "OK"

  ! 1 文が複数行に渡る場合には `&` を用いる.
  write(*,*) &
       & 'This is a continuation line'

  print *, 'This is another way to print out'

  ! プログラムを終了する. 多くの場合無くても構わないがあったほうが無難.
  stop
end program sample
```

2.2.1 使用可能な文字

ソースコードの編集に用いることが出来るのは半角の英数字および下線 (アンダースコア) といくつかの四則演算などに使う特殊文字である. (要するに日本語は使えないと思えば良い.) なお Fortran は英字の大文字と小文字を区別しない (`fortran`, `Fortran`, `FORTRAN` は全て同一と解釈される) という現代においては大変珍しい言語の一つである. これは歴史的な事情によるものである. 古い Fortran 77 のソースコードには大文字だけで記述されているものも多く存在するが, 単に読みにくくなるだけなのでそのような意味のないことはやめよう.

2.2.2 コメント

ソースコードには計算機に実行させる命令だけでなくコメント (注釈) を含めることができる. Fortran では "!" から行末までがコメントとみなされる. コメントはプログラムの実行とは無関係であり, コンパイル時には単純に無視される. ソースコードはプログラミング言語固有のキーワードなどで記述されているため人間には理解しづらいのに対して, コメントには人間が理解できるように自由に説明を加えることができる. コメントはソースコードの可読性を良くするものなので積極的に活用すべきである[3]. なおコメントには日本語を用いても問題ないが, それ以外の部分に日本語を使うとコンパイル時にエラーとなる. 特に気づきにくいのがソースコード中に全角スペースが混じってしまってコンパイルが通らないという現象である. エディタに emacs を用いている場合は, 後に見るように設定によってこの問題を回避出来る.

[3] 自分が 3 日前に書いたコードが何をしているか理解できないというのは日常茶飯事である.

2.2.3 プログラムの構造

Fortran のソースコードは以下のように program と end program で囲まれる．以下のコードの program_name は基本的に何でも良く，分り易い名前を付けるのがよい．ただし Fortran の予約語 (program などのキーワード) は使えない．また後述の組込み関数と同じ名前にしてしまうとその組込み関数は使えなくなってしまうので注意して欲しい．stop はプログラムを終了するという意味であり，end program の直前では省略しても通常は問題無いのだが，入れておいた方がお行儀が良いのでそうしておこう．

```
program program_name

  ! ここに処理を記述する

  stop
end program program_name
```

また

```
write(*,*) 'This will be printed out to the termnal.'
```

これは

```
print *, 'This will be printed out to the termnal.'
```

と書くこともできる．動作はまったく同じであるので，どちらを用いてもよい．

とすると標準出力 (ターミナル) に文字列を表示することができる．write(*,*) はここではとりあえず文字列を出力するためのオマジナイと思っておいて欲しい．プログラミング言語を学ぶ時にはこの「オマジナイ」というやつが多く出てくるのだが，そのうち意味がわかるようになるので心配しなくて良い．なお read(*,*) で標準入力 (ターミナルからのキーボード入力) を読み込むことが出来るが，これについては変数を学んでから説明しよう．なお先ほどのサンプルの 13 行目の様に複数の文字列や後で説明する変数をカンマで区切って並べても良い．これを入出力リストと呼ぶ．

2.2.4 継続行

Fortran では 1 行が 132 文字以下でなければいけないという制限が課せられている．1 行に収まらない長い文を記述するには行の最後に & を記述することで，次の行へと継続することが出来る．実際には 80 文字とか 100 文字とか (要するにエディタの表示範囲で) 改行する方がプログラムが読みやすくなる．サンプルの 16-17 行目のような形で記述すればよい．17 行目の & は必ずしも必要では無いが，あった方が読みやすいので入れておくことを推奨する．なお emacs では改行したい位置で C-c RET と打つと & を自動で挿入して字下げまでしてくれる[4]．

[4] このようにキーバインドを表す際には C が Control，M が Meta キーを表す．例えば C-x は Control キーを押しながら x キーを押すことを，M-x は Escape キーを押して **離してから** x キーを押すことを意味する．

2.3 ソースコード編集にまつわるエトセトラ †

ソースコードの編集に用いるソフトウェアはテキスト形式のファイルが編集できるものであればどんなものであっても構わない．しかし世の中には専用に開発された便利なものがあるので利用しない手は無い．この演習では emacs の使用を推奨することにするが，当然他のエディタを使っても全く問題はない．最近人気の Visual Studio Code や Atom などのフリーでクロスプラットホームな高機能エディタを使ってみるのもよいだろう．また，Eclipse 等に代表される統合開発環境が好みの人もいるかもしれない．ただし，何を使うにしても大事なのはその機能を使いこなすことが出来るか (使いこなせないのであれば何を使っても同じ) である．

ひとつだけコメントしておくと，将来シミュレーションを生業にしたい人は emacs や vi などのエディタにある程度は慣れておいた方が良いかもしれない．というのは，ssh ログインしたリモートマシンでソースコードの編集をする機会が多くなると予想されるためである．vi はもちろん，emacs でも `emacs -nw` でターミナル上でソースコードの編集が可能である．両者とも使いこなすのは難しいが，その機能は折り紙つきなので，興味のある人は自分でどんどん調べてカスタマイズしていくと良い．

以下では emacs で (Fortran に限らず) ソースコードを編集する際に知っておいた方が良いことや，便利な設定などをほんの一部だけ紹介する．知らなくても効率が悪くなるだけで何かが出来なくなるわけではないので興味のない人は無視してもらって構わない．

2.3.1 Control と CapsLock の交換 (setxkbmap)

まず emacs では `Control` をかなり多様するので `A` のキーの左の `CapsLock` を `Control` と交換して使用する人がほとんどである (多分)．最近の Linux 系の OS 環境では `CapsLock` と `Control` を交換するには

```
```

とするのが手っ取り早い．

このコマンドをログイン時に自動で実行して欲しいわけだが，これには `~/.xsession` や `~/.xinitrc` などのファイルに上記のコマンドを書き込むことで実現出来ることが多い．ただし自動実行の設定は環境依存の話であるので自分の環境に合わせて適宜やり方を調べて欲しい．またこのようなキーボードのカスタマイズの仕方は他にも複数の方法があるので必要な人は自分で調べてみよう．

なお，同様に vi ユーザーは `Escape` を多用するのでスペースキーの左もしくは右のキー (普通の日本語キーボードでは「無変換」とか「変換」) を `Escape` として使う人が多いようである．

2.3.2 Control と CapsLock の交換 (古い方法)

以下の方法は古典的な方法で，最近の環境では必要ない (というか紛らわしい) 情報かもしれないが，念のため残しておく．

Linux 系の環境では `xmodmap` というコマンドでキーボードのカスタマイズが出来る．ホームディレクトリに `.Xmodmap` というファイルを作り，以下の様な内容で保存する．

```
        Lock =
     Control =
      37 =
      66 =
   Lock =
   Control =
```

そして

```
```

というコマンドを実行すると CapsLock と 左の Control キーが交換されたハズだ．ログイン時に自動でこのコマンドを実行するように設定しておけば便利である．

2.3.3 基本的な編集作業

まずマウスを使わずに作業できるようになることを目標にしよう．なぜなら将来リモートマシンにログインして作業する際にはマウスを使うことは出来ないからだ[5]．

既にこれまでの演習で学んだことだとは思うが emacs では (Control キーと CapsLock を交換しておけば) キーボードのホームポジションから手を動かさずに全ての作業が出来るようになっている．これにはカーソルキーに対応する C-n (↓ next-line)，C-p (↑ previous-line)，C-f (→ forward-char)，C-b (← backward-char)，と Delete キーに対応する C-h は必須である．また C-a (行頭へ移動)，C-e (行末へ移動) や，C-k (カーソルから行末までを削除) もよく使うので覚えておこう．ちなみに emacs とは関係無いが，多くの環境で Alt-Tab によってウィンドウの切り替えが出来る．またターミナルでのコマンドライン編集時にも emacs と同じキーバインドが使える場合が多いので，これらを覚えておくだけでマウスの使用頻度が激減すること請け合いである．

vi ユーザーの名誉のために述べておくと，当然のように vi でも (Escape の場所を適切にしておけば) ホームポジションから一切手を動かす必要が無い．更に言うと vi では片手でコーヒーを飲みながらカーソル移動が出来るので，少なくともこの点では vi の方が優れている (なので余裕があれば vi も勉強しよう).

2.3.4 Tab の利用

ソースコードは適切に字下げされていると格段に見やすくなるし，明らかな文法間違いに気づくきっかけにもなるので字下げの徹底を強く推奨する．emacs では Tab キーを押すと自動でカーソルのある行の字下げをしてくれる．複数行を一気に字下げしたい場合には字下げしたい領域を選択して M-x indent-region もしくは M-C-\ で選択された領域の字下げが出来る．ファイル全体を字下げしたい時には C-x h M-C-\ とすれば良い．

また Fortran は program に限らず，実行ブロックが end ??? で終わるようになっている．emacs では end まで入力した状態で Tab キーを押すと自動で end の後に適切なキーワードを挿入してくれる．(例えば program sample の場合は Tab キーによって end の後に program sample が挿入される．) これは非常に便利なので是非利用して欲しい．

[5] 出来ないことも無いのだが，やっぱり何をするのにも遅くてイライラするので．

2.3.5 コメントアウト

ソースコードを編集していると (特にデバッグ中は) 複数行をまとめてコメントアウトしたいことが多々ある．そんな時に各行の先頭にいちいち"!"を挿入するのはバカバカしい．emacs では領域を選択して `M-x comment-region` とするとまとめてコメントアウトしてくれるようになっている．ちなみにコメントアウトした領域を元に戻す時は `M-x uncomment-region` とすれば良い．なお多くの emacs のデフォルト環境で `.f90` のファイルを開いた場合には (自動で f90-mode というモードになり) `C-;` が `M-x comment-region` に設定されている．同様に `C-u C-;` が `M-x uncomment-region` である．

2.3.6 全角スペースの表示

前述の通りソースコードに全角スペースが含まれているとコンパイルが出来ない (しかもエラーメッセージからはそれが分からない) という厄介な問題が存在する．これは初心者ほど陥りやすい罠であるが，emacs の設定でこの問題を回避することが出来る．例えば以下の様な設定を `.emacs` などの設定ファイルに書いておくと全角スペースが "□" と表示されるので一目瞭然である．

```
(require 'whitespace)
(setq whitespace-style '(face trailing spaces tabs space-mark tab-mark))
(setq whitespace-space-regexp "\\(\x3000+\\)")
(setq whitespace-display-mappings
      '((space-mark ?\x3000 [?\□])
        (tab-mark   ?\t   [?\xBB ?\t])
        ))
(setq fgcolor "RosyBrown1")
(setq bgcolor "blue3")
(set-face-attribute 'whitespace-trailing nil
                    :foreground fgcolor
                    :background bgcolor
                    :underline t)
(set-face-attribute 'whitespace-empty nil
                    :background bgcolor)
(global-whitespace-mode t)
```

なお上の設定では行末のスペースやタブなども表示するように設定されているが，このあたりは完全に好みである．以下は emacs でソースコードを編集中のスクリーンショットである．

2.3.7 複数ファイルの編集

真の emacs 使いはいちいち emacs を立ち上げたり終了したりはしない．常に emacs を立ち上げておき，複数のファイル (emacs ではバッファと呼ぶ) を縦横無尽に切り替えながら編集するのである[6]．`C-x f` で新しいファイルを開くことは当然出来るとして，過去に編集していたバッファに切り替える方法を覚えておこう．`C-x b` でミニバッファに既に開いているバッファ名 (ファイル名) を入力するとそのバッファに移るこ

[6] emacs 上でコンパイルしたり，emacs をターミナル代わりに使う人もいるのだが少しばかりマニアックな話題なので各自で調べて欲しい．

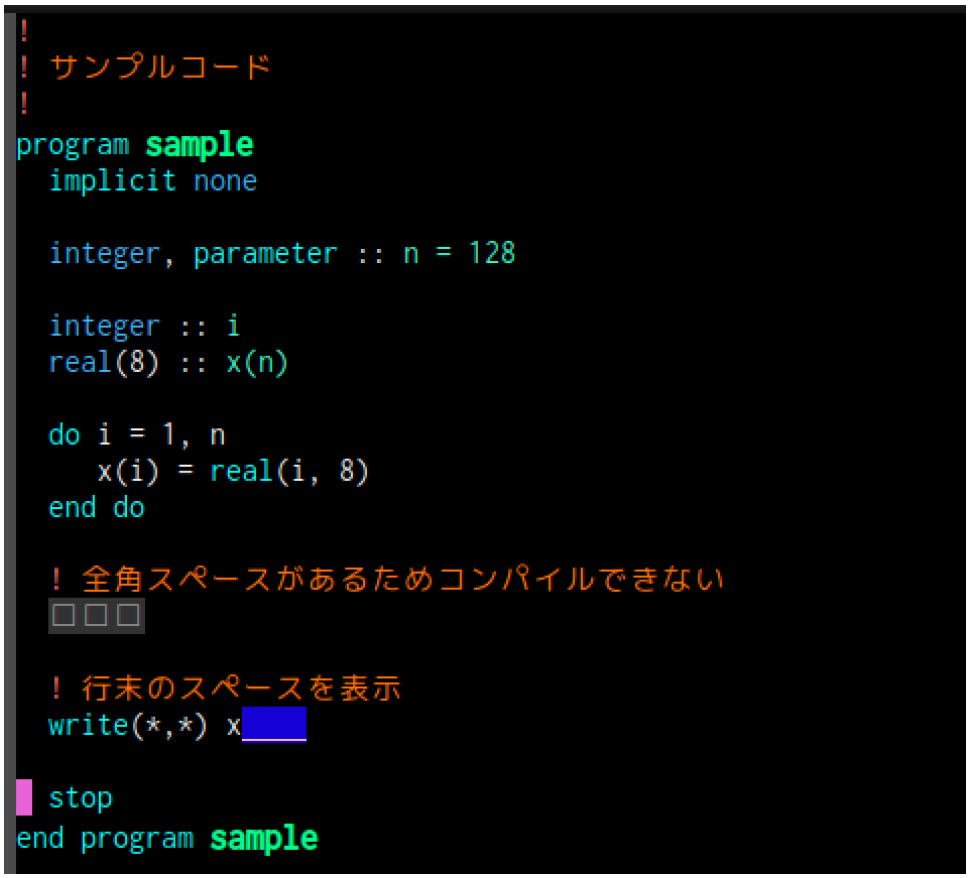

図 1: Emacs のスクリーンショット

とが出来る．また Tab による補完も使うことが出来る．C-x C-b では現在開いているバッファの一覧が表示され，選択することでそのバッファに移ることが出来る．

なお最近では emacs にもタブ (タブブラウザのあれである) を導入することが出来るらしい．興味のある人は tabbar.el で調べてみよう．

2.3.8 まとめ

細かいことはどうでもいいという実践派の人は以下の表を頭に入れておけばよい．正確には体が勝手に覚えるものなのだが．

表 1: これだけは知っとけ emacs のキーバインド

キーバインド	説明
C-p	上へ移動
C-n	下へ移動
C-f	右へ移動
C-b	左へ移動
C-a	行頭へ移動
C-e	行末へ移動
C-h	Delete
C-k	カーソル位置から行末までを削除
C-@/C-space	領域選択を開始
C-w	選択範囲を切り取り
M-w	選択範囲をコピー
C-y	コピーした内容を貼り付け
C-c ;	選択領域をコメントアウト
C-u C-;	選択されたコメントアウトされている領域のコメントを外す
C-c RET	継続行の挿入
M-C-\	選択された領域の字下げ
C-x h M-C-\	ファイルの全てを選択して字下げ
C-x f	ファイルを開く
C-x b	バッファの切替え
C-x C-b	バッファ一覧の表示
C-s	前方検索
C-r	後方検索
M-%	置換 (置換前にその都度確認をする)

2.4 第 2 章 演習課題

参考:

- 課題 2 解答例

2.4.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．

2.4.2 課題 2

サンプルプログラムをエディタで修正，コンパイルし、実行して結果を確認せよ．例えば sample1.f90 の出力を `Hello, Fortran !` に変更して出力を確認せよ．

2.4.3 課題 3 †

本演習では扱わないが，C 言語や Python の場合についてもプログラムの実行の方法も知っておこう．

C 言語は Fortran と同じようにコンパイル言語なので，ソースをコンパイルして実行する．C 言語のコンパイラとしては gcc を用いることが出来る．ここでは hello.c をコンパイル，実行してみよう．

```
```

このように，コンパイラのコマンドとして gcc を使うこと以外は基本的に同じである．

Python はスクリプト言語と呼ばれ，コンパイルが必要なく，直接ソースから実行することが出来る．ここでは hello.py を実行してみよう．

```
```

また，hello.py に実行権限があれば直接実行することもできる．実行権限を付与するには

```
755
```

などとすればよい．このときは

```
```

のように実行することができるだろう．

プログラムの規模が大きくなってくると，次第にコンパイルにも時間がかかるようになってくるが，Python のような言語ではコンパイルの必要がないことが大きな利点となる．一方で実行速度は（書き方に大きく依存するものの）C 言語や Fortran の方が一般には高速である．

第3章　変数・データ型・基本的な計算

ここでは計算機でデータを扱うには必須となる変数とデータ型および基本的な計算の仕方について学ぼう．

参考:

- sample1.f90 : 変数の基本
- sample2.f90 : データ型と精度
- sample3.f90 : 定数
- sample4.f90 : `read` の使い方
- sample5.f90 : 算術演算，代入，組込み関数
- sample6.f90 : 型変換

この章の内容

3.1 変数

プログラム中で何らかの値を保持するためには変数を用いる必要がある．例えば一度には出来ないような複雑な計算の途中結果などは変数に格納することになる．要するにデータの入れ物である．変数を用いるにあたって，

- 変数は宣言しなければならないこと
- 変数には型があること

という 2 点に注意しなければならない．変数の宣言は

```
```

のような形で行う．(ここで" `::` "は必須ではないが，宣言と同時に初期化をしたり，後で出てくる変数の属性を指定する時には必要となる．従ってこの演習では変数宣言時には常に" `::` "を用いることにする．)

例えば以下のコードは n と x という 2 つの変数を宣言し，それぞれ値を代入している．ここで = は数学で用いる記号とは異なり，= の左側の変数に右側の値を代入する (データを格納する) という意味である．

リスト 1: sample1.f90

```
program sample
  implicit none ! 暗黙の型宣言禁止

  ! 変数を使う前には必ず以下のように宣言を行う
  integer :: n ! 整数型の変数の宣言
  real :: x    ! 実数型の変数の宣言

  ! 代入
  n = 10
  x = 3.14

  ! 表示
  write(*,*) 'integer => ', n
  write(*,*) 'real => ', x

  stop
end program sample
```

上の例では n は整数 (integer) を，x は実数 (real) を表す変数であるが，これはそれぞれ 5 行目や 6 行目のように変数を宣言をした時点で確定する．面倒なように思われるかもしれないが，**変数は使う前に用途に合わせて** 宣言しなければならない．これはすぐ後に述べるように計算機が表現できる値に限界があり，また人間のように臨機応変に状況に対処できないからである．

なお Fortran には暗黙の型宣言という悪しき慣習があり，宣言されていない変数でも変数の名前に応じて自動的に型を仮定して宣言されたものとみなす．詳細は省くがこれは明らかにバグの元であり，この機能は使わないことを強く推奨する．先ほどの例では program 文の直後（2 行目）の

```
implicit none
```

が暗黙の型宣言の禁止を意味する．このときは全ての変数を明示的に宣言しなければコンパイルエラーとなる．以降，本演習では必ず implicit none を使うこととする[1]．なお gfortran では -fimplicit-none というオプションを用いると，デフォルトで implicit none を指定した状態にすることが出来る．

[1] 巷で流行りのスクリプト言語では変数宣言は要らないじゃないかという人もいるかもしれないが，それは動的型付き言語だからそれでも良いのである．C や Fortran のような静的型付き言語ではその限りではない．

3.2 データ型と精度

標準の Fortran で用いることができるデータ型として以下のようなものがある．

表 1: 使用可能なデータ型

型名	キーワード	用途
整数型	`integer`	整数 (厳密な表現)
実数型	`real`	実数 (近似的な表現)
複素数型	`complex`	複素数 (実部と虚部を表す 2 つの実数型の組み合わせ)
文字型	`character`	文字を表す
論理型	`logical`	真偽値 (`.true.` または `.false.`)

注意しなければならないのは，特に実数型の `real` (従って当然 `complex` も) はあくまで実数の近似表現であるという点である．例えば `1.0` を代入したとしても，これが厳密に 1 を表しているわけではない．これは 10 進数を 2 進数で無理やり表そうとするために起きる問題であり，回避する手段は無い．そうは言っても多くの場合において十分な精度で実数を近似できているので問題が無いのである．これとは対照に，整数型 `integer` は厳密に整数を表現することが出来る．

しかしながら整数型にしても実数型にしても，どんな値でも表現できるというわけではない．具体的には各データ型に何バイト[2] の領域を持たせるかによって表現できる値の範囲が変わる．Fortran では確保する領域の大きさを変数の宣言時に明示的に指定することが出来る．すなわち

リスト 2: sample2.f90 抜粋

```
! 整数型
integer(kind=4)      :: i4      ! 4 バイト（32 ビット）の整数型
integer(kind=8)      :: i8      ! 8 バイト（64 ビット）の整数型

! 実数型
real(kind=4)         :: r4      ! 4 バイト（32 ビット）の実数型（単精度）
real(kind=8)         :: r8      ! 8 バイト（64 ビット）の実数型（倍精度）
real(kind=16)        :: r16     ! 16 バイト（128 ビット）の実数型（4 倍精度）

! 複素数型（実数 2 つ分の領域が必要になる）
complex(kind=4)      :: c4      ! 8 バイト（64 ビット）の複素数型    = 単精度
complex(kind=8)      :: c8      ! 16 バイト（128 ビット）の複素数型 = 倍精度
```

のように型名の後に () でデータ領域の大きさを指定できる．これを `kind` パラメータと呼ぶ．なお, `real(8)` のように `kind=` は省略して構わない．(`kind` パラメータを用いると移植性の高いプログラムを作成することが出来るが，これは本演習の守備範囲を超えるので以降では `kind=` は省略することとする．) ちなみに特に何も指定しない場合は `integer` が 4 バイト，`real` が 4 バイトとなっていることが多いが，これは処理系依存である．処理系依存などの細かいことはとりあえず忘れると，結果的に表すことのできる値の範囲は以下のようになっていると思えばよい．(実数の値の範囲については **実数の精度と誤差** でもう少し細かく説明する．)

[2] 通常 1 バイトは 8 ビット，すなわち 1 バイトあたり $2^8 = 256$ 通りの表現が可能である．

表 2: 各データ型の表現できる値の範囲

型名	最小値	最大値	備考
integer(2)	-2^{15}	$2^{15}-1$	
integer(4)	-2^{31}	$2^{31}-1$	
integer(8)	-2^{63}	$2^{63}-1$	
real(4)	$\sim 10^{-38}$	$\sim 10^{+38}$	値は絶対値, 精度は約 7 桁
real(8)	$\sim 10^{-308}$	$\sim 10^{+308}$	値は絶対値, 精度は約 16 桁

なお real(4) を単精度，real(8) を倍精度，real(16) を 4 倍精度と呼ぶのが通例である．特に実数型については，単精度の約 7 桁という精度では心もとないので現在では倍精度を用いるのが一般的である．本演習では特段の理由がない限り real(8)，complex(8) を用いる (complex(8) では実部と虚部がそれぞれ real(8) となる).

なお複素数型 complex の定数は (実部，虚部) という形で表す．例えば

リスト 3: sample2.f90 抜粋

```
! 複素数型の変数に 2.71 + 0.99 i を代入
c4  = (2.71, 0.99)
```

は倍精度複素数型の変数 c4 に $2.71 + 0.99i$ を代入している．

また文字型 (character) では通常 kind=1 なので[3]，kind パラメータを指定する必要がない．character で複数の文字 (文字列) を表すには以下のように len= で文字数を指定することになる．(この場合は len= を省略することも出来る．)

リスト 4: sample2.f90 抜粋

```
! 文字列型は少し特殊で通常は kind=1 である。文字列の長さは len=で指定する
character(len=256) :: char  ! 256 文字分
```

論理型 (logical) は真偽値を表すために用いるので，通常は kind パラメータは指定する必要は無い．

3.3 定数

数値などを直接ソースコードに記述するとそれは定数 (定数リテラル) と呼ばれる．例えば 99 や 1.5 などのような表現である．定数に _4 や _8 などを付けることによって kind パラメータを指定することも出来る．先ほどの例では 99_4，1.5_8 などのように書くことが出来る．論理型の定数は .true. もしくは .false. のどちらかである．文字型の定数は既に最初のサンプルで見たように ' (シングルクォート) もしくは " (ダブルクォート) で囲まれた文字列，例えば 'earth' や "physics" などである．

また parameter 属性を用いて変数のように名前付きの定数を使用することも可能である．

[3] ASCII コードは 1 バイトで足りるため.

リスト 5: sample3.f90 抜粋

```
  integer(4), parameter :: n  = 8_4
  real(8), parameter    :: pi = 3.141592653589_8
  integer(4) :: m
  real(8)    :: f, g

  ! 普通の変数と同じように定数を参照出来る
  m = n * 10
  f = pi * 2
  g = 3.0e+10_8

  ! 次のような定数変数への代入を行うコードがあるとコンパイルエラーとなる
  ! pi = 3.14
```

上の例では n を 4 バイトの整数，pi は 8 バイトの実数として，それぞれ値を指定している．これらの変数は parameter が指定されているため定数として扱われ，プログラム中 (15 行目) で誤って pi = 3.14 などとして値を変更しようとするとコンパイルエラーとなる．プログラム中で絶対に変更されない値を扱う場合にはこのように名前付き定数として宣言しておくと値が変更される心配が無いので安心である．(信じられないかもしれないが，プログラムの規模が大きくなってくると，このようなミスによるバグに悩まされることがしばしば起こる．)

また実数で例えば 3×10^{10} を表現するには上の例の 12 行目のように 3.0e+10_8 のように書けば良い．ちなみに Fortran 77 の慣習では倍精度での定数値を表現するのに e の代わりに d を使っていたので，これを 3.0d+10 と書くと倍精度，すなわち 3.0e+10_8 と同じ意味となる．このように実数の定数に e や d を用いる表現は今でもかなり頻繁に見られるので知っておくと良い．

3.4 標準入力から変数への値の代入

以下のように変数 x を宣言しておいて read(*,*) を用いると，プログラムの実行時にキーボードからの入力された内容を読み込み，変数 (この場合は x) に代入することが出来る．

リスト 6: sample4.f90

```
program sample
  implicit none

  real(8) :: x, y

  write(*,*) 'Input two real numbers: '

  ! 整数を読み込む
  read(*,*) x, y

  write(*,*) 'average = ', (x + y)/2
```

(次のページに続く)

(前のページからの続き)

```
  stop
end program sample
```

実行結果は以下のようになる．

```
2                                   # キーボード入力
3                                   # キーボード入力
 average =     2
```

ここで 3 行目と 4 行目はキーボードで入力した値である．この `read(*,*)` の意味は後述するのでここでは再びオマジナイであると思っておこう．なお，`read(*,*)` の場合も，`write(*,*)` と同様に複数の変数を並べて指定することができる．

3.5 算術演算

Fortran では最も基本的な演算である四則演算およびべき乗を以下のように計算することが出来る．当然変数同士での演算も可能である．

リスト 7: sample5.f90 抜粋

```
  write(*,*) 'addition       => ', 2.0 + 3.0  ! 足し算
  write(*,*) 'subtraction    => ', 5.0 - 3.0  ! 引き算
  write(*,*) 'multiplication => ', 2.0 * 3.0  ! 掛け算
  write(*,*) 'division       => ', 5.0 / 2.0  ! 割り算
  write(*,*) 'power          => ', 2.0 ** 3.0 ! べき乗
```

演算実行の優先順位は **べき乗 > 乗算 = 除算 > 加算，減算** の順となっているが，分かりにくい場合は，以下の例のように可読性のために () で明示的に演算の順番が分かるようにしておくと良い．

リスト 8: sample5.f90 抜粋

```
write(*,*) 'w/  parenthesis => ', 2.0 * (2.0 + 3.0)   ! 括弧あり => 10.0
write(*,*) 'w/o parenthesis => ', 2.0 *  2.0 + 3.0    ! 括弧なし => 7.0
```

3.6 代入

既に学んだように = 演算子を用いて左辺で指定する変数に値を代入することが出来る．この時，右辺には任意の演算を含んでも良い．例えば

リスト 9: sample5.f90 抜粋

```
a = 2.0          ! 変数 a に代入
b = a + 3.0      ! 足し算（a + 3.0）の結果を b に代入
c = a - 1.0      ! 引き算（a - 1.0）の結果を c に代入
d = a * b        ! 掛け算（a * b）の結果を d に代入
e = a / b        ! 割り算（a / b）の結果 ed に代入
```

のようにすればよい．

3.7 組込み関数

Fortran には標準で使える関数が多く用意されており，組込み関数と呼ばれる．関数というと数学の関数を思い浮かべるかもしれないが，必ずしも数学関数ばかりではない．関数というのは単に入力値を受け取り何らかの値を返す機能 (function) のことである．例えば数学では $f(x) = \sin(x)$ と書いた時には x という入力に対して $\sin(x)$ という値を返すことを意味する．Fortran でも入力値 **x** 対して **sin(x)** とすることで関数値を計算することが出来る．なお関数に渡すパラメータ (ここでは **x**) のことを **引数** と呼び，関数が返す値のことを **返値** と呼ぶ．

リスト 10: sample5.f90 抜粋

```
!
! 標準入力から値を読み込み変数 x に代入
!
write(*,*)
write(*,*) 'Input a real number: '
read(*,*) x

!
! 標準で様々な関数が用意されている
! 以下はほんの一例
!
! 平方根      => sqrt(x)
```

(次のページに続く)

(前のページからの続き)

```
! 絶対値      => abs(x)
! 三角関数    => sin(x), cos(x), tan(x)
! 指数関数    => exp(x)
! 対数関数    => log(x), log10(x)
! 双極関数    => sinh(x), cosh(x), tanh(x)
! 逆三角関数 => asin(x), acos(x), atan(x)
!
write(*,*) sin(x) ! sin(x) を計算し表示
write(*,*) cos(x) ! cos(x) を計算し表示
```

この他にも様々な関数が用意されているので，必要に応じて調べて欲しい[4]．なお，自分で独自の関数を定義して用いる方法は後に学ぶことになる．

3.8 型変換

異なる型同士の演算を行う場合や，代入する際に左辺と右辺で型が異なる場合には **より一般的な型へと変換された後に演算や代入が実行される**．この機能は便利なようで時に注意が必要な場合がある．

例えば以下の例を考えよう．

[4] 例えば 高木 (2009，3 章).

リスト 11: sample6.f90 抜粋

```
! 以下の代入文は意図した通りに動かない
! 右辺は整数同士の演算なので 0 となり，それが代入時に実数型に変換される
z = 2 / 3
```

z = 0.666... となるかと思いきや，実際には z = 0 となってしまう．これは左辺が整数同士の演算として行われるため (2 / 3 は 0) である．これを回避するには例えば z = 2.0_8 / 3 や z = 2 / 3.0_8 とすれば良い．どちらかが実数であればもう一方も実数に変換されてから計算されるので，演算結果も実数となる．ただし z = 2.0 / 3 のようにしてしまうと z は倍精度 (real(8)) で宣言されているにも関わらず 2.0 は単精度の実数 (real(4)) と解釈され，右辺の計算結果も単精度実数となる．これが左辺の z に代入される時に倍精度 (real(8)) に変換されるため，結果的には精度が失われることになってしまう．

以下の組込み関数を用いて明示的に型変換を行うことも出来る．例えば，real(1, kind=8) によって整数 1 が倍精度実数の 1.0_8 に変換される．ここでも 2 番目の引数を指定し忘れると精度が失われるので注意が必要である．ただし kind= は省略可能であり，real(1, 8) とするだけでも良い．

表 3: 型変換を行う組み込み関数

関数名	説明
int(x)	integer へ変換 (切捨て)
int(x, kind=k)	integer(k) へ変換 (切捨て)
real(x)	real へ変換
real(x, kind=k)	real(k) へ変換
cmplx(x)	complex へ変換 (実部が x，虚部は 0)
cmplx(x, y)	complex へ変換 (実部が x，虚部は y)
cmplx(x, y, kind=k)	complex(k) へ変換 (実部が x，虚部は y)

型変換関数を使った最も丁寧なやり方は

リスト 12: sample6.f90 抜粋

```
! 明示的に変換する
z = real(2, kind=8) / real(3, kind=8)
```

のようなものである．

3.9 第 3 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例

3.9.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

3.9.2 課題 2

標準入力から三角形の底辺 l と高さ h を読み込み，三角形の面積を表示するプログラムを作成せよ．

例えば以下のような実行結果が得られる．

```
3                                       # キーボード入力
5                                       # キーボード入力
                    7
```

3.9.3 課題 3

単精度実数型 `real(4)` および倍精度実数型 `real(8)` で，それぞれ $\tan(\pi/4) = 1$ なる関係式を用いて π の値を求めて表示し，その値の精度を確認せよ．(組み込み関数 `atan(x)` が数学の $\tan^{-1}(x)$ に対応している．) なお，精度を確認する際には 4 倍精度実数型 `real(16)` でも同様に π の値を求め，これを正確な値 (真値) とみなし，それとの相対誤差を確認すること．ただし相対誤差は $|1 - $ 近似値/真値 $|$ で評価せよ．(絶対値を返す関数 `abs(x)` を用いよ．)

例えば以下のような結果が得られる．

```
3                                       2
↪
3                                       3
↪
3
```

結果は上から，単精度の π とその精度，倍精度の π とその精度，4 倍精度の π をそれぞれ表している．ただし細かい数値は結果は環境依存である．(ここで，例えば `2.7E-0008` は 2.7×10^{-8} を表すのでとても小さい値であることを意味する．)

3.9.4 課題 4

標準入力から複素数 $z(=x+iy)$ を読み込み， e^z および $e^x(\cos y + i\sin y)$ をそれぞれ計算し，その結果が等しいことを確認せよ (組み込み関数 `exp(x)` および `sin(x)` ， `cos(x)` を用いればよい)．ただし倍精度の複素数型 `complex(8)` を用いること．なお複素数 `z` の実部は `real(z)` ，虚部は `aimag(z)` という組み込み関数でそれぞれ求めることができる．またキーボードから複素数の入力は (実部，虚部) という形式となることに注意せよ．例えば $z=1+i$ について $\exp(z)$ を求めるには

```
(1     1   ) # キーボード入力
(  1                          2                          )
(  1                          2                          )
```

のように (実部，虚部) という形式で入力すれば良い．

3.9.5 課題 5

キーボード入力で与えられた実数 x について，テイラー展開の公式

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}\cdots$$

を適当な次数 (例えば 2 次とか 3 次) で打ち切りることで $\sin x$ の近似値を求め，組み込み関数 `sin(x)` で求めた値と比較するプログラムを作成せよ．例えば $x=0.01,\ 0.1,\ 0.2$ などについて，実行して結果を確かめること．

例えば以下のような結果が得られればよい

```
0                          # キーボード入力
 0                         # 1 次近似
 0                         # 3 次近似
 0                         # 5 次近似
 0                         # 7 次近似
 0                         # 組み込み関数
```

第4章　制御構造

ここではプログラムの動作を制御するための文法について学ぼう．と言っても覚えなければいけないことは `if` による条件分岐，`do` による繰り返し，`select` による条件分岐のみである．`goto` という構文も存在するのだが，これはバグのもとになることから一般的には使わないほうが良いとされており，従ってここでも敢えて扱わない．

参考:

- sample1.f90 : 条件分岐 (if)
- sample2.f90 : 反復処理 (do ループ 1)
- sample3.f90 : 反復処理 (do ループ 2)
- sample4.f90 : 反復処理 (do ループ 3)
- sample5.f90 : 条件分岐 (select)

この章の内容

4.1 条件分岐 (if)

`if` による条件分岐は例えばユーザーの入力によって動作を変更する場合などに用いる．典型的な使い方は以下のようなものである．(これは閏年の判定例である．)

リスト 1: sample1.f90 抜粋

```
integer :: year

! 標準入力から整数を読み込む
write(*,*) 'Input year: '
read(*,*) year

! 基本的な if による分岐
if (mod(year, 400) == 0) then
   write(*,*) 'Leap year'
else if (mod(year, 100) == 0) then
```

(次のページに続く)

(前のページからの続き)

```
    write(*,*) 'Common year'
  else if (mod(year, 4) == 0) then
    write(*,*) 'Leap year'
  else
    write(*,*) 'Common year'
  end if
```

このように if に続く () の中に条件式 (conditional) を記述し，その条件が真 (.true.) の時には then に続く処理が実行され，偽 (.false.) の時には else if または else で更に条件判定をすることになる．また

```
if( conditional ) then
  ! 処理
end if
```

や

```
if( conditional ) then
  ! 処理 1
else
  ! 処理 2
end if
```

のように書くことも出来る．構文自体はそれほど難しくないので，ここで注意すべきは条件判定の部分だけであろう．以下の表に条件式に用いられることの多い演算子をまとめてある．なお Fortran 77 では > のような演算子 (関係演算子と呼ばれる) は正式にはサポートされていなかった．このため古いコードには .gt. のような演算子を見かけることもあるかも知れないが，自分で新しくプログラムを作成する際にはこのような古い形式は使うべきではない．新しい形式は /= 以外のものについては C 言語を始めとする他の多くの言語と同じなのでこちらを用いることを強く推奨する．ちなみに C 言語などでは /= ではなく != が用いられる．

特に実数の値が等しいかどうかを判定する際には注意が必要である．すなわち，実数に対しては == を使うことは出来ない．なぜなら 2 つの実数がほぼ等しいように見えても == による判定では全てのビットが厳密に等しくなければ真とは判定されないからである．従って，代わりに例えば差の絶対値 abs(A-B) が十分小さいかどうかで判定しなくてはならない[1]．

表 1: 条件演算子

演算子	Fortran 77 形式	意味
A > B	A .gt. B	A の方が B よりも大きければ真
A >= B	A .ge. B	A が B 以上であれば真
A < B	A .lt. B	A の方が B よりも小さければ真
A <= B	A .le. B	A が B 以下であれば真
A == B	A .eq. B	A と B が厳密に等しければ真
A /= B	A .ne. B	A と B が等しくなければ真

また条件判定が複雑な時には以下の論理演算子を用いることになるだろう．

[1] abs(x) は x の絶対値を返す組込み関数である．

表 2: 論理演算子

演算子	意味	使い方
.and.	論理積	(条件式 1) .and. (条件式 2)
.or.	論理和	(条件式 1) .or. (条件式 2)
.not.	否定	.not. (条件式)
.eqv.	論理等価	(条件式 1) .eqv. (条件式 2)
.neqv.	論理非等価	(条件式 1) .neqv. (条件式 2)

使い方は例えば

```
integer :: n

if ( 2 < n .and. n < 5 ) then
  write(*,*) 'n is larger than 2 and smaller than 5'
end if
```

と言った具合である．2 < n < 5 のような数学的な書き方はできないので注意が必要である．

さらに複雑な条件分岐の場合には以下のように if 文を入れ子で使うことも出来る．

```
if ( conditional 1 ) then
  if ( conditional 2 ) then
    ! 処理 1
  else
    ! 処理 2
  end if
end if
```

ただし何重にも深く入れ子になった if 文の実行効率はあまり良くないので出来るかぎり浅い条件分岐に留めておいた方が良い．

以下は sample1.f90 の 11-19 行目の閏年の判定と全く同じことを入れ子にした if で実装した例である．

リスト 2: sample1.f90 抜粋

```
  ! 同じことを入れ子の if で実現
  if (mod(year, 4) == 0) then
     if (mod(year, 100) == 0) then
        if (mod(year, 400) == 0) then
           write(*,*) 'Leap year'
        else
           write(*,*) 'Common year'
        end if
     else
        write(*,*) 'Leap year'
     end if
  else
```

(次のページに続く)

(前のページからの続き)

```
    write(*,*) 'Common year'
  end if
```

4.2 反復処理 (do)

4.2.1 決まった回数の繰り返し (do)

決まった繰り返しの処理をするために用いるのが do (従ってこの反復処理は do ループと呼ばれる) である．これも使い方は至ってシンプルである．

リスト 3: sample2.f90 抜粋

```
  write(*,*) '--- Do loop #1 ---'
  sum = 0
  do i = 1, 10
     sum = sum + i
     write(*,*) i, sum
  end do
```

は 1 から 10 までの和を順次求めながら結果を出力している．より一般には

```
do i = lower, upper, stride
  ! 繰り返し処理
end do
```

のような形で書くことになる．上の例では整数型変数 i は do 変数と呼ばれ，do ループの中で i の値が lower から upper まで stride ずつ変化する．stride は省略することも可能であり，その場合は 1 と解釈される．また stride は負の値であっても良い (当然この時は lower > upper でなければループ内の処理は実行されない)．通常 do 変数は整数型でなければならないが，実数型などでもコンパイル出来てしまう環境もあり，そのような場合は思わぬバグの原因となってしまう．間違いを未然に防ぐためにも do 変数には整数型を用いること．

例えば次は先ほどの例の和を求める処理を 1 つおきに実行する．

リスト 4: sample2.f90 抜粋

```
  write(*,*) '--- Do loop #2 ---'
  sum = 0
  do i = 1, 10, 2
     sum = sum + i
     write(*,*) i, sum
  end do
```

また if 文の場合と同様に do ループに関しても以下のように入れ子 (多重ループ) にすることが出来る．以下は 2 重ループの例である．

リスト 5: sample2.f90 抜粋

```
  write(*,*) '--- Do loop #4 ---'
  do i = 1, 3
     do j = 1, 3
        write(*,*) '(', i, ',', j, ') => ', 3*(i-1) + j
     end do
  end do
```

この部分の実行結果は以下のようになる．

```
            #4 ---
(           1           1 ) =           1
(           1           2 ) =           2
(           1           3 ) =           3
(           2           1 ) =           4
(           2           2 ) =           5
(           2           3 ) =           6
(           3           1 ) =           7
(           3           2 ) =           8
(           3           3 ) =           9
```

4.2.2 条件を指定した繰り返し (do while)

繰り返しの処理には基本的に先ほどの do ループを用いれば良いのだが，これを少し違った形式で行う do while なる構文も用意されている．これは

```
do while( conditional )
  ! 繰り返し処理
end do
```

のような形で用い，() 内の条件式が真 (.true.) の間は繰り返し処理が行われる．例えば

```
integer :: i
real(8) :: sum

i = 1
sum = 0
do while(i <= 10)
   sum = sum + i
   i = i + 1
  write(*,*) i, sum
end do
```

は先ほどの例 sample2.f90 の最初の do ループと同じ処理を実行する．この場合はあえて do while を用いる意味はあまり感じられないが，繰り返し回数が予め分からない処理ではこのような形式を用いるとスマートに書ける場面にもしばしば遭遇する．例えば反復計算によって実数型の値の収束判定をする場合などは

```
real(8) :: x

do while(abs(x) > 1.0e-8_8)
  ! 繰り返し処理
end do
```

などのように非常にスッキリと記述できる．この例では abs(x) の値が 10^{-8} 以下になるまで反復を続ける．例えば，以下は逐次計算によって平方根を求める計算である．

リスト 6: sample3.f90 抜粋

```
  !
  ! 逐次近似によって平方根を求める
  !
  do while( abs((sqrt_x0-sqrt_x1)/sqrt_x0) > tolerance )
     sqrt_x0 = (sqrt_x0 + sqrt_x1) * 0.5_8
     sqrt_x1 = x / sqrt_x0
  end do
```

4.2.3 複雑な処理 (exit と cycle)

単純な繰り返しだけでなく，より柔軟な制御を行うには exit や cycle を用いる．exit では do ループの中から途中で抜けることが出来，cycle ではループ内のそれ以降の処理を行わずにループ先頭に戻ることがで出来る．これらを用いると意図的に作った無限ループから条件を満たした時だけ抜け出すようなプログラムも簡単に作ることができる．例えば以下の例を見てみよう．

リスト 7: sample4.f90 抜粋

```
  ! 無限ループ
  do while( .true. )
     ! 標準入力から読み込む
     write(*,*) ''
     write(*,*) 'Input a positive integer (less than 10): '
     read(*,*) increment

     if ( increment <= 0 ) then
        write(*,*) 'error : input <= 0'
        exit   ! ループを抜ける
     else if ( increment >= 10 ) then
        write(*,*) 'error : input >= 10'
        cycle  ! ループの先頭へ(これ以下の処理は行わない)
     end if
```

(次のページに続く)

(前のページからの続き)

```
    ! count を増やす
    count = count + increment

    write(*,*) 'current count = ', count
  end do
```

この例では標準入力からの 10 より小さい正の整数を受け取り，受け取った数の和を求める．ただし 0 以下の値を受け取った場合は処理を終了する．また，10 以上の値受け取った場合はエラーを表示し，和はとらずに無視する．実行結果は例えば以下のようになる．

```
                    (          10)
5                                   # キーボード入力
         count =            5

                    (          10)
4                                   # キーボード入力
         count =            9

                    (          10)
10                                   # キーボード入力
               = 10

                    (          10)
0                                   # キーボード入力
               = 0
      count =            9
```

なお，同じ動作を実現する方法は 1 つとは限らない．例えば `while` の条件判定を `increment <= 0` と変更すれば，この条件に対応する `if` は必要がなくなる．複数の方法がある場合にはより分かりやすい方 (すなわち間違いが発生しにくい方) を採用すれば良い．ちなみに無限ループを作るには `while (.true.)` は必ずしも必要では無く，

```
do
  ! exit で抜けないかぎりここの処理が無限に繰り返される
end do
```

のように書くことも可能である．

4.3 条件分岐 (select)

select 構文を用いても条件分岐を行うことも出来る．基本的には if を用いれば同じことは実現出来るのだが，場合によっては select を用いた方がよりスッキリとした形で書ける事があるので知っておいて損はない．典型的には整数や文字列の値で場合分けを制御する際に用いる (実数型には用いることは出来ない)．

一番基本的な使い方は以下のようなものである．

リスト 8: sample5.f90 抜粋

```
! 整数を読み込む
write(*,*) 'Input integer : '
read(*,*) i

! 整数の値で場合分け
select case(i)
case(0)         ! i == 0 のとき
   write(*,*) 'your input was zero'
case(1)         ! i == 1 のとき
   write(*,*) 'your input was one'
case(2)         ! i == 2 のとき
   write(*,*) 'your input was two'
case(3)         ! i == 3 のとき
   write(*,*) 'your input was three'
case default    ! 上記以外全て
   write(*,*) 'your input was too large'
end select
```

case で入力された整数の値に応じて処理を切り替えている．また，どの case の条件にも当てはまらない場合の処理は case deafault によって行えば良い．

この例を見ただけでは if-elif-else を使うのとほとんど変わらないように感じられるので，もう少し select の利点が生かされた例として以下を見てみよう．ここでは入力された整数 score (テストの点だと思おう) の値によって場合分けをしている．

リスト 9: sample5.f90 抜粋

```
! 点数を読み込む
write(*,*) ''
write(*,*) 'Input score : '
read(*,*) score

select case(score)
case(0)              ! 0点
   write(*,*) 'zero'
case(1:29)           ! 1-29点
   write(*,*) 'poor'
case(30:59)          ! 30-59点
```

(次のページに続く)

(前のページからの続き)

```
    write(*,*) 'fair'
  case(60:89)           ! 60-89点
    write(*,*) 'good'
  case(90:100)          ! 90-100点
    write(*,*) 'excellent'
  case default          ! それ以外
    write(*,*) 'invalid input'
  end select
```

この例のように case では単一の値だけでなく値の範囲を指定することができる．範囲の指定は case(下限:上限) のような形ですれば良い．

さらに

リスト 10: sample5.f90 抜粋

```
! 文字列を読み込む
write(*,*) ''
write(*,*) 'Input language : '
read(*,*) c

! 文字列の値で場合分け
select case(c)
case('c', 'c++', 'fortran')
   write(*,*) 'compiled language'
case('python', 'perl', 'ruby')
   write(*,*) 'script language'
case('english', 'japanese', 'french', 'chinese')
   write(*,*) 'natural language'
case default
   write(*,*) 'others'
end select
```

は文字列の値によって分岐する例である．このように 1 つの case で複数の値をカンマで区切って指定することも出来る．

4.4 第 4 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例
- 課題 6 解答例
- 課題 7 解答例

4.4.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

4.4.2 課題 2

標準入力から 2 つの整数 (n, m とする) を読み込み，その大小を比較するプログラムを作成せよ．例えば $n = 1, m = 2$ なら以下のように `1 is smaller than 2` と表示する．

```
1    # キーボード入力
2    # キーボード入力
          1                              2
```

同様に $n = 2, m = 1$ なら `2 is larger than 1`, $n = m = 1$ なら `1 is equal to 1` などと表示するものとする．

4.4.3 課題 3

0° から 180° まで 10° 刻みの θ および， $\sin\theta, \cos\theta$ を標準出力に表示するプログラムを作成せよ (以下のように各 θ の値ごとに改行せよ)．またこの結果をリダイレクトを用いてファイルとして記録し，gnuplot を用いてこのファイルのデータと gnuplot に組み込みの三角関数を共に図示せよ．なお三角関数の引数はラジアン単位であることに注意せよ．

実行結果は例えば以下のようなものになる．

```
0                    0                    1
10                   0                    0

170                  0
180                   1
```

以下のようにリダイレクトでデータファイルを作成した場合には

```

```

gnuplot では

```
        'data.dat'       1            (        )
        'data.dat'       1            (        )
```

などとして結果を確認すればよい．

4.4.4 課題 4

標準入力から与えられた 2 つの整数 $m, n \geq 1$ の最大公約数を表示するプログラムを作成せよ．最大公約数を求めるには以下のアルゴリズム (ユークリッドの互除法) を用いるとよい．

1. m を n で割った余り r を求める．
2. もし $r = 0$ ならば n が最大公約数である．
3. もし $r \neq 0$ ならば， m に n を， n に r を代入して [1] に戻る (繰り返す).

実行結果は例えば以下のようなものになる．

```
12
20
                                    4
```

なお，組み込み関数 mod を用いて

```
r = mod(m, n)
```

とすれば m を n で割った余りを r に代入することが出来る．

4.4.5 課題 5

以下の級数計算により自然対数の底 e の近似値を求めるプログラムを作成せよ．

$$e \simeq \sum_{n=0}^{N} \frac{1}{n!}. \quad (0! = 1 \text{ に注意せよ})$$

ただし以下の条件を満たすこと．

- 上式の N および許容誤差 ϵ を標準入力から読み込む．
- $N > 1$ でない場合および $0 < \epsilon < 1$ でない場合にはエラーメッセージを表示して終了する．
- 誤差が ϵ 以下になった時点か， $n = N$ まで計算した時点で級数計算を打ち切る．
- 最後に収束したかどうか，最終的な項数 n ，真値，近似値，相対誤差を表示して終了する．

実行結果は例えば以下のようなものになる．

```
10          # キーボード入力
1           # キーボード入力

                          10
                    2
                    2
                   1
```

4.4.6 課題 6

標準入力から文字列 (英単語) を読み込み，それが food, animal, vehicle, others (それ以外) のいずれかを判定し，表示するプログラムを作成せよ．ただし exit が入力されるまでプログラムは終了せず何度でも入力を受け付けるものとする．なお以下の英単語リスト以外のものは others と判断してよい: apple, orange, banana, dog, cat, lion, car, airplane, motorcycle .

実行結果は例えば以下のようなものになる．

```
        # キーボード入力
        # キーボード入力
        # キーボード入力
        # キーボード入力
        # キーボード入力
        # キーボード入力
exit    # キーボード入力
    exit
```

4.4.7 課題 7

以下の漸化式

$$p_{n+1} = p_n + \alpha p_n(1 - p_n)$$

で定義される数列 $p_n (n = 0, 1, \ldots)$ を考える．初期値 $p_0 = 0.9$ から数列を生成し，そのうち $n = 100, \ldots, 200$ までを α の関数として $1 < \alpha < 3$ の範囲でプロットせよ． α を 10^{-3} 刻みで変えながらプロットすると結果は以下のようになるだろう．

このような写像はロジスティック写像と呼ばれ，非常に単純な式ながら一定の条件を満たすときにはカオスを生み出すことが知られている．

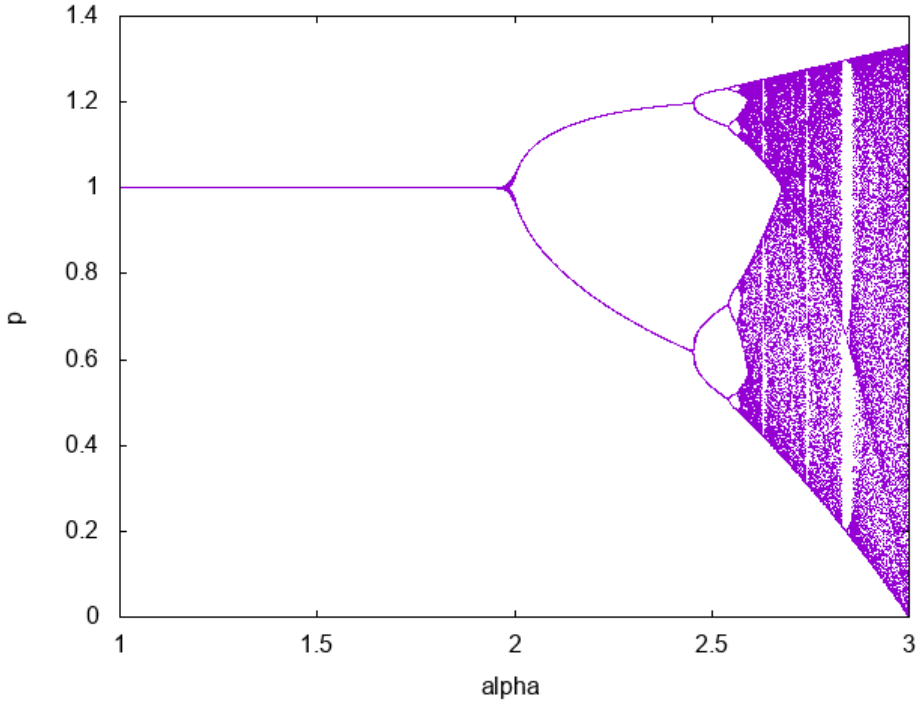

図 1: ロジスティック写像

第5章　配列

大量のデータをまとめて扱うのに便利な配列について，その基本的な使い方や配列に関する組込み関数の使い方などを学ぼう．

参考:

- sample1.f90 : 配列の基本
- sample2.f90 : 配列の定数と初期化
- sample3.f90 : 動的配列
- sample4.f90 : 多次元配列
- sample5.f90 : 配列の入出力 1
- sample6.f90 : 配列の入出力 2
- sample7.f90 : 配列に関する組込み関数
- sample8.f90 : 部分配列と配列演算

この章の内容

5.1 基本的な使い方

配列とは **同じ型** の複数のデータを効率的に扱うために用いるデータ構造[1] である．配列も通常の変数と同じように宣言が必要であり，宣言時には配列であることを明示的に示さなければならない．配列を宣言すると計算機の **メモリ上の連続した領域** が確保され，それぞれのアドレスに添字を用いてアクセスできるようになる．

具体的には以下のように配列を宣言する．

[1] 計算機の中でデータの塊を扱う形式のことを一般にデータ構造と呼ぶ．配列は最も単純なデータ構造の一つと考えることが出来る．

リスト 1: sample1.f90 抜粋

```
! 最も基本的な配列の宣言
integer :: a(5)

! 配列の添字範囲を指定して宣言
integer :: b(0:4)
integer :: c(6:10)
```

7 行目が最も基本的な (ここでは整数型の) 配列の宣言であり，この場合は長さ 5 の a という名前の配列を宣言している．この例では a(1) が最初の要素であり， a(5) が最後の要素ということになる．C 言語を始めとする多くの言語では配列の添字は 0 から始まることになっているので注意して欲しい．ただし Fortran では，10-11 行目のような宣言によって宣言時に配列の添字の範囲を指定することができる．これらの例はどちらも長さ 5 の配列を宣言しているが， b はから 0 から 4 まで， c は 6 から 10 までが正しい配列の添字の範囲である．配列の要素にアクセスするには

リスト 2: sample1.f90 抜粋

```
! do ループで配列の各要素を処理する
do i = 1, 5
   a(i) = i
end do

do i = 0, 4
   b(i) = i
end do

! 各要素同士の演算も出来る (添字に注意)
do i = 1, 5
   c(i+5) = 2*a(i) + b(i-1)
end do
```

のように () で添字を指定すればよい．ここで， b や c は添字の範囲が異なっていることに注意しよう．

また，以下は長さ 100 の配列 x の総和を計算する例である．

リスト 3: sample1.f90 抜粋

```
! 配列の和を求める
sum = 0.0_8
do i = 1, 100
   sum = sum + x(i)
end do
```

このように配列の各要素に対する処理には do ループを用いる事になる．

5.2 配列の定数と初期化

配列は宣言する時に同時に初期化することも可能である．例えば

リスト 4: sample2.f90 抜粋

```
integer :: a(5) = (/1, 2, 4, 8, 16/)
```

のようにすればよい．ここで宣言した配列の長さと右辺の要素数は同じになっていなければならない．これを用いると parameter 属性を付けて定数配列を宣言することも出来る．

リスト 5: sample2.f90 抜粋

```
integer, parameter :: b(3) = (/-1, 0, 1/)
```

通常の定数変数と同じように，定数として宣言された配列は参照は出来るが値の変更は出来ないようになっている．

配列名を指定せずに無名の定数配列を作ることも出来る．これには"(/ "と"/)"で全体を，", "で各要素を区切って記述する．例えば以下の例では長さ 3 の定数配列を出力する．

リスト 6: sample2.f90 抜粋

```
write(*,*) (/1, 2, 3/)
```

5.3 動的割付け

通常の配列はコンパイル時に静的に配列のサイズが決定される．予め必要な領域 (メモリ) サイズが分かっていればこれで良いのだが，実行してみるまで必要な領域サイズが分からない場合にはこれでは対処できない．このような時には allocatable 属性を用いることで，実行時に動的に配列用にメモリを割り付けることができる．具体的には以下のように allocatable な配列を宣言すればよい．

```
integer, allocatable :: x(:)
```

ここでは整数型の x という配列を宣言しているが，その長さはコンパイル時には不定 (実行するときまで分からない) の で x(:) のように長さが指定されていないことに注意しよう．これを配列として使う前には

```
allocate(x(100))
```

のように allocate 関数によって，メモリを割り付ける必要がある．これによって，以降は x は (この場合は長さ 100 の) 配列として使うことができる． allocate で確保したメモリは使い終わったら

```
deallocate(x)
```

のように deallocate で開放してやるのが作法である．いわゆるメモリリークという厄介なバグはこのような動的に割り付けたメモリの解放忘れによって発生するので気をつけよう[2]．

なお，メモリが既に割りつけられているかどうかを確認するために allocated という関数も用意されている．この関数はメモリが割り付けられている場合には真を返す．例えば

リスト 7: sample3.f90 抜粋

```
! 動的配列 (実行時にしかサイズが分からない場合)
integer, allocatable :: x(:)

! 以下は動的 (allocatable) 配列の使い方
write(*,*) 'Input array size: '

! 配列サイズ
read(*,*) n

! allocate されていないことを確認してから allocate
if( .not. allocated(x) ) then
   allocate(x(n))
else
   write(*,*) 'Error: already allocated'
end if

! 確かに allocate されたか?
if( allocated(x) ) then
   write(*,*) 'Successfully allocated'
end if
```

のように使うことができる．これは標準入力から与えられた整数を長さとする配列を割り付ける例である．単に allocate するだけでなく，その前に allocated で既にメモリが割り付けられているかどうか確認している．この場合はプログラムの全体像がひと目で分かる規模のためこの確認作業は冗長であるが，場合によってはこのようなチェックが必要なこともあるだろう．

[2] プログラムが終了する際には当然全てのメモリが解放されるので必要以上に心配する必要は無い．また，Fortran 95 以降では allocatable な配列は，スコープから外れた時 (後述のサブルーチンなどから出た時) には自動的に deallocate されるということになったようである．従って通常はあえて deallocate しなくても良いかも知れない．ただし，一般的に借りたものは必ず返すというのがプログラミングでは礼儀になっているので，ちゃんと deallocate するように癖をつけておいた方が無難である．例えば C 言語では malloc などでメモリを割り付けた場合は free で明示的に解放しない限りプログラム終了までメモリを保持し続ける．

5.4 多次元配列

ここまで扱った配列は 1 次元配列と呼ばれるものであったが，多次元の配列も使うことができる．分り易い例として 1 次元配列はベクトル，2 次元配列は行列と考えればよいだろう．多次元配列の宣言には次元の分だけ (各次元の) 長さを指定すれば良い．具体的には以下のような宣言となる．

リスト 8: sample4.f90 抜粋

```
! 2 次元配列 (10 x 10 => 計 100 要素)
real(8) :: a(10,10)

! 3 次元配列 (4 x 8 x 16 => 計 512 要素)
real(8) :: b(4, 8, 16)

! 動的配列も同様に宣言できる
real(8), allocatable :: c(:,:)

! 動的配列: 4 x 8
if( .not. allocated(c) ) then
   allocate(c(4,8))
end if
```

7 行目は 2 次元配列，10 行目は 3 次元配列を宣言している．このように多次元配列を宣言するには各次元の長さをカンマ区切りで指定すればよい．13 行目のように多次元の動的配列も宣言することができるが，次元だけはあらかじめ指定しなければならないので `c(:,:)` のように次元の数だけ `:` をカンマ区切りで指定する．多次元の動的配列にメモリを割り付けるには 17 行目のように `allocate` の際に次元の数だけ長さを指定する必要がある．（このように多次元配列の次元数はコンパイル時に決定され，実行時には変更できない．）

宣言した配列には，次元の数だけ添字を指定して各要素にアクセスすればよい．例えば

リスト 9: sample4.f90 抜粋

```
do j = 1, 8
   do i = 1, 4
      c(i,j) = i*j
   end do
end do
```

のような形である．

なお配列の次元数を rank(次元)，各次元の要素数の組を shape(形状)，全要素数を size(サイズ) などと呼ぶことが一般的である．これらの言葉の意味は次の表を見てもらえばすぐに理解出来るであろう．

表 1: 配列宣言の例

配列宣言	rank (次元)	shape (形状)	size (サイズ)
`a(10)`	1	(10,)	10
`b(2, 5)`	2	(2, 5)	10
`c(10,10,10)`	3	(10,10,10)	1000
`d(0:9,0:99)`	2	(10, 100)	1000

5.5 配列の入出力

配列データの入出力についてもこれまでと同様に各要素を `read(*,*)` や `write(*,*)` に対する入出力リストとして与える方法もあるが，例えば配列全体を入出力リストとして与えることなども出来る．詳細は **ファイル入出力** で説明するが，ここではとりあえずアスキー形式 (人間の目で読める形式) のことだけを考えることにする．

5.5.1 入力

配列の読み込みには少し注意が必要である．

リスト 10: sample5.f90 抜粋

```
integer :: i
real(8) :: x(10), y(10), z(10)

! do ループで全要素を順に読み込む
! この場合は改行があっても構わない
read(*,*) x

! このように書いても動作は同じ
read(*,*) (y(i), i = 1, 10)
```

(次のページに続く)

(前のページからの続き)

```
! よく理解していない場合にはこの形式は使わない方がよい
! 1 行に一つの要素が書かれている場合の読み込みはこれでよい
do i = 1, 10
   read(*,*) z(i)
end do
```

これは標準入力から 3 つの長さ 10 の配列 x, y, z のデータを読み込む例である．ここで与えるデータの改行の扱いに注意が必要である．ここでは入力として， sample5.dat をリダイレクトを用いて

```
```

のように与えることを想定している．

12 行目の read は以下の sample5.dat の 1 行目のデータを全て読み込む．ここで改行や空白，カンマ，タブなどは Fortran が自動的に無視して，数値だけを読み込んでいることに注意しよう．

リスト 11: sample5.dat 抜粋

```
1    2    3    4    5    6    7    8    9    10
```

15 行目の read は以下の sample5.dat の 3-12 行目のデータを全て読み込む．

リスト 12: sample5.dat 抜粋

```
0
0
0
0
0
0
0
0
0
1
```

最後に 19-21 行目の do ループ中の read は

リスト 13: sample5.dat 抜粋

```
1
1
1
1
1
1
1
1
```

(次のページに続く)

(前のページからの続き)

```
1
2
```

を読み込んでいる．ここで注意しなければならないのは 19-21 行目のような do ループと read を組み合わせた読み方では

```
1    1    1    1    1    1    1    1    1    2
```

のような改行なしの形式のデータをうまく読み込むことができない，ということである[3]．

あまり細かいことを考えたくない人は特に理由がない限りは sample5.f90 の 12 行目，もしくは 15 行目のように **一文で配列データを全て読む** (do ループは使わない) ようにするのが無難である．

5.5.2 出力

配列の出力の仕方を見てみよう．

リスト 14: sample5.f90 抜粋

```
! do ループで全要素を順に出力
do i = 1, 10
   write(*,*) x(i)
end do

write(*,*) x                    ! 改行せずに 1 行に全要素を出力
write(*,*) (y(i), i = 1, 10)    ! これも同じ
write(*,*) (z(i), i = 1, 10, 2) ! 1 つ飛ばしで出力
```

24-26 行目のように do ループを用いて行ってもよいが，28 行目や 29 行目のように 1 行で出力することもできる．違いは write を一度呼び出しするごとに改行が挿入されるという点だけである．また 30 行目のように 1 つ飛ばしで出力することも可能である．

5.5.3 多次元配列について

多次元配列の読み込みについては少し注意が必要である．例えば以下の sample6.dat

リスト 15: sample6.dat

```
 1    2    3
 4    5    6
 7    8    9
10   11   12
```

を 2 次元配列として読み込む例を考えてみよう．

[3] どうやら read は改行までを一区切りとして読み込むようである．これが Fortran の標準なのか gfortran 独自の仕様なのかは不明である．

リスト 16: sample6.f90 抜粋

```
  real(8) :: x(3,4)

  ! 配列のメモリが連続した要素に順に読み込まれる
  read(*,*) x
```

ここでもリダイレクトを用いて

```
```

のように読み込むことを想定している．11 行目の `read` で 3x4 の 2 次元配列 `x` にデータを読み込んでいることが分かるであろう．

ここで，sample6.dat の **見た目** は 4x3 の行列のように見えるのに対して，Fortran では 3x4 の 2 次元配列を宣言して読み込んでいることに気をつけよう．このプログラムを実行すると，`x(1,1)`，`x(2,1)`，`x(3,1)`，`x(1,2)`，`...` にそれぞれ `1.0`，`2.0`，`3.0`，`4.0`，`...` が代入されることになる．これは入力が先頭から順々に行われることと，Fortran の多次元配列のメモリ並びがこの順番になっているためである (メモリ並びについては *Column major* と *Row major* 参照).

配列の形状が何であってもかならずこの順番で読み込まれるため，例えば

```
real(8) :: x(2,6)

read(*,*) x
```

であれば，`x(1,1)`，`x(2,1)`，`x(1,2)`，`x(2,2)`，`...` の順で `1.0`，`2.0`，`3.0`，`4.0`，`...` が代入されてしまう．このように多次元配列の読み込みは (初心者にとっては) 必ずしも意図する結果にならないことがあるので注意して欲しい．配列はあくまで規則的にデータを並べただけのものであり，数学的な行列の概念とは必ずしも一致しないのである．

なお，ここでも `read` 一文でデータを全て読み込まなければならないことに再度注意しよう．以下のような 2 重ループ

```
integer :: i, j
real(8) :: x(3,4)

! 注意: これは動かない !
do j = 1, 4
  do i = 1, 3
    read(*,*) x(i,j)
  end do
end do
```

では正しく読み込むことが出来ない．

5.6 配列に関する組込み関数

Fortran にはいくつか配列に関する便利な組込み関数が用意されている．細かい使い方についてはサンプルコードや自分で実際にコードを書いてみて動作確認をしてみるのが一番の近道である．

例えば以下の例では行列とベクトルの積を計算する `matmul` ，およびベクトル同士の内積を計算する `dot_product` の使い方を示している．（ここでは `a` は 2 次元配列， `b` および `x` は 1 次元配列である．）

リスト 17: sample7.f90 抜粋

```
! 行列 a とベクトル x の積を b に代入: b_{i} = a_{i,j} * x_{j}
do j = 1, n
  do i = 1, n
    b(i) = b(i) + a(i,j) * x(j)
  end do
end do

write(*,*) 'b = ', b

! 組み込み関数を使用して同じ計算を行う
b = matmul(a, x)

write(*,*) 'b = ', b

! ベクトル同士の内積を計算
inner = 0.0_8
do i = 1, n
  inner = inner + b(i) * x(i)
end do

write(*,*) 'inner product 1 = ', inner

! 組み込み関数を使用して同じ計算を行う
inner = dot_product(b, x)

write(*,*) 'inner product 2 = ', inner
```

この例では 29-33 行目と 38 行目はどちらも行列とベクトルの積を求めるものである．同様に 43-46 行目と 51 行目も全く同じ処理 (内積計算) を行なっている．組込み関数を用いることで非常に簡単に処理が記述できることが分かるだろう．数学関数に加えてよく使われる組み込み関数をいくつか以下の表に挙げておこう．念のために言うとこれらは必ずしも記憶して置かなければいけないものでは無く，必要になった時に自分で調べて使いこなすことが出来ればそれで良い．(例えば富田・齋藤 (2011，6 章) が配列に関する組み込み関数について詳しい．)

表 2: 配列に関する組み込み関数の例

関数名	説明
dot_product(x, y)	ベクトル (1 次元配列) x と y の内積を返す
matmul(x, y)	行列 (2 次元配列) 同士，もしくは行列とベクトル (1 次元配列) の積を返す
transpose(x)	行列 (2 次元配列) の転置を返す
sum(x)	配列 x の各要素の和を返す
product(x)	配列 x の各要素の積を返す
size(x)	配列 x の全要素数 (サイズ) を返す
shape(x)	配列 x の形状を 1 次元の整数型配列として返す
reshape(x, s)	配列 x の形状を新しい形状 s に変換したものを返す
maxval(x)	配列 x の全要素の最大値を返す
minval(x)	配列 x の全要素の最小値を返す

なお reshape を使うと多次元の配列定数を初期化することが出来る．以下はその例である．

```
integer, parameter :: x(2,3) = reshape((/1, 2, 3, 4, 5, 6/), (/2, 3/))
```

reshape の第 1 引数は任意の配列であり，この配列の形状を変更したものを返す．第 2 引数には新しい配列の形状を指定している．ここでは左辺の配列の形状が (2,3) であるので reshape の第 2 引数は (/2, 3/) と形状を 1 次元の整数配列として指定している．当然，元々の入力配列のサイズと新しい配列のサイズは同じでなければならない[4]．

5.7 部分配列

これまでは各要素に添字を用いて例えば x(10) のような形でアクセスしていた．Fortran ではこれに加えて **部分配列** という便利な機能があり，配列の複数の要素にまとめてアクセスすることが出来る．これには添字の代わりに x(lower:upper:stride) のような形式を用いる．lower，upper，stride の意味は do 変数の指定方法 (**決まった回数の繰り返し** *(do)*) と同じである．従って例えば

```
integer :: x(10) = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)

write(*,*) x(1:10:2) ! 1, 3, 5, 7, 9 が出力される
```

のように書くことが出来る．lower，upper，stride などは省略することも出来，その場合は lower は配列の最初の要素，upper は最後の要素，stride は 1 と解釈される．ただし stride はともかく lower，upper は明示的に書いておいた方が分かりやすい．またこれらの指定に変数を使う事もできる．

[4] このように配列形状を変更できることを不思議に思うかもしれない．しかし，実際には 1 次元配列も多次元配列も中身は同じ 1 次元的なメモリ領域を指しており，使う側には便宜上違う次元のもののように見えているだけなのである．詳しくは *Column major と Row major* を参照のこと．

5.8 配列演算

さらに，Fortran には非常に強力な **配列演算** という機能が用意されている．例えば

リスト 18: sample8.f90 抜粋

```
real(8) :: a(n), b(n), c(n)
```

のように定義された配列 a， b， c に対して，以下のような処理を行なう．

リスト 19: sample8.f90 抜粋

```
! 代入
do i = 1, n
   b(i) = a(i)
end do

! 配列演算による代入 (上の do ループと同じ)
b = a

write(*,*) 'b = ', b

! 演算
do i = 1, n
   c(i) = 0.5_8*a(i) + cos(b(i))
end do

! 配列演算 (上の do ループと同じ)
c = 0.5_8*a + cos(b)

write(*,*) 'c = ', c
```

ここで，上の例の 27-29 行目と 32 行目，37-39 行目と 42 行目はそれぞれ等価である．このように Fortran では **配列同士の演算をあたかも通常の変数であるかのように記述することができる** ．これを配列演算と呼ぶ．数学で用いるような直感的な表現が出来ることに加えて，これを用いることでかなりタイプ量を減らすことができるのが一目見て分かるだろう．タイプ量が少ないと当然無用なバグの混入を避けることができる．さらに，配列演算はコンパイラによる最適化の恩恵を受けやすいという利点がある．

部分配列と配列演算を組み合わせることも当然可能である．例えば

リスト 20: sample8.f90 抜粋

```
real(8) :: x(m), y(m/2)
```

のように定義された配列 x， y に対して

リスト 21: sample8.f90 抜粋

```
y = 2*x(1:m:2) + 1
```

のような記述ができる．部分配列や配列演算の機能は多次元配列に対しても同様に使用することができるが，配列演算は **同じ形状 (次元およびサイズ) の配列に対してしか行うことが出来ない** ことに注意しよう．それ以外の場合には演算が定義されないのでこれは当たり前の話である．

また，数学におけるベクトルの内積やベクトルと行列の積の計算規則とは異なり，配列演算はあくまで各要素ごとの演算であるという点に注意しよう．例えば `x(100)` と `y(100)` のような 2 つのサイズの等しい 1 次元配列の積 `x*y` は同じサイズ 100 の配列となり，スカラー値を計算する内積の計算規則とは異なる．また行列 `M(100,100)` とベクトル `x(100)` の積を計算しようとして `M*x` と記述しても `M` と `x` は形状が異るのでエラーとなってしまう．このような場合は先に見た `dot_product` や `matmul` を使えば良い．

5.9 補足 †

5.9.1 メモリ領域

Fortran の通常の静的配列 (static array) の場合はメモリはスタック (stack) と呼ばれる領域に保持される．環境によっては (おそらく多くの Linux 環境のデフォルトでは) スタックに大きなメモリ領域を保持できないようになっている．この設定は例えば sh 系のシェル (bash など) では以下のように `ulimit` コマンド (csh 系のシェルならば `limit`) で確認することが出来る．

```
ulimit
                    (          ) 0
                    (          )
                    (          )
                    (          )
                    (          )
                           (  ) 256
                 (512          ) 1
                    (          ) 8192
  time              (          )
                           (  ) 709
                    (          )
```

上の `ulimit` コマンドの出力結果から，この環境ではスタック領域が 8MB に制限されているので大きな静的配列を確保することが出来ないことが分かる．プログラムの実行直後に原因不明の `Segmentation fault` などのエラーで終了してしまう場合はスタック領域が足りずにメモリが確保出来なかったことが原因かもしれない．

どうしても静的配列を使いたい場合には `ulimit` コマンドで使用可能なスタック領域を増やせば良い．もしくは静的配列の使用をやめて `allocatable` 配列を用いるようにすればスタック領域の制限は受けない．これは `allocatable` 属性付きで宣言された配列のメモリは (`allocate` によって) ヒープ (heap) と呼ばれる別の領域にメモリが確保されるためである．なおスタックとかヒープについて必ずしも理解している必要は無いが，原因不明のエラーが発生した時にはこのことをふと思い出して欲しい．

5.9.2 Column major と Row major

既に説明したように配列は計算機の連続したメモリ上に確保されることが保証されている．これは 1 次元の場合には分かりやすいが，多次元配列の場合はどうなっているのであろうか？計算機のメモリは 1 次元的なアドレスからなっているので，実は多次元配列であってもメモリは内部的には 1 次元的に連続な領域を指している．多次元配列は単にそれらを使いやすく表示したものに過ぎない．一般的に Fortran では例えば 2 次元配列 `x(10,10)` の場合は `x(1,1)`, `x(2,1)`, ..., `x(10,1)`, `x(1,2)`, `x(2,2)`, ... のような並び，すなわち配列の一番左の添字がメモリの連続した方向となっている．これ を column major と呼ぶ．これに対して C 言語などでは row major と呼ばれるメモリ並びが採用されており一番右側の添字がメモリの連続する方向となっている (図参照)．従って，C 言語で書かれたライブラリを Fortran から呼び出す際 (もしくはその逆) にはこの違いに注意しなければならない．

またこのことから，効率的なプログラムとするためには多次元配列のループの書き方も注意が必要である．以下の例を考えてみよう．

```
integer :: i, j
real(8) :: a(10,10), s



! 例 1
s = 0.0_8
do j = 1, 10
  do i = 1, 10
    s = s + a(i, j)
  end do
end do

! 例 2
s = 0.0_8
do i = 1, 10
  do j = 1, 10
    s = s + a(i, j)
  end do
end do
```

この例では 5-11 行目 (例 1) と 13-19 行目 (例 2) は全く同じ処理 (配列内の全要素の総和計算) を行なっているが，多重 do ループの添字の順番が異なることに注目して欲しい．例 1 では左側の添字 `i` が内側のループで走り，例 2 では右側の添字 `j` が内側のループで走っている．基本的に計算機というのは単純作業 (例えば `if` 分岐などがないループ) を一気に，メモリの連続している方向に順番に処理するのが得意になっている．従って，この例では左側の添字が内側ループで走る例 1 の方が効率の良いプログラムということになる[5]．最初はそれほど気にすることは無いが，単に「動く」だけのプログラムでは無く，「良い」プログラムとなるように細かい点についても気を配れるようになって欲しい．

[5] 実際にはプログラムの構造やループ内でのメモリ使用量，CPU やコンパイラの性能に大きく依存する (かしこいコンパイラはループの順序を交換したりすることもある)．またこの程度の小さな配列ではほとんど差が見られないであろう．

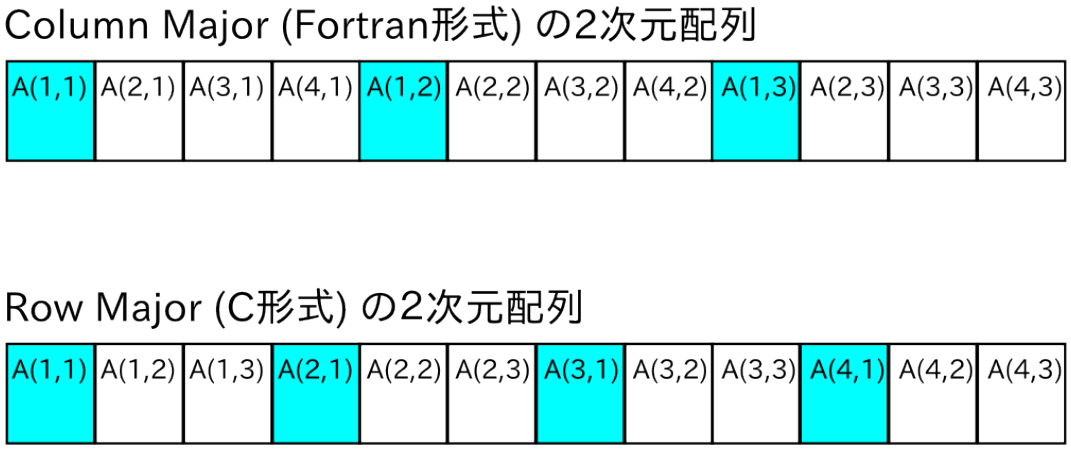

図 1: Column major と Row major．メモリは左から右に連続的に並んでいる．(C 言語の場合は実際には配列添字は 0 から始まり，添字も [] で指定することに注意．)

5.9.3 配列境界チェック

配列の添字の範囲をはみ出した場合には何が起こるだろうか? 実はこの時何が起こるかは実行してみるまで分からない．何事も無かったかのように正常終了するように見える場合もあるし，"Segmentation fault"などのエラーが表示されて異常終了することもある．1 つだけ言えることはそのようなプログラムは例え正しく動いているように見えたとしてもかなり危険な状態である．なぜならプログラムで自分が「使いたい」と要請したメモリ領域とは異なる領域へアクセスしていることになるので，自分のプログラムで用いているメモリ領域はおろか，OS がプログラムの実行に必要とする情報 (コールスタックなどと呼ばれる) をも意図せず書き換えてしまうかもしれない．異常終了しなかったとしても，それはたまたま運が良かっただけなの話である．たった 1 行ソースコードを書き換えただけでも，プログラム中のメモリ配置が変わることで動作がおかしくなるかもしれない．(1 行 `write` 文を入れるかどうかだけの違いで動作が変わるような場合もあるが，そういう時には大抵おかしなメモリ領域にアクセスしているものである．)

そもそも配列の添字範囲をはみ出すのは明らかなバグである．通常は効率を重視するため配列添字の境界チェックは行われないが，gfortran ではコンパイル時に `-fbounds-check` というオプションをつけることでこの配列境界チェックを行うことが出来る．(多くの Fortran コンパイラが同じようなオプションを有しているので他のコンパイラを用いる時にはチェックしてみて欲しい．) これによってもし境界をはみ出した場合にはその旨エラーが出力されてプログラムが終了する．

```
program check
  implicit none

  integer :: i = 11
  integer :: x(10)

  x(i) = 1

  stop
end program check
```

例えば上のソースコードを check.f90 として保存し，コンパイル・実行した結果は以下のようになる．

```
       7
                          '11'             1          'x'                    10
```

配列 x の上限 (10) を超えた 11 番目の要素にアクセスしているのでエラーが表示されているのが分かる．ただし，このようなチェックを逐一行うことで，当然実行時のパフォーマンスは犠牲になる．従って，デバッグの段階でこのような配列境界チェックを行い，時間のかかる計算を実行する際にはこのオプションは外しておこう．

5.10 第 5 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例
- 課題 6 解答例
- 課題 7 解答例

5.10.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

5.10.2 課題 2

与えられた月日 (例えば 4 月 1 日であれば 4 と 1) を標準入力から読み込み，その日が 1 年のうちで何日目かを表示するプログラムを作成せよ．ただし閏年は無視して考えて良い．以下のような配列を用いるとよいだろう．

```
integer, parameter :: days(12) = &
     & (/31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31/)
```

実行結果は例えば以下のようなものになる．

```
4     # キーボード入力
1     # キーボード入力
                         91
```

5.10.3 課題 3

学生のテストの点数を自動的に処理するプログラムを作成せよ．すなわち，標準入力から学生の人数および人数分のテストの点を順に読み込み，最高点，最低点，平均点，標準偏差をそれぞれ表示するプログラムを作成せよ．ただし標準偏差はデータ数 N ，各データの値 $x_i (i = 1, \ldots, N)$ ，平均値 $\bar{x}$ を用いて

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

と定義される．

データファイル score1.dat を手元にコピーして，以下のようにリダイレクトによって作成したプログラムに読み込ませ，結果を確認せよ．実行結果は例えば以下のようなものになる．

```
                98
                 6
        46
        25
```

なおデータファイルには，1 行目にデータ数 N ，それ以降に各データ x_i が記述されているので，まずはデータ数を読み込み配列のメモリを `allocate` した後に各データを読み込めば良い．(sample3.f90 を参照せよ．)

5.10.4 課題 4

標準入力から 2 つのベクトルを読み込み，両者の内積を計算し表示するプログラムを作成せよ． do ループを用いて地道に計算した結果と組込み関数 `dot_product` を用いた結果を比較すること．

以下はデータファイル vector.dat を入力とした場合の結果である．

```
        do                  5
                            9
```

ただしデータは，ベクトルの長さ N ，1 つ目のベクトルの要素 (N 個)，2 つ目のベクトルの各要素 (N 個)，の順に並んでいるものとする．

5.10.5 課題 5

標準入力からベクトルと行列を読み込み，積を計算して表示するプログラムを作成せよ．これについても 2 重 do ループを用いて地道に計算した結果と，組込み関数 `matmul` を用いた結果を比較すること．

以下はデータファイル matvec.dat を入力とした場合の結果である．これと同じ結果が得られることを確認せよ．

```
                    do


 0

 1
 1




 0

 1
 1
```

データは，ベクトルの長さ N ，ベクトルの要素 (N 個)，行列の各要素 (N^2 個)，が順に並んでいるものとする．また行列の要素は $a_{11}, a_{21}, a_{31} \ldots$ の順に読み込まれることと，ベクトルと行列の積 $b_i = \sum_j a_{i,j} x_j$ の添字の順番に注意せよ．

matvec.dat は以下のようなファイルになっているが，行列の部分を Fortran で読み込むとあたかも転置行列を読み込んだような形になることに注意せよ．（実際に読み込んで確かめてみよ．）

```
8

 0
 1
 1
 0
 0


 0

       0     0     0     0     0     0     0
 1           1     0     0     0     0     0
 0     1           1     0     0     0     0
 0     0     1           1     0     0     0
 0     0     0     1           1     0     0
 0     0     0     0     1           1     0
 0     0     0     0     0     1           1
```

(次のページに続く)

(前のページからの続き)

```
 0  0  0  0  0  0  2
```

注釈: プログラムの入力は数学的な「ベクトル」や「行列」を読んでいる訳ではなく，単なる数値の羅列を決められた順番で読み込む．それをどのように「ベクトル」や「行列」として解釈するのかはプログラムを書く人間が決めることである．(**配列の入出力** を理解するまで熟読せよ．)

5.10.6 課題 6

標準入力から与えられた整数 $n(\geq 2)$ 以下の全ての素数 (1 は素数に含めない) を表示するプログラムを作成せよ．以下のエラトステネスのふるいと呼ばれるアルゴリズムを用いるとよい．

各整数 $i = 2, \ldots, n$ について順に

- i が素数でなければ無視 ($i+1$ の処理へ)
- i が素数であれば i から n の整数のうち i の倍数のものを消去 (素数以外と判定)

の処理を行う．なお各整数が素数かどうかを判定するには長さ n の論理型配列を用いれば良い．この配列を全て `.true.` に初期化し，素数でないと判定されたものは `.false.` を代入して消去する．

実行結果は例えば以下のようなものになる．

```
30                          # キーボード入力
                          2
                          3
                          5
                          7
                         11
                         13
                         17
                         19
                         23
                         29
```

5.10.7 課題 7

標準入力から 3 つの整数 `L`, `M`, `N` を読み込み，形状が `(L, M, N)` の整数型の 3 次元配列，および長さ `L*M*N` の整数型の 1 次元配列を作成せよ．その上で，

- 組み込み関数 `size`, `shape`, `lbound`, `ubound` の引数に上記の 2 つの配列をそれぞれ与えた結果を出力し，その動作を確認せよ．
- `reshape` を用いて 1 次元配列の中身を 3 次元配列にコピーできることを確認せよ．(ここで 1 次元配列に適当な値を代入してからコピーすることで `reshape` の動作を確認することもできる．)

```
                         (     )
2  5  7

      (                    )                      70
      (                       (       )            2         5       ␣
↪7
      (                       (   1  1 )           1         1       ␣
↪1
      (                       (       )            2         5       ␣
↪7

      (                    )                      70
      (                       (     )             70
      (                       (  )                 1
      (                       (     )             70
```

なお, L, M, N の値はそれぞれせいぜい 100 程度かそれ以下にしておいた方が良い.

第6章　書式指定・ファイル入出力・文字列処理

ここではこれまではオマジナイとして使ってきた `write(*,*)` や `read(*,*)` の意味を理解し，ファイル入出力や，出力時の書式指定の仕方，さらには文字列処理の方法について学ぼう．

参考:

- sample1.f90 : 書式指定
- sample2.f90 : ファイル入出力 (open と close)
- sample3.f90 : ファイル入出力 (アスキー形式)
- sample4.f90 : ファイル入出力 (バイナリ形式)
- sample5.f90 : 文字列処理
- sample6.f90 : ファイル各行の逐次処理

この章の内容

6.1 書式指定

実は `write(*,*)` の 2 つめの引数は出力される値の書式を表す **編集記述子** を指定するためのもので，`*` はデフォルトの書式を意味している (したがって一般にコンパイラによって出力の書式は異なる)．ここに決められた形式で書式を指定することで桁を揃えたりして，綺麗な出力を得ることが出来る．例えば

リスト 1: sample1.f90 抜粋

```
character(128) :: fmt
character(128) :: text
integer :: year = 2014

real(8) :: x, y, z

! 6 桁で出力
write(*, '(i6)') year
```

(次のページに続く)

(前のページからの続き)

```
! 編集記述子を文字型変数として渡す
fmt = '(i6)'
write(*, fmt) year
```

のように * の代わりの編集記述子 (ここでは `'(i6)'` や `fmt`) を指定することになる．11 行目のように直接指定してもよいし，14-15 行目のように文字型の変数を用いてもどちらでも構わない．

さて，問題は書式の指定の仕方 (上の例では `i6`) についてである．書式指定の詳細については適宜他の文献[1] を参照してもらうとして，ここでは以下の表に挙げる典型的な例を理解しておけば良い．

表 1: 編集記述子の使い方の例

データ型	記述子	例	例の説明
整数型	`I`	`i6`	6 桁で出力する．
		`i8.6`	8 桁で出力する．ただし 6 桁に満たない部分は 0 で埋められる．
実数	`F`	`f12.5`	12 桁で出力，うち 5 桁が小数点以下．
	`E`	`e20.7`	科学的表記[2] の 20 桁で出力，うち 7 桁が小数点以下．
文字型	`A`	`a`	与えられた文字型の字数に対応する桁数を確保して出力．
		`a30`	30 桁で右寄せの出力．
空白	`X`	`5X`	指定された数だけ空白を出力．この場合は 5 桁．
改行	`/`	`/`	改行される．

上の表は編集記述子の使い方の典型的な例を示している．基本的には **編集記述子** と **表示桁数** の組み合わせの形で表すことになっている．以下にいくつか注意点を挙げておこう:

- 数値データは指定された書式で出力が出来ない場合の出力は **全て "*" が出力されてしまう** ので気をつけよう．実数データの出力では符号，小数点，指数部の桁数を考慮しなくてはならないので，`E` 型記述子では 全桁数 ≥ 小数部の桁数 + 6 となっている必要がある．`F` 型編集記述子の場合はデータ (整数部の桁数) に依存するが少なくとも 全桁数 ≥ 小数部の桁数 + 2 が必要である．これは以下のような例によって確認することが出来る．

```
write(*,'(f5.3)') 3.1415      ! これは OK
write(*,'(f5.3)') 3.1415 * 10 ! これはダメ
write(*,'(e9.3)') 3.1415      ! これは OK
write(*,'(e5.3)') 3.1415      ! これはダメ
```

- 文字型データでは，桁数が渡された文字数よりも少ない場合は文字型の先頭から桁数分だけが出力される．
- 複数の編集記述子をカンマ , で区切って並べることが出来る．この際に，編集記述子ではない通常の文字列も同様にカンマで区切って並べることが出来る．ただし指定子の文字列が ' (シングルクオート) で区切られている場合には，その中の文字列は " (ダブルクオート) で囲まなければならない．使い方は以下の通りである．

[1] 高木 (2009，8 章) が詳しい．
[2] 科学的表記とは 3×10^{10} のような表記法のことである．Fortran の出力では `0.3e+11` のような形になる．

リスト 2: sample1.f90 抜粋

```
! 複数の記述子を並べる
write(*,'(e10.3, e10.3, e10.3)') x, y, z

! 文字列および改行を使う
write(*,'("x = ", e10.3, /, "y = ", e10.3, /, "z = ", e10.3)') x, y, z
```

- 同じ書式指定子を複数回繰り返す場合には書式指定子の前に繰り返す回数を指定する省略記法が使える．例えば `'(a10, 3i5)'` は `'(a10, i5, i5, i5)'` と等しい．

また書式指定は `write(*, fmt='(a)')` のように行うこともできる[3]．他には改行を抑制するために `advance='no'` のような指定も使うことがしばしばあるので覚えておいて損はない．Fortran では通常 `write` 文によって最後に自動的に改行が挿入されてしまうが，

リスト 3: sample1.f90 抜粋

```
! 出力後に改行しない
write(*, fmt='(a)', advance='no') 'Input some text : '
```

のようにすれば `'Input some text :  '` が表示された後に改行されない．

なお `read(*,*)` についても同様の指定が可能であるが，それほど必要な場面はないかもしれない．一つ考えられるのはスペースを含んだ文字列 (例えばファイルの 1 行分) を一つの文字型変数に読み込みたいときであろう．このときには

リスト 4: sample1.f90 抜粋

```
! 改行まで文字を読み込む
read(*, '(a)') text
```

のように `'(a)'` を指定しないとスペースまでの文字列が `text` に取り込まれてしまう．これは Fortran がデフォルトでスペースや改行などを変数の区切りとして読み込んでしまうからである．

6.2 ファイル入出力

6.2.1 open と close

これまではファイルから何らかのデータを読み込んだり，ファイルに出力する時にはシェルのリダイレクト機能を用いてきた．しかし，これだと例えば複数のファイルから別々に違うデータを読み込んだり，複数ファイルへの出力などの柔軟は処理はできない．このような場合にはソースコードの中で明示的に入出力に用いるファイルを指定し，そのファイルに対する入出力処理を行うように指定しなければならない．Fortran プログラムからファイルを開くには open 文を用いることになる．例えば以下の例は予め存在しないファイル `filename.dat` を新規作成して開く．(この例では，ファイルが既に存在している時にはエラーとなる．)

[3] このような引数の渡し方はこの後の open 文などでも出てくる．これは *optional* 属性とキーワード引数 † と呼ばれ，これによって関数やサブルーチンに順番を気にすること無く引数を渡すことが出来る．

リスト 5: sample2.f90 抜粋

```
integer :: ios

!
! ファイルを開く
!
! unit     : 装置番号 (ファイルを識別するための数値)
! iostat   : 正常時は 0
! file     : ファイル名
! action   : 操作
! form     : formatted(アスキー形式) or unformatted(バイナリ形式)
! status   : new, old, replace
! position : rewind, append
!

! * ファイルの新規作成 (既に存在する場合はエラー)
open(unit=10, iostat=ios, file='ascii.dat', action='write', &
     & form='formatted', status='new')
```

open には多くの引数を指定することになる．(あまりに多いので通常は *optional* 属性とキーワード引数 † で必要なものだけ渡す．) それぞれの意味は以下の様なものである．

unit

装置番号を指定する．装置番号とはファイルを特徴付ける一意な整数である．なお標準入力は 5，標準出力は 6，標準エラー出力は 0，と予め決められているのでこれら以外の値を指定すること．1 つのプログラム中で同時に複数のファイルを開く際には違う値を指定しなければならない．自分でプログラムを書く際には 10 以上の重複しない整数にしておくのが無難である．

iostat

ここに指定した整数型の変数に open 文の終了ステータスが格納される．ファイルが正常に開けた場合には 0，そうでない場合には 0 以外の値が返される．この変数をチェックすることでエラーチェックをするのが定石である．

file

ファイル名を指定する．

action

ファイルに対する操作を指定する．読み取り専用なら read，書き込み専用なら write，読み書きどちらもしたい場合には readwrite を指定する．デフォルトでは (可能ならば) readwrite となる．

form

アスキー形式 (書式付き) なら formatted，バイナリ形式 (書式なし) なら unformatted を指定する．デフォルトでは formatted となる．

status

ファイルの状態を指定する．new はファイルの新規作成を意味し，そのファイルが既に存在する時にはエラーが発生する．replace も新規作成であるが，そのファイルが既に存在する場合は中身を破棄して空のファイルとして開く．old は既に存在するファイルを開く．このときそのファイルが存在し

ていない場合はエラーが発生する．他にも処理系依存の unknown があり，これがデフォルトである．

ファイルが正常に開けたかどうかは以下のようにチェックすることが出来る．

リスト 6: sample2.f90 抜粋

```
! ファイルが正常に開けたかどうかをチェックする
if (ios /= 0) then
   write(*,*) 'Failed to open file for output'
   stop
end if
```

いちいちチェックするのは面倒なようだが，プログラミングにおいてこのようなエラーチェックは非常に大切なのでいつもチェックする癖をつけよう．プログラム作成の際にその半分がエラーチェックになるというのもよくある話である[4]．

以下にいくつかよく使う例を挙げておこう．(念の為に言っておくと open に渡す引数の詳細を覚えておく必要は無い．大まかな使い方さえ知っておけば，後は google で検索する方が早い．)

- ファイルを新規作成する．ただしファイルが既に存在する場合にはその中身を破棄する (上書き).

```
open(unit=10, iostat=ios, file='ascii.dat', action='write', &
     & form='formatted', status='replace')
```

- 既に存在するファイルを開き，位置をファイル終端に指定する (position='append'). 既存のファイルの終端にデータを追加する場合に用いる．

```
open(unit=10, iostat=ios, file='ascii.dat', action='write', &
     & form='formatted', status='old', position='append')
```

- 既に存在するファイルを読み込み専用で開き，位置をファイル先頭に指定する (position='rewind'). 既存のファイルの先頭からデータを読み込む場合に用いる．ただし position=rewind は必ずしも指定しなくても良い．

```
open(unit=10, iostat=ios, file='ascii.dat', action='read', &
     & form='formatted', status='old', position='rewind')
```

なお，open で開いたファイルは close で閉じるのが作法である．ファイルを閉じないままプログラムが異常終了してしまうと，せっかくの結果が正しく出力されないこともあり得るので注意して欲しい．close には open で開いた時に用いた装置番号を指定する．例えば装置番号が 10 であれば

```
close(10)
```

とすれば良い．

[4] 書き殴りでその後 2 度と使わないようなコードの場合はそれほど気にする必要はないかもしれないが．

6.2.2 アスキー (書式付き) 入出力

アスキー形式とかテキスト形式と呼ばれるファイルはテキストエディタや cat などのコマンドで人間が理解できる形で表示できるファイルである．アスキー形式でファイルを開くには open で form='formatted' と指定するのは既に説明した通りである．

開いたファイルへの入出力をするには，write や read に装置番号を指定しなければならない．実はこれまで使ってきた write(*,*) や read(*,*) の 1 番目の引数は装置番号を意味するものである．ここでも"* "はデフォルトの装置番号を意味し，通常は write であれば標準出力の 6，read であれば標準入力の 5 を指定したことと同じ意味となる．ここに open 文で指定した装置番号を代わりに指定することで，write の出力先，read の入力先ファイルが指定出来る．

なお装置番号が何であっても 2 番目の引数には同じように編集記述子を指定すれば良い．例えばファイル (装置番号=10) にデータを出力するには

リスト 7: sample3.f90 抜粋

```
! ファイル (装置番号=10) にデータを書き出し
do i = 1, 64
   x = real(i,8)/real(n-1,8)
   y = cos(2*3.1415_8 * x)
   write(10, '(e20.8, e20.8)') x, y
end do
```

のようにすればよい．もちろん編集記述子をデフォルトの * にすることも可能である．

6.2.3 バイナリ (書式なし) 入出力

バイナリ形式とは一般にテキスト形式以外の，人間がそのままでは解釈できない形式のファイルの総称である．アスキー形式での入出力では計算機の内部表現 (メモリ上のビット列) を人間に解釈できる形式に逐一変換して入出力を行っている．Fortran では open で form='unformatted' を指定して開いたファイルに対してはこのような変換が行われず，メモリ上のビット列が (ほぼ) そのまま出力されることになる．アスキー形式への変換が行われないため，編集記述子は指定することは出来ず，write(10)，read(20) のような形で装置番号のみを指定して入出力を行う．

バイナリ入出力を行うメリットとして，まず高速であることが挙げられる．これはテキスト形式への変換を行わないことに加えて，テキスト形式での入出力に比べてデータ量を少なく抑えることが出来るためである．また内部的に 2 進数で表されている実数は 10 進数では正確に表現出来ないことから，実数型のデータはアスキー形式への変換に伴い情報が失われてしまうが，バイナリ形式での入出力ではこの問題がない．

デメリットは人間の目ではデータの中身が判別出来ないことである．従って，バイナリで出力されたデータを読み込むにはどのような形式で出力されたのかを予め正確に知っていなければならない．例えば，

```
write(10) x
write(10) y
```

と出力されたデータを読み込むには

```
read(10) x
read(10) y
```

としなければならないし，出力が

```
write(10) x, y
```

であれば入力は

```
read(10) x, y
```

とする必要がある．すなわち write で指定した変数並びと全く同じ変数並びで read しなければ正しく読み込みが出来ない．(大変困ったものであるが，ストリーム入出力 † を用いれば少なくともこの問題は生じない．) また，当然であるが，x や y は 配列の入出力 の時と同様に配列でも良い．

以下では，unformatted 形式でファイルにデータを出力しそれを読み込む例 sample4.f90 を見ながら具体的に考えてみよう．

リスト 8: sample4.f90 抜粋

```
! 配列サイズの出力
write(10) n

! 配列の出力
write(10) x, y
```

では配列 x および y の長さである整数 n と，配列そのものをファイルに出力している．ここで作成したファイルを一度閉じた上で，

リスト 9: sample4.f90 抜粋

```
! 配列サイズを読み込む
read(20) n

allocate(xx(n))
allocate(yy(n))

! 配列を読み込む
read(20) xx, yy
```

によってデータを読み込む．ここで， `write` の形式と `read` の形式が全く同じになっていることが分かるだろう．

なお，同じ形式で入出力をしたとしても一般には異なる環境で作成したバイナリファイルには互換性が無い．ただし，この問題については多くの場合に対応が可能である (次節参照)．

6.2.4 バイトオーダー †

マルチバイトのデータを計算機のメモリ上に配置する方法のことをバイトオーダーとかエンディアンなどと呼ぶ．分かりやすい (？) 例として IP アドレスを考えよう．IP アドレスは (IPv4 では)4 バイトで表され，192.168.1.0 のように 1 バイトごとに"."で区切って記述するのが一般的である．これをメモリに格納する際に 192，168，1，0 の順に格納する方法をビッグエンディアン，逆に 0，1，168，192 の順に格納する方法をリトルエンディアンと呼んでいる．このバイトオーダーは CPU 依存である．Intel 系の CPU の場合はリトルエンディアンが採用されているので，普通の PC を扱っている限りはバイトオーダーを気にする必要は無い．一方でスーパーコンピューターなどではビッグエンディアンが採用されている CPU も比較的多く，そのような計算機で出力したデータを手元の PC で読み込む際にはバイトオーダーの変換が必要になってくる．

この変換は手動で行うことも出来るが，コンパイラのオプションで自動的に変換を行うように指定することも可能である．例えば gfortran の場合はコンパイルオプションに `-fconvert=big-endian` を用いると，入出力がビッグエンディアンのバイトオーダーで行われる．他にも環境変数 `F_UFMTENDIAN` を指定することでもバイトオーダーを指定することが出来るようである．

6.2.5 ストリーム入出力 †

Fortran の `unformatted` のバイナリデータは一般には C 言語の `fread`，`fwrite` による入出力と互換性が無い．これは Fortran は各 `write` 文で出力されるデータの前後に余計なデータ (ヘッダーおよびフッター) を付加するためである[5]．新しい Fortran 2003 規格ではストリーム入出力という C 言語の `fread`，`fwrite` と同じ (余計なデータを付加しない) 読み書きが出来るようになっている．これには以下のように `access='stream'` を指定してファイルを開く．

[5] ただしこれは慣習的にそうなっているだけで，標準化されているわけではなく，完全に処理系依存である．ヘッダーやフッターは 4 バイトのこともあるし 8 バイトのこともある．同じコンパイラであってもオプションで変更することも出来る．ここで扱うストリーム入出力であればバイトオーダーさえ把握していればヘッダーやフッターのことを気にする必要が無い．

```
open(unit=10, file='binary.dat', access='stream', form='unformatted')
```

このように開かれたファイルに対しては

```
write(10) x, y, z
```

と

```
write(10) x
write(10) y
write(10) z
```

では全く同じデータが出力される．このように生成されたデータファイルは C 言語の `fread` を用いて正常に読み込みが出来るし，また `fwrite` で出力したデータも Fortran の `read` 文で読み込むことが可能である．

6.3 文字列処理

6.3.1 文字型変数の宣言

文字型変数の宣言は

```

```

のように行えば良いことは既に学んだ．ここで指定した文字数に足りない部分はスペースで埋められる．従って，例えば長さ 10 の文字型変数に `'ABCDE'` を代入すると，その後ろに空白文字が 5 文字代入される．定数の場合は文字型の初期化時には長さを陽に指定しなくても

リスト 10: sample5.f90 抜粋

```
  character(len=*), parameter :: text = 'initialization by this string'
```

のように `len=*` を指定して初期化すると，自動的に初期化文字列の長さを持った文字型変数となる[6]．

6.3.2 結合と代入

文字型変数は `//` 演算子を用いて結合することや，結合した文字列を他の文字型変数に代入することが出来る．ただしこの際には空白の存在に注意しなければならない．例えば

リスト 11: sample5.f90 抜粋

```
  character(len=16) :: a, b, c, d
```

のように宣言して

[6] 後述の関数やサブルーチンの仮引数でもこの形式が使える．

リスト 12: sample5.f90 抜粋

```
a = 'This'
b = 'is'
c = 'a'
d = 'pen'

! 何も考えずに出力
write(*,*) a, b, c, d

! 文字列を結合して出力
write(*,*) a // b // c // d
```

のようにした場合には，15 行目と 18 行目の出力はいずれも

```

```

のようになるであろう．これは a， b， c， d のどれも 16 文字の文字列であるので，16 文字に満たない部分は全て空白で埋められているためである．したがって，そのまま出力しても (15 行目)， // で結合しても (18 行目)，空白は取り除かれずに残ることになってしまう．

文字列の前後の空白文字を除去するには trim という組み込み関数を用いれば良い．空白を除去して各文字列間に 1 文字分ずつ空白を入れて出力するには

```
write(*, '(a,x,a,x,a,x,a)') trim(a), trim(b), trim(c), trim(d)
```

などとすれば良い．ここで書式記述子の x が空白を表す．これによって以下の様な出力が得られる．

```

```

6.3.3 部分文字列

配列の場合には a(1:10) のように部分配列を用いることが出来た．これと同じことが文字型変数に対しても出来る．例えば

リスト 13: sample5.f90 抜粋

```
character(len=64) :: str = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

のように宣言された str に対して，たとえば str(1:1) であれば A，str(6:10) であれば FGHIJ のように文字列の一部 (部分文字列) を取り出すことができる．以下は特定の文字列の検索を行い，該当部分だけを取り出すような処理の例である．

リスト 14: sample5.f90 抜粋

```
! XYZ と一致する部分文字列を検索し，開始位置のインデックスを返す
i1 = index(str, 'XYZ')
i2 = i1 + len('XYZ')
write(*,*) str(i1:i2)  ! XYZ を出力
```

ここで組込み関数 index(s1, s2) は文字列 s1 の中から s2 に一致する部分文字列を検索し，開始位置のインデックスを返す関数である．文字列操作に関するその他の組み込み関数については富田・齋藤 (2011, 7 章) などを参照のこと．

6.3.4 内部ファイル

プログラム中で必要な文字列のフォーマットを揃えるのには **内部ファイル** と呼ばれる機能を知っていると何かと便利である．これまでに write の一番目の引数には装置番号を指定することを学んだが，以下の例では装置番号の代わりに文字型変数 str が指定されている．

リスト 15: sample5.f90 抜粋

```
! 内部ファイルを用いて連番のファイル名を作成
do n = 1, 8
   write(str, '("data",i3.3,".dat")') n
   write(*, '(a)') str
end do
```

これによりフォーマットされた文字列が str に代入される．この例では data001.dat , data002.dat , ... のように連番のファイル名を作成して出力する．

6.4 ファイル終端までの逐次処理

文字列処理ではファイルの中身を一行ずつ読み込んで何らかの処理を行うことが多い．以下のサンプルコードは標準入力から 1 行ずつ順に読み込み処理をする例である．ファイルが終端に達したかどうかを調べるには read の終了ステータス iostat を取得すれば良い．正常に読み込みが実行された時には iostat が 0 となり，これが負の場合にはファイルの終端，正の場合には何らかのエラーが発生したことを意味する．なお，実際にはこのような処理はシェルスクリプトや Perl，Python，Ruby などを使って実装する方が圧倒的に簡単である．(何でもかんでも Fortran でやろうとするのは無駄が多い．)

リスト 16: sample6.f90 抜粋

```
integer, parameter :: max_line = 256
integer :: ios, nline
character(max_line) :: line

nline = 0
```

(次のページに続く)

(前のページからの続き)

```
!
! fmt='(a)'  : 行末までを読み込むのに必要 (途中のスペースで途切れないように)
! iostat=ios : ステータスが ios に代入される
!
read(*, fmt='(a)', iostat=ios) line

!
! ios >  0 : 何らかのエラー
! ios == 0 : 正常に読み込まれた
! ios <  0 : ファイルの終端に達した
!
do while(ios == 0)
   ! 空白行以外をカウント (line == '' なら空白行である)
   if (line /= '') then
      nline = nline + 1
   end if
   ! 次の行を読み込む
   read(*, fmt='(a)', iostat=ios) line
end do
```

ここでは標準入力からリダイレクトによって

```
```

のように与えられたファイルのうち空白行以外の行数を数える処理となっている.

6.5 第 6 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例
- 課題 6 解答例
- 課題 7 解答例
- 課題 8 解答例

6.5.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

6.5.2 課題 2

以下のように掛け算九九の表を標準出力にキレイに表示するプログラムを作成せよ．

```
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81
```

6.5.3 課題 3

helix1.dat から実数データ $x_i, y_i, z_i (i = 1, \ldots, 32)$ を読み込み，全く同じフォーマットで標準出力に表示するプログラムを作成せよ．以下のようにリダイレクトでファイルに出力し，`diff` コマンドによってフォーマットが同じかどうかを確かめよ．

```

```

上記の `diff` コマンドを実行して，出力が何も無ければファイルが同一であることを意味する．

データファイルには i 行目に x_i, y_i, z_i の 3 つの実数データが記述されている． `open` を用いて読み込むこと．

6.5.4 課題 4

課題 3 と同様に読み込んだデータ x_i, y_i, z_i をそれぞれ別のファイル (例えば `x.dat`, `y.dat`, `z.dat`) にアスキー形式で出力せよ．各データの書式は helix1.dat のものと同一とする．

このとき以下のコマンドによって結果を確認できる．

```
          " "
```

`helix1.dat` と `test.dat` が同一ファイルになっていれば (`diff` コマンドが何も出力しなければ) 良い．

6.5.5 課題 5

バイナリファイル helix2.dat を open を用いて開き，実数データ $x_i, y_i, z_i (i = 1, \ldots, 32)$ を読み込み，先ほどの helix1.dat と全く同じフォーマットで標準出力に表示するプログラムを作成せよ．

作成したプログラムを

```

```

のように実行し，結果が helix1.dat と同じになることを diff コマンドによって確かめよ．

なお，このバイナリファイルは form='unformatted' で open したファイルに (その装置番号を 10 として)

```
write(10) x
write(10) y
write(10) z
```

のように出力したものである． ただしここで， x, y, z はそれぞれ長さ 32 の倍精度の実数配列である．すなわち

```
real(8) :: x(32), y(32), z(32)
```

のように宣言されたものであると考えれば良い．

6.5.6 課題 6

Fortran のソースコードから，何らかの Fortran の命令文を含む行数 (コメントのみの行および空白行を除いた行数) を数えるプログラムを作成せよ．

入力はリダイレクトによって

```
                                                    24
```

のようにすれば良い． (チェックが出来ればファイル名は何でもよい．)

なお，コメントのみの行は最初の空白以外の文字が "！"である行，空白行は空白のみで表される行であるとして判定すれば良い．組込み関数 adjustl を用いると良い．

6.5.7 課題 7 †

Fortran の通常の unformatted バイナリファイルは一般には他の言語と互換性が無いが，ストリーム入出力 を使うことで他の言語と同様にバイナリファイルを扱うことが出来る．ここでは C 言語で

```
// 配列サイズ
const int N = 10;

// 倍精度実数の配列
double x[N];

// x には 1.0 から 5.5 まで 0.5 刻みでデータを格納

// x に格納された倍精度実数を N 個分ファイルにバイナリで出力
fwrite(x, sizeof(double), N, fp);
```

のように生成した cbinary.dat を Fortran から読み込むプログラムを作成せよ．実行結果は例えば以下のようになる．

```
     read                    in
1
1
2
2
3
3
4
4
5
5
```

なおこのデータを作るのに用いた C 言語のコードは mkbin.c である．

6.5.8 課題 8 †

Fortran の unformatted バイナリファイル helix2.dat をストリーム入出力を用いて読み込み，課題 5 と同様に出力するプログラムを作成せよ．ここで多くのコンパイラが unformatted の場合には実際のデータの前後に 4 バイトずつヘッダーとフッター（データのバイト数を表す整数）を付与するので，これらを読み飛ばす必要があることに注意せよ．

これを理解しておけば多言語からもデータの読み書きが可能である．例えば，C 言語では helix.c ， Python では helix.py が同じ動作をするプログラムになっている．(Python の場合は scipy がインストールされていれば scipy.io.FortranFile を使って簡単に読み込むことが出来る．)

第7章　関数とサブルーチン

これまでのプログラムは全ての処理が `program` から `end program` で囲まれた部分に記述されていたことと思う. これをメインプログラムと呼ぶ．これに対して，メインプログラム以外にもまとまった処理を 1 つのプログラム単位として記述しておくことが出来る．これをサブプログラムと呼んでいる．サブプログラムを用いることで同じ処理を何度も書かずに済むようになるため，プログラムを簡潔に記述することが出来る (従って無用な間違いも減らすことが出来る).

なお Fortran でのサブプログラムには関数 (`function`) とサブルーチン (`subroutine`) の 2 種類が有る[1]. 関数は値を返すのに対してサブルーチンは値を返さないという違いが有るが，どちらも同じようなものである．(実際にほとんどの言語でサブルーチンと関数の区別は存在しない．Fortran で言うところサブルーチンは C/C++では単に返値が `void` 型の関数でしかない.) 最初は少し取っ付きづらいかもしれないが，関数やサブルーチンを使いこなせるようになると格段にプログラムの開発が楽になるので積極的に利用しよう.

参考:

- sample1.f90 : 関数
- sample2.f90 : サブルーチン
- sample3.f90 : 変数のスコープ
- sample4.f90 : intent, 配列渡し，save
- sample5.f90 : optional とキーワード引数
- sample5.f90 : 再帰呼び出し
- sample7.f90 : 内部手続き
- sample8.f90 : 外部手続き

この章の内容

[1] 後で学ぶモジュールもサブプログラムになるのだが，ここではこの 2 つのみを考えれば良い.

7.1 概要

関数やサブルーチンなどのサブプログラムはひとまとまりの処理を実行する独立なプログラムと考えることが出来る．例えば，3 次方程式 $ax^3 + bx^2 + cx + d = 0$ の数値解を与えられた初期値からなんらかの反復法を用いて求める処理を考えよう[2]．

ここで係数 a, b, c, d の値が何であっても行う処理は基本的に同じであろう．では 2 つの異なる係数の組に対して数値解を知りたい場合はどうすれば良いだろうか? 同じ処理なのだからコピー＆ペーストして係数の値だけ書き換えるというのも一つの案である．しかし，100 個の異なる係数の組に対する解を知りたい場合にはどうすれば良いだろうか? 100 回コピー＆ペーストするというのは得策とは言えない．このような場合には与えられた係数の組に対して処理を実行し，結果を返す **関数** が定義されていれば，それを 100 回呼び出すだけである．($\cos x$ のような数学関数を思い浮かべてもらえば良い.)

このように，何らかのまとまった処理を 1 つの独立したプログラム単位として定義することが出来れば圧倒的にプログラム作成が簡単になる．プログラム作成時に大事な事は人間は間違える動物であるという前提に立つことである．間違いを少なくするには問題を簡単にするしか方法は無い．サブプログラムはそれ単体で独立したプログラムであるから，仕様さえ確定してしまえばプログラマはそのサブプログラムの実装に集中すれば良く，それがどのように使われるかに気を使う必要は無いのである．また，サブプログラムが機能的に小さければ小さいほど (問題設定が単純になるので) 実装が簡単で間違いが少ない[3]．従って，規模の大きなプログラムを開発する際には，機能を出来る限り分割した小さな関数やサブルーチン群を実装し，それらを利用して最終的に所望の機能を実装する戦略の方が圧倒的にバグが入りにくいし，またあったとしても修正が容易になる．プログラムを作成する際には関数やサブルーチンにまとめられるような処理が無いか常に注意しておくべきである．

次図は関数とサブルーチンの概念図である．いずれもひとまとまりの処理を実行するものであり，何らかの入力を受け取り，出力を返すことが出来る．(もちろん入力や出力は無くても良い.) ひとたび正しく実装してしまえば，利用者 (呼び出し側) はサブプログラムの内部の実装を気にする必要は無く，ブラックボックスとして用いることが出来る．($\cos x$ の値を内部でどのように計算するのか意識しながら用いることは無いのと同じである.) このため，利用者はその他の処理の実装に集中することが可能になるのである．

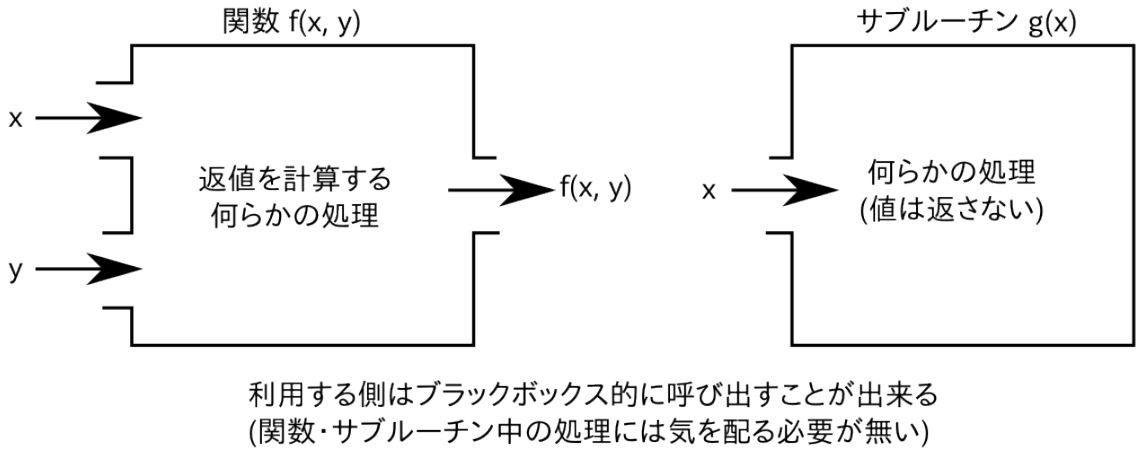

図 1: 関数とサブルーチンの概念図.

[2] 3 次方程式には解の公式が存在するが，実用的には反復法の方が精度や速度の面で有利なことが多い.

[3] 全体像を完全に把握出来るプログラムの規模は常人にはせいぜい 1000 行程度が限界であろう．実際には 100 行でも怪しいものである.

サブプログラムはメインプログラムとは独立したプログラムなので，メインプログラムとは他の場所に定義して呼び出すことになる．以下の図はメインプログラムで全ての処理を実行する場合と，サブプログラムを利用する場合の処理の流れを概念的に示している．サブプログラムを呼び出すと，その定義に記述されている処理を実行し，終了すると呼び出し元に処理が返る仕組みになっている．もちろんサブプログラムが他のサブプログラムを呼び出すことも可能である．

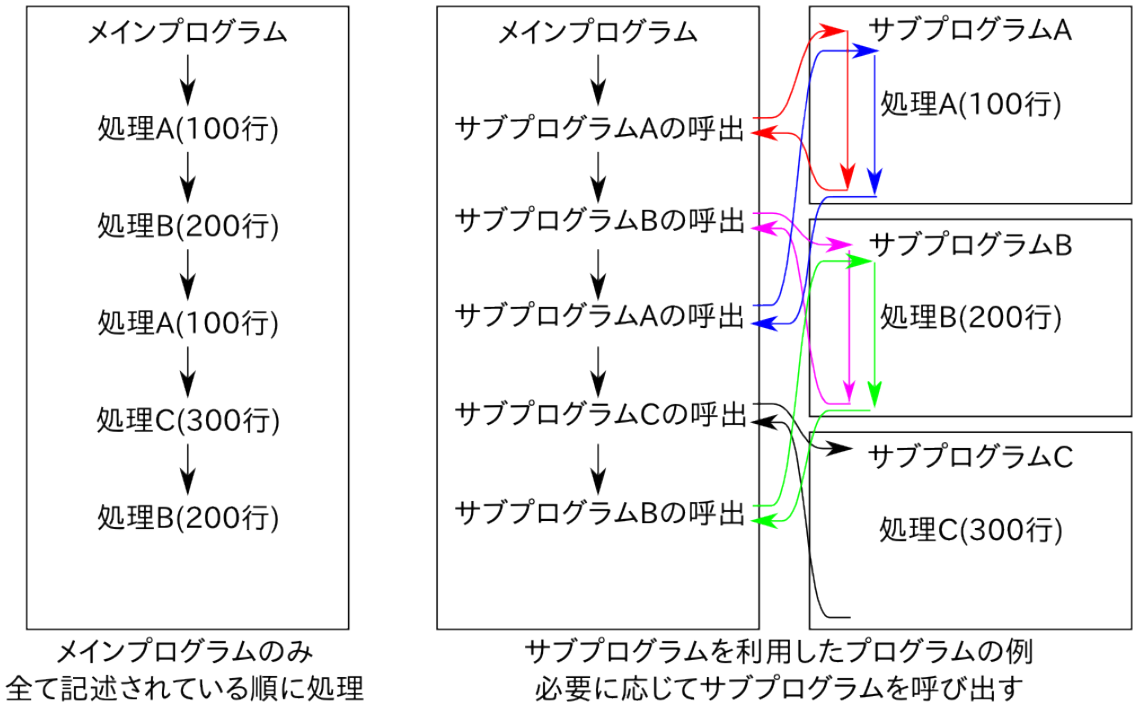

図 2: サブプログラムを利用したプログラムの概念図.

7.2 定義と呼び出し

まずはサブプログラムの定義の仕方と，その呼び出し方を学ぼう．

7.2.1 定義場所

詳細は 内部手続きと外部手続き † に譲るが，基本的には関数やサブルーチンはメインプログラムの `stop` の後に `contains` を挿入し，そこから `end program` の間に定義すれば良い．

```
program sample
  implicit none

  ! メインプログラムの処理

  stop
contains
```

(次のページに続く)

(前のページからの続き)

```
  ! ここに関数やサブルーチンを定義する

end program sample
```

この方法で定義した関数やサブルーチンは特に準備せずにメインプログラムから呼出しが出来る．私見では，後に学ぶモジュールを使う場合 (ソースファイルを複数に分割する場合) を除けば，このやり方が最も間違いを減らすことが出来る方法である．従って特に理由がない限りはこの方法で関数・サブルーチンを定義することを推奨する．

7.2.2 関数

関数とは何らかの値を返すサブプログラムであり, 以下の様な形式で定義される．

```

```

例えば以下は倍精度実数の 2 乗を返す関数の定義である．

リスト 1: sample1.f90 抜粋

```
!
! 関数の宣言 (1) : 関数名と同じ名前の変数に返値を代入する形式
!
real(8) function square1(x)      ! square1 という名前で関数を宣言
  implicit none                  ! 暗黙の型宣言の禁止
  real(8) :: x                   ! 引数を宣言

  square1 = x**2                 ! 返値は関数名と同じ名前の変数に代入

  return                         ! 呼び出し元に制御を戻す
end function square1
```

メインプログラムが program で始まり end program で終わるのと同様に，関数定義も function で始まり end function で終わる一つの独立したプログラム単位である．従って implicit none によって暗黙の型宣言を禁止し，使用する変数は明示的に宣言をする必要がある．メインプログラムとの大きな違いは，サブプログラムには引数 (この場合は x) が入力として与えられることであり，引数についてもデータ型を宣言しなければならない．引数はカンマで区切って複数与えても良いし，それぞれが別のデータ型であっても構わない．なおサブプログラムの定義時の引数のことを仮引数と呼び，呼び出すときに実際に与えられる引数のことを実引数と呼ぶ．一方，返値は **関数名と同じ名前の変数に値を代入する** ことでそれが返値

となる．返値のデータ型は 1 行目先頭の real(8) で指定される．return 文はサブプログラムからその呼び出し元へ制御を戻すことを意味しており，上のように他の処理が全て終わった後であれば省略しても構わない．処理の途中で return することも可能で，その場合にはそれ以降の処理は行われずに呼び出し元へと制御が戻される. 例えば if 文で何らかの条件判定を行い，それ以降の処理を行う必要がないと判断した場合にはその時点で return によって関数から抜けることが出来る．

なお関数宣言時に function の前に返値のデータ型を指定するのでは無く，以下の様に result(変数名) のような形で指定することも出来る．

```
```

この場合は result で指定された変数に値を代入することで，それが関数の返値となる．従って，先ほどの square の定義は以下のように行うことも出来る．

リスト 2: sample1.f90 抜粋

```
!
! 関数の宣言 (2) : result を使った形式
!
function square2(x) result(y)    ! square2 という名前で関数を宣言
  implicit none                  ! 暗黙の型宣言の禁止
  real(8) :: x                   ! 引数を宣言
  real(8) :: y                   ! 返値となる変数 (result) の宣言

  y = x**2                       ! 返値を代入

  return                         ! 呼び出し元に制御を戻す
end function square2
```

関数の呼出は組込み関数と全く同じで，以下のように適宜引数を与えて呼び出せば良い．

```
write(*,*) 'square1 => ', square1(2.0_8)
write(*,*) 'square2 => ', square2(2.0_8)
```

ただし，引数の型は宣言したものと同一でなければならない．この場合は引数は real(8) で宣言されているので，

```
write(*,*) square1(2.0), square2(2.0)
```

のような呼出しはコンパイルエラーとなることに注意して欲しい．

7.2.3 サブルーチン

サブルーチンは関数と良く似ているが，値を返さないという違いがある．定義は以下の様な形式となる．

```
```

例えば

リスト 3: sample2.f90 抜粋

```
!
! サブルーチンの宣言
!
subroutine hello(name)              ! hello という名前でサブルーチンを宣言
  implicit none                     ! 暗黙の型宣言の禁止
  character(len=*) :: name          ! 引数を宣言（任意長の文字列）

  write(*,*) 'Hello ', name         ! 内部の処理

  return
end subroutine hello
```

は引数で渡された文字列 name を Hello に続けて標準出力に表示するだけのサブルーチンである．関数と非常によく似た構造になっていることが分かるだろう．実際に，関数で必要だった返値の型指定が無いことを除くとほとんど同じである．

サブルーチンの呼び出しは関数と異なり call を用いて以下のような形でなければならない．

```
call hello('Michel')
```

(逆に関数呼び出しに call を用いることはできない.)

7.3 変数のスコープ

注意しなければならないのは，サブプログラムは 1 つの独立したプログラム単位であるので，その中で宣言する **変数は外部の変数とは完全に独立** であるという点である．例えばメインプログラムで宣言されている x という変数とサブプログラム中で宣言されている x という変数は完全に別のものとして扱われる．また当然サブプログラム中で使用している変数を外部から使用することは出来ない．例外 (内部手続き を参照) はあるものの，基本的にはサブプログラムとメインプログラム及び他のサブプログラムは全く独立なものとして考えて良い[4]．**サブプログラムと外部の情報のやり取りは基本的には引数と返値を通じて行う** ものと理解して欲しい．

[4] 内部手続き以外の例外としては Fortran 77 の common を用いる場合が挙げられるが，とりあえずはこのように理解して欲しい．

以下に挙げる例ではサブルーチン内部で変数を宣言し使用している．

リスト 4: sample3.f90 抜粋

```
! sub1
subroutine sub1(exponent)
  implicit none
  real(8) :: exponent

  !
  ! 基本的に内部でのみ使う変数はここで宣言して使う
  ! 特にループ変数 (以下の例では i) は必須
  ! これらの変数は外部からは見えないので，メインプログラムや他のサブプログラム
  ! で同名の変数が宣言されていてもそれらとは完全に独立であることに注意
  !
  ! (引数に値を代入して返す場合，メインプログラムの変数を参照する場合について
  ! はレジュメの intent 属性や内部手続きの節を参照のこと)
  !
  integer, parameter :: n = 10
  integer :: i
  real(8) :: x(n), sum

  ! 代入
  do i = 1, n
     x(i) = i ** exponent
  end do

  ! 和を計算
  sum = 0.0_8
  do i = 1, n
     sum = sum + x(i)
  end do

  write(*,*) 'sum of array = ', sum

end subroutine sub1
```

この変数は全てこのサブルーチン内部でのみ参照されるものであり，外部からはこれらの変数は参照出来ない．従って，別のサブルーチンが

リスト 5: sample3.f90 抜粋

```
! sub2
subroutine sub2()
  implicit none

  ! このように定義されていても問題無い (sub1 の n とは独立な変数である)
```

(次のページに続く)

(前のページからの続き)

```
    integer, parameter :: n = 100

    write(*,*) 'n = ', n

  end subroutine sub2
```

のように定義されていても， sub1 中の n と sub2 中の n は全く別の変数である．

7.4 引数の詳細

7.4.1 intent 属性

関数は返値として値を返すことが出来るが，返値はあくまでも一つだけである．実用的には複数の値を結果として返して欲しい場合も多いが，そのような場合には引数に結果の値を代入して返すことが出来る[5]．このことからすぐ分かるように関数とサブルーチンには本質的な違いは無い．サブルーチンを使っても返したい値を引数に代入して返せば良いからだ．どちらを使うかは好みの問題であろう．(Fortran しか使わない人はあまり関数を使いたがらない傾向があるように思える．一方で C 言語では関数の返値はエラーチェックに使うことが多いので，C 言語から入った人は関数を好むかもしれない.)

さて，実際には引数で与えた変数の値を勝手に変更して欲しく無い場合もあるだろう．そのため，以下のようにサブプログラムの定義時に引数の入出力特性を指定することが出来る．

- `intent(in)`
  入力用の変数に指定する．値は内部で参照されるのみで変更はされない．(変更しようとするとコンパイルエラーとなる．)
- `intent(out)`
  出力用の変数に指定する．サブプログラム中で値が代入されることを意味する．
- `intent(inout)`
  入出力のどちらにも用いる変数に指定する．何も指定しない場合のデフォルト．

例えば以下のように引数に属性を指定することによって意図せず第 1 引数 a や第 2 引数 b の値が変更されてしまうバグを防ぐことが出来る．

リスト 6: sample4.f90 抜粋

```
!
! <<< intent 属性 >>>
!
! * intent(in)    => 入力用変数 (値の変更不可)
! * intent(out)   => 出力用変数
! * intent(inout) => 入出力
!
! 以下は
!
```

(次のページに続く)

[5] 他にも 構造型 を用いるという方法も無いこともない．

(前のページからの続き)

```
! c = a + b
!
! のような処理を行うことを意図している．ユーザーはこの場合に a や b が変更されると
! は予想しないであろう．誤ってサブルーチン内で a や b の値を変更するのを防ぐために
! intent(in) を指定する．
!
subroutine add(a, b, c)
  implicit none
  real(8), intent(in)  :: a, b        ! 入力用変数 (変更不可)
  real(8), intent(out) :: c           ! 出力用変数

  ! 以下はコンパイルエラー
  !a = 1.0_8

  ! 出力用の変数に値を代入
  c = a + b

end subroutine add
```

C 言語の経験者は C 言語の関数の引数が値渡しなのに対して Fortran の関数やサブルーチンでは参照渡しであることに注意して欲しい．C 言語では明示的にポインタを (または C++での参照を) 渡さない限り呼び出し元の値が変更されることは無いが，Fortran ではサブプログラム中で引数の値を変更すると呼び出し元の値まで変更されてしまうのである．

7.4.2 配列渡し

配列も同様に関数やサブルーチンに引数として渡すことが可能である．以下の例の `average1` では任意のサイズの配列を渡すことが出来る．(ただし次元は予め指定しておく必要がある.) 配列のサイズや形状が必要であれば `size` や `shape` などの組込み関数を使って求めることが出来る．一方で `average2` では配列サイズを引数として明示的に渡している．配列の添字範囲を指定するなどの特別な事情が無い限りは `average1` のような書き方の方がシンプルで良い．

リスト 7: sample4.f90 抜粋

```
!
! <<< 形状引継ぎ配列の使い方 >>>
!
! 引数の配列のサイズは自動的に呼出し時に与えた配列のサイズになる
! サイズが必要な場合は組み込み関数 size を用いて取得可能
!
function average1(x) result(ave)
  implicit none
  real(8), intent(in) :: x(:)         ! サイズは自動的に決まる
  real(8) :: ave
```

(次のページに続く)

(前のページからの続き)

```
    ave = sum(x) / size(x)

  end function average1

  !
  ! <<< 配列サイズの引数渡し >>>
  !
  ! 配列のサイズを引数として明示的に受け取る
  !
  function average2(n, x) result(ave)
    implicit none
    integer, intent(in) :: n            ! サイズを引数として受け取る
    real(8), intent(in) :: x(n)         ! サイズは引数として渡された整数
    real(8) :: ave

    ave = sum(x) / size(x)

  end function average2
```

7.4.3 save 属性

関数やサブルーチン内で save 属性付きで宣言された変数は前回の呼び出し時の値を記憶しておくことが出来る (C 言語の static 変数と同等である)．従って，例えば自分が呼び出された回数を保持することなどもできる．

save 属性付きの変数はプログラムの開始時に一度だけ宣言文で代入された値に初期化される．例えば以下のサブルーチン `fibonacci` ではプログラムの開始時に `n = 1`，`f0 = 0`，`f1 = 0` と値が初期化されるが，呼び出しごとに値が変更され，プログラムが終了するまでその値を内部に保持し続ける．

リスト 8: sample4.f90 抜粋

```
  !
  ! <<< save 属性 >>>
  !
  ! save 属性付きの変数はプログラム実行中はその値を保持するので，複数回呼び出され
  ! た場合には前回の呼出し時の値を記憶したままとなる
  !
  subroutine fibonacci()
    implicit none
    ! 以下の 3 つが save 属性付き（初回の呼出し時の値は宣言文で与える）
    integer, save :: n  = 1
    integer, save :: f0 = 0
    integer, save :: f1 = 0
```

(次のページに続く)

(前のページからの続き)

```
    integer :: f2

    if (n == 1) then
       write(*,*) 'Fibonacci number [', 0, '] = ', f0
       f2 = 1
    else
       f2 = f0 + f1
    end if

    write(*,*) 'Fibonacci number [', n, '] = ', f2

    ! 次回呼び出し用（これらの値を記憶し続ける）
    n  = n + 1
    f0 = f1
    f1 = f2

  end subroutine fibonacci
```

なお Fortran では変数宣言時に同時に初期化を行うと，それを自動的に save 属性付きと扱うようである．(個人的にはこれは大変紛らわしい仕様だと思うのだが・・・)

すなわち

```
integer :: n = 1
```

と宣言された変数には自動的に save 属性が付加されるため n = 1 に初期化されるのは一度だけである．一方で，

```
integer :: n
n = 1
```

では毎回 n = 1 に初期化される．混乱を防ぐために save 属性付きとしたい変数は明示的に save を指定し，それ以外の変数は宣言時の初期化は避けたほうが無難である．

7.4.4 optional 属性とキーワード引数 †

引数の型宣言において optional 属性を指定した引数は，呼出し時に省略することが出来る． optional 属性付きの引数は，その引数が与えられたかどうかを検査する present という組込み関数と共に用いる．すなわち present(引数) は引数が与えられていれば真，そうでない場合には偽を返すので，if による条件分岐と組み合わせて用いれば良い．以下の例では引数 unit が与えられた場合にはその装置番号へ，与えられていない場合は標準出力へと出力を行う．

リスト 9: sample5.f90 抜粋

```
!
! <<< optional 属性 >>>
!
! optional 属性付きの引数は呼出し時に与えなくても良い．与えられなかった場合のデ
! フォルトの振る舞いはユーザーの責任で実装しなければならない．
! 以下では出力先を引数で与える装置番号にするか，デフォルトの標準出力にするかを
! 選択することが出来る．
!
subroutine hello(name, unit)
  implicit none
  character(len=*), intent(in)  :: name
  integer, intent(in), optional :: unit    ! optional 属性付き引数

  integer :: u

  ! 組込み関数 present で引数が呼出し時に指定されたかどうかを調べることが出来る．
  ! 返値が真なら指定有り，偽なら指定無し（偽の場合はデフォルトの動作を実装）
  if( present(unit) ) then
     u = unit                          ! unit を指定
  else
     u = 6                             ! デフォルトは標準出力
  end if

  write(u,*) 'Hello ', name            ! 表示

  return
end subroutine hello
```

これまで関数やサブルーチンを呼び出す際には定義時の引数並びの順番通りに与えなければならなかった．しかしキーワード引数という機能を用いて，順番を気にせず引数を与えることも可能である．(open 文の使い方を思い出そう.) すなわち，上で定義された hello を呼び出す際に

```
call hello(unit=0, name='Albert') ! 標準エラー出力へ
call hello(name='Einstein')       ! 標準出力へ
```

のように引数を"仮引数名 = 値"という形式で渡すことで，引数の順番を意識せずに呼び出しが出来る．なお，この例では unit は optional 属性付きで宣言されているので省略することも出来る. optional 属性が無い引数については，キーワード引数を用いれば順番は気にしなくて良いが，全ての引数を指定する必要がある．

なお，このようにキーワード引数の機能を用いるには 内部手続き として宣言するか，`外部手続き`_ の場合には interface 宣言で明示的に仮引数名を呼び出し側に知らせてやらなければならない．(やはり外部手続きは面倒である.)

7.5 再帰呼び出し (recursive) †

再帰呼び出しとは，関数やサブルーチンの中で自分自身を呼び出すことである．このような再帰手続は明示的に `recursive` を用いて関数やサブルーチンを定義しなければならない．なお，何も考えずに自分自身を呼び出すと簡単に無限ループになってしまうので，そうならないように注意しよう．例えば以下は階乗の計算をする例である．

リスト 10: sample6.f90 抜粋

```
!
! <<< recursive (再帰的呼び出し) >>>
!
! ちなみに以下の実装で正しい値が得られるのは n = 12 までである．なぜか？
!
recursive function fact(n) result(m) ! recursive を指定する
  implicit none
  integer, intent(in) :: n
  integer :: m

  if(n == 1) then                    ! 無限ループにならないように
     m = 1
  else
     m = n*fact(n-1)                 ! 自分自身を (異なる引数で) 呼び出す
  end if

end function fact
```

ここで $n! = n \times (n-1)!$ という漸化式を用いている．ある種のアルゴリズムは再帰を使うと非常にスッキリと書くことができるので重宝することも多いだろう．ただし関数やサブルーチンの呼び出しそのものにもコスト (時間) がかかるので，不用意に用いるとパフォーマンスのボトルネックになることもあるため注意して欲しい．(基本的には再帰呼び出しを使わない方がパフォーマンスは良くなる場合が多い．)

7.6 内部手続きと外部手続き †

定義場所 では，簡単のため複数あるサブプログラムの定義方法の 1 つのみを扱った．このように定義されたサブプログラムは「内部手続き」と呼ばれるが，これとは別に「外部手続き」なるものも存在する．実際には外部手続きは後で学ぶモジュールを用いてモジュールの内部手続として実装する方が良いのだが，このやり方も知っておいて損は無い．特にモジュール化されていない外部ライブラリを用いる場合や，古い Fortran 77 仕様のプログラムを扱う際には (内部手続きが存在しないため) 外部手続きの理解が必須となる．

7.6.1 内部手続き

定義の仕方は 定義場所 を参照して欲しい．ここで注意しなければならないのは変数のスコープについてである．じつは，**メインプログラム中で宣言した変数には内部手続きからアクセスすることが出来る** (ただし逆は出来ない) ことに注意して欲しい．このことを表すサンプルが sample7.f90 である．これはそれほど長くないので，以下に全体を示す．

リスト 11: sample7.f90

```
program sample
  implicit none

  ! 16 進数変換のためのテーブル (内部手続きからも参照される)
  character(len=1), parameter :: hex_char(0:15) = &
       & (/'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', &
       &   'A', 'B', 'C', 'D', 'E', 'F'/)

  integer :: n = 10
  character(len=8) :: hexstr

  ! どちらの n を参照するか？
  call sub()

  ! 整数を 16 進数に変換して表示
  n = 15*16**6 + 4*16**4 + 3*16**3 + 16**2 + 1
  call decimal2hex(n, hexstr)
  write(*,*) 'decimal = ', n, ' ===> hex = ', hexstr

  stop
contains
  !
  ! 内部手続きのスコープについて (1)
  !
  ! 内部手続きからはメインプログラムで宣言された変数を参照可能．ただし逆は不可．
  !
  ! n という名前の変数をサブルーチン内で宣言するかどうかで挙動が変わる
  subroutine sub()
    implicit none
    ! もし以下の行があればメインプログラムの n とサブプログラムの n は独立
    !integer :: n

    write(*,*) n          ! メインプログラム中の変数 n にアクセス
  end subroutine sub

  !
  ! 内部手続きのスコープについて (2)
```

(次のページに続く)

(前のページからの続き)

```
  !
  ! メインプログラムで定義された変数は内部手続きから参照出来るが，一般論としては
  ! 引数として渡すようにした方が安全である．以下の例のようにプログラム全体で共通
  ! に用いる定数であれば問題は起こらないことが多い．
  !
  ! 10 進数を 16 進数に変換
  subroutine decimal2hex(decimal, hex)
    implicit none
    integer :: decimal
    character(len=*) :: hex

    integer :: i, n, d

    d = decimal
    do i = 1, 8
       n = d / 16**(8-i)
       d = d - n * 16**(8-i)
       ! メインプログラムで宣言された変数 (hex_char) を参照
       hex(i:i) = hex_char(n)
    end do

  end subroutine decimal2hex

end program sample
```

33 行目で内部手続き sub からメインプログラム中に定義された変数 n にアクセスしている．しかし，もし内部手続き sub 中で変数 n が定義されている場合 (31 行目のコメントを外した場合) には，この変数は sub 内部のみで有効な (メインプログラム中の n とは独立な) 変数になる．一般的には，サブプログラムからメインプログラム中の変数を不用意に直接参照するのは間違いのもとになりやすい．それよりは，引数や返値を通じて値のやり取りを明示的に行う方が分かりやすいプログラムとなることが多い．

例外としてはプログラム全体で共通に用いる定数が挙げられる．定数の場合は参照されるだけなので，問題にならない場合が多い．例えば 5-7 行目で宣言している定数配列 hex_char を内部手続き decimal2hex の 56 行目で参照している．この場合は定数を参照するだけで，不用意に値が変更されることがないため，問題になることはあまりないであろう．

まとめると，メインプログラムで定義された変数に内部手続きからアクセスすることができるものの，気をつけて使わないと思わぬ動作を引き起こす可能性があるため，**基本的には関数の返値や関数・サブルーチンの引数を介してデータをやり取りする** 方が間違いが少ないであろう．

7.6.2 外部手続き (非推奨)

外部手続きは program から end program で囲まれた範囲 **以外** に定義される．以下に示す例のように，定義する場所はメインプログラムの前でも後でもどちらでも良い．適切にコンパイル・リンクすれば別ファイルで定義したサブプログラムを用いることも可能である．

リスト 12: sample8.f90

```
!!!!!!!!!! 外部手続きの定義場所 (1) !!!!!!!!!!
function square_ext1(x) result(y)
  implicit none
  real(8) :: x
  real(8) :: y

  y = x**2

  return
end function square_ext1
!!!!!!!!!!

! メインプログラム
program sample
  implicit none
  !
  ! 外部関数の interface 宣言
  ! (本当は後で学ぶモジュールを使うほうがスマート)
  !
  interface
     real(8) function square_ext1(x)
       real(8) :: x
     end function square_ext1
  end interface

  !
  ! 外部サブルーチンの interface 宣言
  !
  interface
     subroutine sub_ext()
     end subroutine sub_ext
  end interface


  ! 外部関数を呼び出すには実はこの書き方でも良いが，色々と問題が多いので非推奨
  real(8), external :: square_ext2
```

(次のページに続く)

(前のページからの続き)

```
  ! 外部関数呼び出し
  write(*,*) square_ext1(2.0_8), square_ext2(4.0_8)

  ! 外部サブルーチン呼び出し
  call sub_ext()

  stop
end program sample

!!!!!!!!!! 外部手続きの定義場所 (2) !!!!!!!!!!
function square_ext2(x) result(y)
  implicit none
  real(8) :: x
  real(8) :: y

  y = x**2

  return
end function square_ext2

subroutine sub_ext()
  implicit none

  write(*,*) 'sub_ext'

  return
end subroutine sub_ext
!!!!!!!!!!
```

内部手続きとは異なり，外部手続きを使う場合にはメインプログラムで使用する外部手続きを明示的に宣言する必要がある．これは square_ext1 については 20-24 行目，sub_ext については 29-32 行目のように interface を用いて行なっている． interface では関数やサブルーチンの呼び出し形式のみ (引数のデータ型やその順番，関数の返値など) を宣言する[6]．

実際にはもう少しサボることが出来てしまう．関数については squre_ext2 は 36 行目で返値のみを external 属性をつけて宣言することで呼び出しができる．サブルーチンについては実は特に何も宣言しなくても呼び出しが可能である．しかし，ここでは interface を使って明示的に外部手続きを宣言すること強く推奨しておきたい．なぜなら，メインプログラムの外で定義された関数やサブルーチンについてはコンパイラが (引数の数や型などの) 呼び出し形式を知る方法が無いため，間違った呼び出し方をしていてもコンパイルが通ってしまう．しかし不正な呼び出しをしているため，当然実行時にはエラーが発生してプログラムが異常終了することになる．一般的に実行時のエラーの方がコンパイルエラーよりも厄介でデバッグにも時間がかかるため，コンパイル時にチェックが可能な interface による宣言の方が良いのである[7]．(内部手続きでは文字通りメインプログラムの内部に定義されているので，コンパイラがメインプログラムをコ

[6] C 言語で言うところのプロトタイプ宣言である．

[7] 初心者の頃はコンパイルエラーに辟易とすることが常であるが，コンパイルエラーでは一応コンパイラが (大変分かりにくくはあるものの) エラーメッセージを出力してくれるのに対して，実行時のエラーは通常何のヒントにもならない無情な Segmentation fault のみである．

ンパイルする際に呼び出し形式のチェックが可能である.)

とにかく外部手続きは (行儀よく使おうと思うと) 面倒なので，特に理由が無い限りは内部手続きを用いる方が良い．どうしてもメインプログラムの外で手続きを定義する必要がある場合には後で学ぶモジュールを用いる方が間違いが圧倒的に少ないのである．

7.7 第 7 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例
- 課題 6 解答例
- 課題 7 解答例

7.7.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

7.7.2 課題 2

与えられた倍精度実数 $a(>0)$ の平方根の近似値を返す関数を実装せよ．ただし平方根は以下のような逐次近似で計算するものとする．

$$x_{n+1} = \frac{1}{2}\left(\frac{a}{x_n} + x_n\right)$$

ここで x_n は $\sqrt{a}$ の n 番目の近似値である．初期値としては $x_0 = a$ を与え，反復は例えば $\epsilon = 10^{-5}$ に対して， $\|x_{n+1} - x_n\| < \epsilon\|x_n\|$ となるまで繰り返せば良い．(反復の途中結果は必要ないので x_n に配列を使う必要はないことに注意せよ.)

実装した関数と組み込み関数 `sqrt` の結果を比較し， ϵ で与えた精度の範囲内で正しいことを確認すること．例えば，以下は標準入力から与えられた数値 (この例では 2.0) の平方根を計算して表示する例である．

```
2
    ( ) =     1
approx  =     1
```

7.7.3 課題 3

テストの点数が整数配列から与えられた時にヒストグラムを作成するサブルーチン histogram を実装せよ．例えば点数配列とビン幅を入力とし，作成されたヒストグラムの各ビンの中央値，各ビン内の人数を出力とする以下の様な形式のサブルーチンを作成すればよい．ただし，与える整数は $0 \leq n < 100$ とするが，もしこの範囲を超えた入力があった場合にはエラーを表示して終了すること．

```
subroutine histogram(score, binw, binc, hist)
  implicit none
  integer, intent(in)  :: score(:)  ! 点数 (人数分)
  integer, intent(in)  :: binw      ! ビンの幅 (例えば 10 点)
  real(8), intent(out) :: binc(:)   ! ビンの中央値 (例えば 5, 15, ..., 95)
  integer, intent(out) :: hist(:)   ! 各ビン内の人数

  ! ここでヒストグラムを作成

end subroutine histogram
```

この場合のように固定幅のビンでヒストグラムを作成するのは簡単である． i 番目の点数がヒストグラムの j 番目の要素に入るとすると， j は

```
j = score(i) / binw + 1
```

のように求められる (複雑な if 文による分岐は必要ない!)．このインデックスが配列 hist の上限と下限の間に収まっていなければエラーとすれば良い．

このサブルーチンを用いて， score2.dat からデータを読み込みヒストグラム作成するプログラムを作成せよ．またその結果を gnuplot で図示せよ (with boxes を用いると良い)．出力結果は例えばビン幅を 10 とした場合には以下のようになる．

```
  5                                0
 15                                3
 25                               26
 35                               63
 45                              152
 55                              248
 65                              254
 75                              159
 85                               80
 95                               15
```

なお score2.dat の形式は 5 章の課題 3 の score1.dat と同じであるが，データは異なるものになっている．

同様に範囲外のデータを含む score3.dat を読みこませると

```
```

のようなエラーを表示するように実装せよ．

7.7.4 課題 4

データ $x_1, x_2, \ldots, x_n$ を大きさの順に並べ替える処理をソートという．ソートのアルゴリズムの中でも一番簡単なのがバブルソートと呼ばれるものである．これは以下の処理を $n-1$ 回繰り返すことで実現される．

$i = 1, \ldots, n-1$ まで順に x_i と x_{i+1} の大小を比較し，$x_i > x_{i+1}$ なら順番を入れ替える

この処理を k 回行うと x_i のうち k 番目に大きい要素が x_{n-k+1} に配置されるので，$n-1$ 回繰り返すことで全ての要素を並び替えることが出来る．

ただし 1 回目の処理が終了した時点で最大値が x_n になっており，この要素については並べ替えの必要がない．従って 2 回目は $x_1, \ldots, x_{n-2}$ までを処理すれば十分である．同様に m 回目の処理が終了した時点で $x_{n-m+1}, \ldots, x_n$ までの位置は確定しているので，$m+1$ 回目には $x_1, \ldots, x_{n-m}$ までの処理を行えばよい．これによって比較回数を $(n-1)^2$ 回から半分に減らすことができる．

このバブルソートによって整数配列をソートするサブルーチン bsort を実装せよ．これは例えば以下のような形になるだろう．

```
subroutine bsort(array)
  implicit none
  integer, intent(inout) :: array(:) ! 配列にはソートされた結果が代入される

  ! バブルソート

end subroutine bsort
```

作成したプログラムを用いて rand.dat のデータをソートし，結果が正しいことを確認せよ．ここでもデータファイルの形式は上の score2.dat と同一である．従って同様にリダイレクトで

```
```

のように読みこませれば良い．

7.7.5 課題 5

データ $x_1, x_2, \ldots, x_n$ からある値 X に等しいものを探しだす処理を探索という．その中でも以下は二分探索と呼ばれるアルゴリズムである．

- まず $x_i (i = 1, \ldots, n)$ をソートする．
- $l = 1, r = n$ として以下の処理を繰り返す．
  1. $l > r$ ならば失敗 (見つからなかったので処理を終える)．
  2. $m = (l + r)/2$ とし，$X = x_m$ なら成功 (見つかったので処理を終える)．
  3. $X > x_m$ なら $l = m + 1$ ，$X < x_m$ なら $r = m - 1$ として [1] に戻る．

これをサブルーチン bsearch として実装せよ．

また open 文で rand.dat からデータを読み込み， bsort で配列をソートした後に bsearch で実際に適当な値の探索を行うプログラムを作成せよ．例えば標準入力から与えられた値をソート後の配列から探索を行い，結果を表示するようにすれば良い．この時の実行結果は以下のようになる．(この例では 10 や 2004 が入力値である．)

```
                10    # キーボード入力
    10


                2004  # キーボード入力
  2004                        100
```

なおサブルーチン bsearch は例えば以下のような形式とすればよい．見つかった場合には探しだす値と等しい値が格納されている配列のインデックスを，見つからなかった場合には -1 を仮引数 idx に代入して返すようにするとよいだろう．

```
subroutine bsearch(array, var, idx)
  implicit none
  integer, intent(in)  :: array(:) ! ソートされた配列
  integer, intent(in)  :: var      ! 探したい値
  integer, intent(out) :: idx      ! 見つかった要素へのインデックス

  ! 二分探索

end subroutine bsearch
```

7.7.6 課題 6

1 行に 1 つの英単語が記述されているファイルを読み込み，英単語を辞書順にソートして出力するプログラムを作成せよ．空白行かファイルの終端に達するまで全ての単語を読み込むこと．ただし英単語は 10 文字以下，また読み込む単語の数は 100 個以下であると仮定してよい． words.txt を入力として作成したプログラムの動作を確認せよ．

実行結果は例えば以下のようになるだろう．(空白行の後の NOT_TO_BE_READ は読まれていないことに注意!)

```

```

なお，文字列の大小比較は辞書順となる (例えば 'apple' < 'banana') ので， bsort を一部修正するだけで文字列のソートが出来る．文字型変数の配列は例えば

```
character(len=10) :: char_array(100)
```

などのように宣言すれば良い．ここで char_array は 10 文字分の文字列を保持する長さ 100 の配列である．

7.7.7 課題 7 †

与えられたファイルに含まれるアルファベットの各文字の出現回数をヒストグラムとして標準出力に表示するプログラムを作成せよ．ファイルの長さは任意とする．即ちファイルの終端まで全ての文字を読み込まなければならない (空白行があっても読み込み続ける)．ただし以下の条件を満たすこと．

- アルファベット a-z，A-Z 以外は無視してよい．
- 大文字と小文字は区別しない．
- 文字数があまりに多い時にはヒストグラムを適当に規格化すること．

wikipedia.txt (Wikipedia より引用) を処理した時には例えば以下のような出力となるだろう．この例では最大で 60 個の 'o' が出力されるように規格化してある．各アルファベットの後の括弧内の数字は文字数を表している．(wikipedia.txt で試す前に words.txt などの小さいデータで試した方が良いだろう．)

```
( 110)
(   9)
(  44)
(  32)
(  99)
(  31)
(  38)
(  32)
(  87)
(   2)
(   2)
(  38)
(  49)
(  91)
(  81)
(  39)
(   0)
(  93)
(  63)
(  79)
(  37)
(  14)
(   5)
(   1)
(  15)
(   0)
```

組込み関数 ichar , char を用いるとよい．また規格化の際には四捨五入をする関数 nint を用いることが出来る．

特に仕様は限定しないが，関数やサブルーチンを使った分り易いプログラムを目指すこと．

第8章　数値解析の基礎

これまでは主にプログラミングの作法を学んできた．基本的には現在までの知識を組み合わせれば原理的にはどんな問題にも対応できるようになっている[1]．そこで，これまでに学んだ知識を用いてもう少し実践的な内容に取り組もう．具体的には非線形方程式の求根法や，関数の数値積分，また乱数の使い方などを扱う．

参考:

- sample1.f90 : 桁落ちと情報落ち
- sample2.f90 : 二分法
- sample3.f90 : Newton 法
- sample4.f90 : 数値積分
- sample5.f90 : 乱数

この章の内容

8.1 実数の精度と誤差

これまで何気なく用いてきた実数型だが，数値解析を始める前に計算機における実数の取り扱いやその誤差について理解しておこう．

8.1.1 浮動小数点数の表現

実数は計算機の内部では浮動小数点数と呼ばれる形式になっており，

$$x = \underbrace{(-1)^s}_{\text{符号}} \times \underbrace{(1.f)}_{\text{仮数部 (2 進表現)}} \times \underbrace{(2^{e-127})}_{\text{指数部}}$$

のような表現で表される．仮数部は有効桁を決める部分であり，指数部は絶対値を調整するためのものである． 仮数部が $1.f$ となっているのは (最上位ビットは常に 1 なので) 実質 1 ビット分だけ精度を稼ぐため

[1] もちろん数学や物理，更には数値解析の知識は必要になってくるわけだが，それはこれから学んでいくことになる．

である (これをケチ表現と呼ぶ). 例えば標準的な規格 (IEEE754 規格) では単精度の場合，符号部に 1 ビット，仮数部に 23 ビット，指数部に 8 ビットが割り当てられている． 指数部が 8 ビットであることから，$e = 0, 1, \ldots, 255$ であり，即ち絶対値の範囲としては $2^{-127} \sim 10^{-38} \lesssim x \lesssim 2^{127} \sim 10^{+38}$ 程度以内の数値しか表現できない． また $\log_{10}(2^{23+1}) \sim 7.2$ なので 10 進での有効桁数は 7 桁程度である． 同様に倍精度では符号部に 1 ビット，仮数部に 52 ビット，指数部に 11 ビットが割り当てられていることから絶対値の範囲は $2^{-1024} \sim 10^{-308} \lesssim x \lesssim 2^{+1024} \sim 10^{+308}$，有効桁数は $\log_{10}(2^{52+1}) \sim 15.9$ となり 10 進での有効桁数は 16 桁程度である．

8.1.2 丸め誤差

実数を有効桁で「丸め」て表現することから生じる誤差を丸め誤差と呼ぶ．これは場合によっては求めたい計算結果の精度に悪影響を及ぼすこともあるため，注意が必要である．

- 桁落ち

  絶対値の非常に近い 2 つの数の差を計算すると絶対値が非常に小さくなり，その分だけ相対誤差が大きくなってしまう．これを桁落ちと呼ぶ．
  例えば $\sqrt{1+x} - 1$ のような演算は x が非常に小さい場合にはその誤差が無視できない．簡単のため 10 進数で有効桁数が 5 桁の場合を考えよう． $x = 0.001$ とすると， この精度の範囲では $\sqrt{1+x} \approx 1.0005$ なので， $\sqrt{1+x} - 1 \approx 0.0005$ となり有効桁数が 1 桁に低下してしまう．これは例えば以下のような式変形によって減算を無くすことで回避が可能である．

  $$\sqrt{1+x} - 1 = \frac{x}{\sqrt{1+x} + 1}$$

- 情報落ち

  絶対値の大きい数に小さい数を加えてもほとんど変化が無い．これを情報落ちと呼ぶ．
  同様に有効桁数が 5 桁の場合を考えよう．$a = 1.0000$， $b = 1.0000 \times 10^{-5}$ はどちらとも 5 桁の有効桁数を持つが， $a + b = 1.00001$ の小数点第 5 位は精度は有効桁数の範囲外となるため，情報が失われてしまう．例えば，総和計算の際に非常に大きな数と小さな数を多数加える場合にはこれが問題となることがある．このときには小さい方から順に和を計算することで回避できる．

これらを具体的に示した サンプルコード参照 の実行結果は

```
0
0

0
0
```

のようになる．3 行目と 4 行目が桁落ち対策ありとなしの比較，6 行目と 7 行目が情報落ちの対策ありとなしの比較である．このように，数学的に等価な演算であっても無視できないような誤差が発生する可能性があることは認識しておくべきである．なお，丸め誤差の影響を調べるのに一番安直だが確実な方法は単精度から倍精度，倍精度から 4 倍精度に精度を上げてみて結果が変わらないことを確認することである．

8.2 求根法

解析的には解けない方程式 (非線形方程式) の解を求める方法を考えよう．通常は右辺と左辺の両方に式があるわけだが，移項してしまって

$$f(x) = 0$$

という一般的な形にして解くことにしよう．実はこの問題は意外と難しい問題であり，どんな問題にも使うことのできる汎用的で，かつ高速に解を求められるような手法は存在しない．と言うのはどんな手法であっても基本的には初期値として近似解のあたりを付けて，少しずつ真の解に近づけていく反復法だからである．初期値の選び方によっては正しい解に収束しない場合や，欲しい解 (例えば物理的な解) に収束しないもあり得る[2]．計算機は初期値までは面倒を見てくれないので，人間が適切な初期値を与えてあげなければならない．とりあえず初期値は与えられたとして，そこから近似解を反復によって求めるアルゴリズムを見ていこう．

8.2.1 二分法

もしある区間に解が 1 つあると分かっているならば，反復によって近似解が必ず真の解に収束する，二分法 (bisection method) と呼ばれるアルゴリズムが知られいる．まず $[x_1, x_2]$ $(x_1 < x_2)$ に対して $f(x_1) < 0, f(x_2) > 0$ ならばこの区間に解があることが分かる．このとき，二分法によって近似解を求める手順は以下のようなものとなる．

1. $x = (x_1 + x_2)/2$ を近似解とする．
2. $f(x) < 0$ なら $x_1 = x$，$f(x) > 0$ なら $x_2 = x$ とし，(1) に戻る．

この手順を解が収束するまで繰り返せばよい．収束判定は例えば許容誤差を ϵ として $|x_2 - x_1| < \epsilon$ (区間の幅が許容誤差よりも小さい) などとすればよい．

実際には $f(x_1) > 0, f(x_2) < 0$ の場合も考慮しなければならないが，組み込み関数 `sign` を用いてこれは簡単に実現出来る．以下のコードには二分法の実装例を示している．ただし `f` は関数として定義されているものとする．

リスト 1: sample2.f90 抜粋

```
  sig = sign(1.0_8, f(x2)-f(x1))
  status = .false.
do n = 1, nmax
   x = (x1 + x2) * 0.5_8
   y = f(x)

   ! 収束判定
   if (abs(x2-x1) < tolerance) then
      status = .true.
      write(*,fmt='("Converged at step ", i4)') n
      exit
```

(次のページに続く)

[2] 1 変数ならまだ良いのだが，多変数関数に拡張するとほとんどお手上げである．

(前のページからの続き)

```
    else
       write(*,fmt='("Error at step ", i4, " = ", e20.8)') n, abs(y)
    end if

    ! 次の値を推定
    if (y*sig < 0.0) then
       x1 = x
    else
       x2 = x
    end if
  end do
```

8.2.2 Newton 法

二分法は解の含まれる範囲を正しく指定すれば必ず収束するという利点はあるものの，あまり収束の速いアルゴリズムではなかった．一方で，初期値によっては収束しないかもしれないが，収束するならばその収束自体は速いというアルゴリズムも考えられる．それがここで紹介する Newton 法と呼ばれるものである．これは $f(x)$ に加えてその微分 $f'(x)$ も用いるのが特徴である．すなわち，近似解 x が与えられたときに x の周りでのテイラー展開した

$$f(x+\delta) \simeq f(x) + \delta f'(x) + O(\delta^2)$$

を用いて， $f(x+\delta) = 0$ とすると

$$\delta = -\frac{f(x)}{f'(x)}$$

を得る．即ち $x - f(x)/f'(x)$ を新しい近似解として採用すればよい．大きな特徴は関数の値だけではなく，その微分値 (接線) も用いて収束を加速している点である．しかし，当然ながら初期値によっては収束しないことも十分に考えられる (どういった場合であろうか？)．以下のコードは Newton 法のアルゴリズムを実装したものである．プログラムの構造は二分法の場合とほぼ同様であるが，微分値を返す関数 df も用いている．

リスト 2: sample3.f90 抜粋

```
  status = .false.
  do n = 1, nmax
     ! 次の値の推定
     y  = f(x)
     dy = df(x)
     dx =-y / dy
     x  = x + dx

     ! 収束判定
     if (abs(dx) < tolerance) then
        status = .true.
```

(次のページに続く)

(前のページからの続き)

```
      write(*,fmt='("Converged at step ", i4)') n
      exit
    else
      write(*,fmt='("Error at step ", i4, " = ", e20.8)') n, abs(y)
    end if
  end do
```

8.3 数値積分

次に関数の積分

$$S = \int_a^b f(x)dx$$

の数値的な評価を考えよう．区分求積法の原理を思い出せば，積分領域 $[a, b]$ を小さな領域 $h = (b - a)/N$ に分割し，積分を微小区間の積分の総和で近似すればよいことが分かるだろう．ここで分割数 N を十分大きくとることができれば，近似式の誤差は十分小さく抑えることができる．$x_j = a + jh$ とし，微小区間の端点 $[x_i, x_{i+1}]$ で与えられた関数値 f_i, f_{i+1} から，関数系を

$$f(x) = \frac{f_{i+1} - f_i}{h}(x - x_i) + f_i$$

のように線形近似することで，以下の **台形公式** が得られる．

$$S = \frac{h}{2}\left[f(x_0) + 2\sum_{j=1}^{N-1} f(x_j) + f(x_N)\right] + O(h^2).$$

ここで $O(h^2)$ は誤差が刻み幅 h の 2 乗で小さくなることを意味する．ただし例外として，元の関数系が線形であれば，当然この評価は厳密な積分値を与える．

この考え方をさらに発展させ，x_{i-1}, x_i, x_{i+1} の 3 点の関数値 f_{i-1}, f_i, f_{i+1} から関数系を 2 次関数で近似すれば，以下の *Simpson* **の公式** が得られる．

$$S = \frac{h}{3}\left[f(x_0) + 4\sum_{j=1}^{N/2} f(x_{2j-1}) + 2\sum_{j=1}^{N/2-1} f(x_{2j}) + f(x_N)\right] + O(h^4).$$

ここで Simpson の公式の誤差は h の 4 乗に比例する．当然ながら同じ精度を実現するために必要な計算量は Simpson の公式の方が台形公式よりも小さくて済む．

Fortran プログラム中では $f(x)$ を関数として定義し，分割数 N を適当に定めれば do ループによって総和計算をすることで積分値は簡単に求まる．例えば台形公式であれば

リスト 3: sample4.f90 抜粋

```
  !
  ! 台形公式による数値積分
  !
  integral = 0.5_8 * (f(x1) + f(x2))
```

(次のページに続く)

(前のページからの続き)

```
do n = 1, nmax-1
   integral = integral + f(x1 + dx*real(n,8))
end do
integral = integral * dx
```

Simpson の公式を使った場合であれば

リスト 4: sample4.f90 抜粋

```
!
! Simpson の公式による数値積分
!
integral = f(x1) + f(x2)
! odd
do n = 1, nmax-1, 2
   integral = integral + 4.0_8 * f(x1 + dx*real(n,8))
end do
! even
do n = 2, nmax-2, 2
   integral = integral + 2.0_8 * f(x1 + dx*real(n,8))
end do
integral = integral * dx / 3.0_8
```

のように実装することが出来る．ここで `f(x)` は被積分関数である．

8.4 乱数

確率的な現象を計算機を用いて模擬する際には乱数を用いることになる．ただし計算機で用いることのできる乱数は擬似乱数と呼ばれ，乱数のように見えるが実際には決定論的な手法に基づき生成される数列である．従って質の良い (周期の長い) 乱数を用いなければ，用途によっては乱数とみなすことのできない場合もあるため注意が必要である．

Fortran には乱数を発生させる組込みのサブルーチン `random_number` が存在する．

```
call random_number(x)
```

とすれば `x` に区間 $[0,1)$ の一様乱数が代入される．`x` は実数型 (単精度もしくは倍精度) であれば配列でも良い．配列の場合は全ての要素に一様乱数が代入される．

擬似乱数は決定論的な数列であることは既に述べた通りであるが，その初期値を指定することも出来る．これは乱数のシード (seed) と呼ばれ，組込みのサブルーチン `random_seed` を用いて行うことが出来る．使い方は

1. シードを格納領域のサイズを取得 (サイズはコンパイラ依存)
2. 必要な領域を確保 (`allocate` を用いる)

3. シードを指定

といった流れとなる．以下のサブルーチン `random_seed_clock` は計算機の時刻[3] に応じてシードを指定するものであり，これを用いれば実行する度に (時刻が異なるので) 得られる乱数値が異なることが保証される．逆に固定のシードを用いるようにしておくと毎回同じ結果が得られるため，乱数を用いるプログラムをデバッグする際には都合が良い．

リスト 5: sample5.f90 抜粋

```
!
! 乱数の seed をシステムクロックに応じて変更
!
subroutine random_seed_clock()
  implicit none
  integer :: nseed, clock
  integer, allocatable :: seed(:)

  ! システムクロックを取得
  call system_clock(clock)

  call random_seed(size=nseed)
  allocate(seed(nseed))

  seed = clock
  call random_seed(put=seed)

  deallocate(seed)
end subroutine random_seed_clock
```

なお， `random_seed_clock` を使った乱数シードの初期化は sample5.f90 の例のようにプログラムの実行の最初に一度だけ行えば十分である．

8.5 第 8 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例
- 課題 6 解答例

[3] Unix 系 OS の場合は 1970 年 1 月 1 日からの経過時間.

8.5.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

8.5.2 課題 2

2 次方程式 $ax^2 + bx + c = 0$ の解は以下の解の公式を用いて求めることが出来る．

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

しかしこの公式を単純に用いた場合には $b^2 \gg 4ac$ の時には，桁落ちによってどちらか一方の解の精度が悪くなってしまう．このことを実際に確認し，式変形によってその精度を改善せよ．

例えば $a = 1, b = -10^9, c = 1$ とした時の解は

$$x \simeq 1.\underbrace{000000000000000}_{0 \text{ が } 15 \text{ 個}} \times 10^9, \quad 1.\underbrace{000000000000000}_{0 \text{ が } 15 \text{ 個}} \times 10^{-9}$$

であるが，解の公式を用いた場合と式変形によって桁落ち対策をした場合で数値解を比較せよ．(桁落ち対策が必要なのはどちらか一方の解のみである．)

8.5.3 課題 3

Newton 法では関数の微分値を解析的に与えて用いるが，割線法 (Secant method) と呼ばれる手法では連続した 2 つの近似解を用いて，微分を差分で近似する．即ち n 番目の近似解を x_n と書いたときに，Newton 法の公式

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

において，微分値を以下のように近似する．

$$f'(x_n) \simeq \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

(この手法では微分値の計算をする必要がないため，反復 1 回あたりの計算量が少なくなる可能性があるが，収束は Newton 法よりも少し遅くなる．)

この割線法を実装し，以下の方程式

$$f(x) = \frac{1 - x}{\sqrt{M}} - \exp(x)$$

を数値的に解くことで，その収束の速さを二分法および Newton 法と比較せよ．ただし $M = 1836$ とする．なお解は $x \simeq -2.5$ なので二分法ではこの前後に 2 つの初期値を指定すればよい．割線法も 2 つ初期値が必要になるが，(この関数に関しては) 初期値には敏感ではないので，例えば 0 と 1 を初期値として用いればよい．ここでは収束の速さを比較すればよいので，各反復ごとの収束判定はせずに 20 回程度の反復を行い誤差の減少の様子を gnuplot を用いて図示せよ．ただし連続する 2 つの近似解の差を誤差と定義する．

結果は以下の例ようになるであろう．

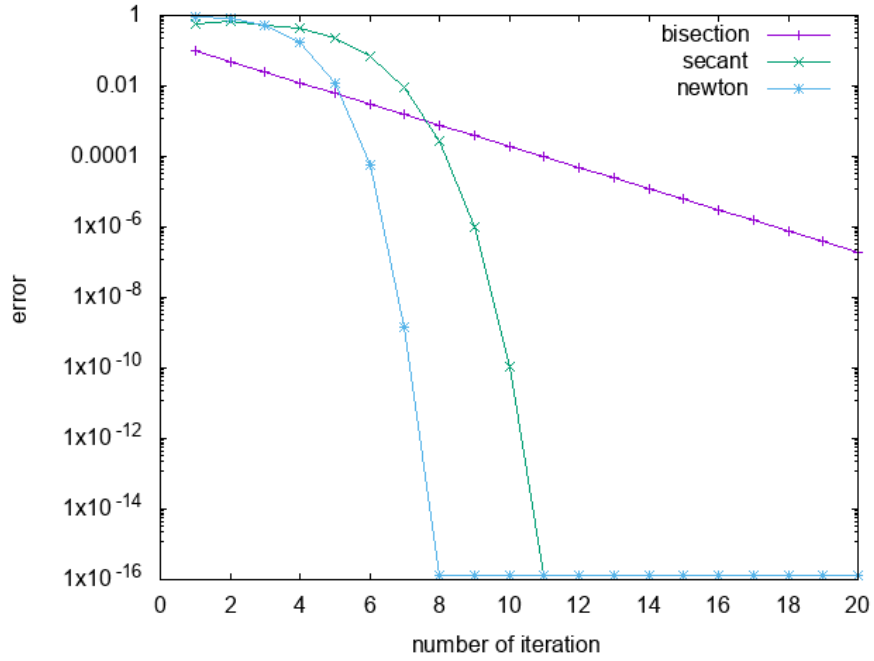

図 1: 求根法の収束比較

8.5.4 課題 4

$f(x) = \dfrac{4}{\pi}\dfrac{1}{1+x^2}$ および $f(x) = (n+1)\,x^n$ $(n = 0, \ldots, 5)$ のそれぞれについて

$$\int_0^1 f(x)dx$$

の積分を台形公式および Simpson の公式で数値的に行い，誤差の分割数に対する依存性を gnuplot を用いて図示せよ．例えば分割数を $2^n (n = 1, 2, \ldots, 16)$ の範囲で変えて依存性を調べれば良い．

例として $f(x) = \dfrac{4}{\pi}\dfrac{1}{1+x^2}$ について誤差をプロットすると以下のようになるであろう．

このような依存性となる理由を考えよう．

8.5.5 課題 5

一様乱数を用いてある確率分布に従う乱数を発生させるために逆関数を用いる方法 (変換法や逆関数法などと呼ばれる) を考える．区間 $[0, 1)$ での一様乱数 x およびその確率分布 $f(x)$，また必要な乱数 y とその確率分布 $g(y)$ とする．ただし乱数の値域は $a \leq y < b$ とする． y を x から何らかの変換で $y = P(x)$ のように表すとき，確率密度の保存から

$$g(y)dy = f(x)dx = dx$$

が成り立つ．(最後の等式は一様乱数であることから自明である．) この両辺を積分して

$$x = \frac{\int_a^y g(y')dy'}{\int_a^b g(y')dy'} \equiv G(y)$$

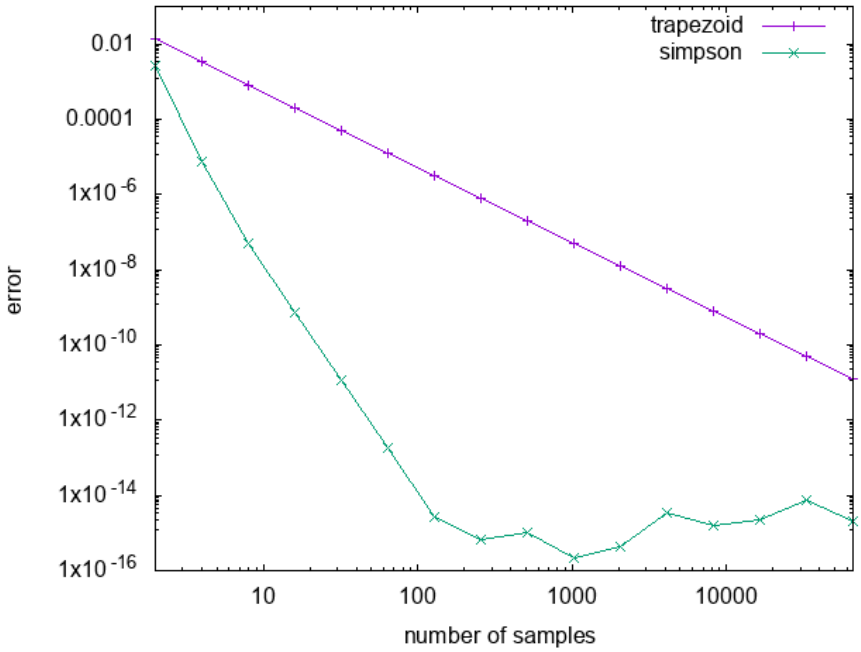

図 2: 数値積分誤差の分割数依存性

によって関数 $G(y)$ を定義する．ただし $G(a) = 0$ ， $G(b) = 1$ となるように規格化した．これより

$$y = G^{-1}(x)$$

を得る．即ち， $G^{-1}(x)$ を解析的に求めることができれば，一様乱数 x を用いて必要な確率分布 $g(y)$ に従う乱数を作ることができる．（以下の図を参照）

このことを用いて指数分布

$$g(y;\lambda) = \lambda \exp(-\lambda y)$$

に従う乱数分布を発生させるプログラムを作成せよ．また分布のヒストグラムを作成し，gnuplot を用いてヒストグラムを解析的な分布と共に図示し，乱数の発生数を増やした時に乱数分布が真の分布に近づくことを確かめよ．

以下は 60000 個の乱数を発生させた場合のヒストグラムの例である．

8.5.6 課題 6 †

乱数を利用した数値積分法としてモンテカルロ法が知られている．これを用いて n 次元ユークリッド空間における単位超球の体積 V_n を求めるプログラムを作成せよ．

ただし n 次元超球の体積は n 次元空間の座標を $x_i\,(i = 1, \ldots, n)$ とし，

$$\sum_{i=1}^{n} x_i^2 \leq 1$$

なる領域の体積と定義される．ここで対称性から $0 \leq x_i \leq 1$ の領域のみを考えれば，この体積は $V_n/2^n$ となる．従って n 個の一様乱数 $0 \leq x_i < 1\,(i = 1, \ldots, n)$ を発生させ，上式の条件を満足するかどうかを調べる試行を多数回行い，その確率を求めることによって $V_n/2^n$ を推定すればよい．

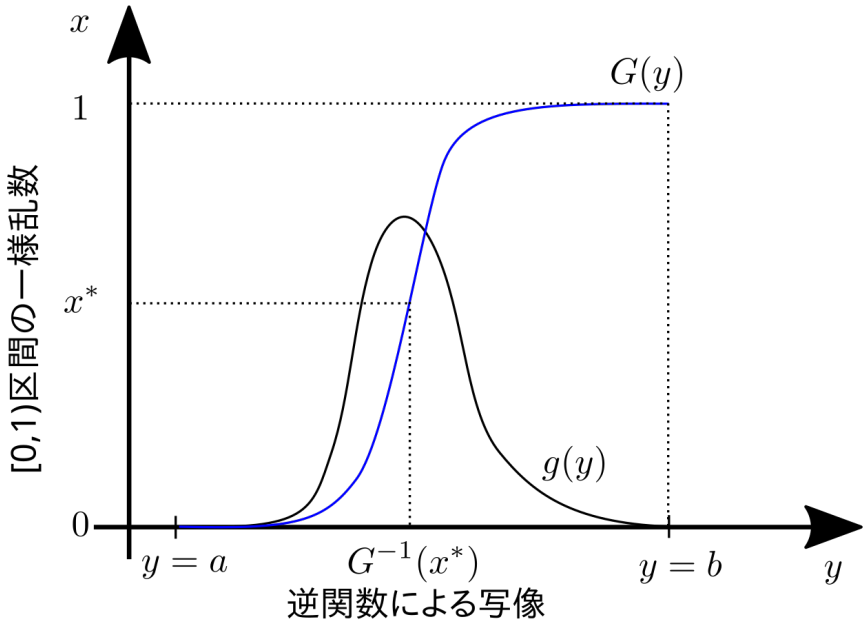

図 3: 逆関数法の概念図

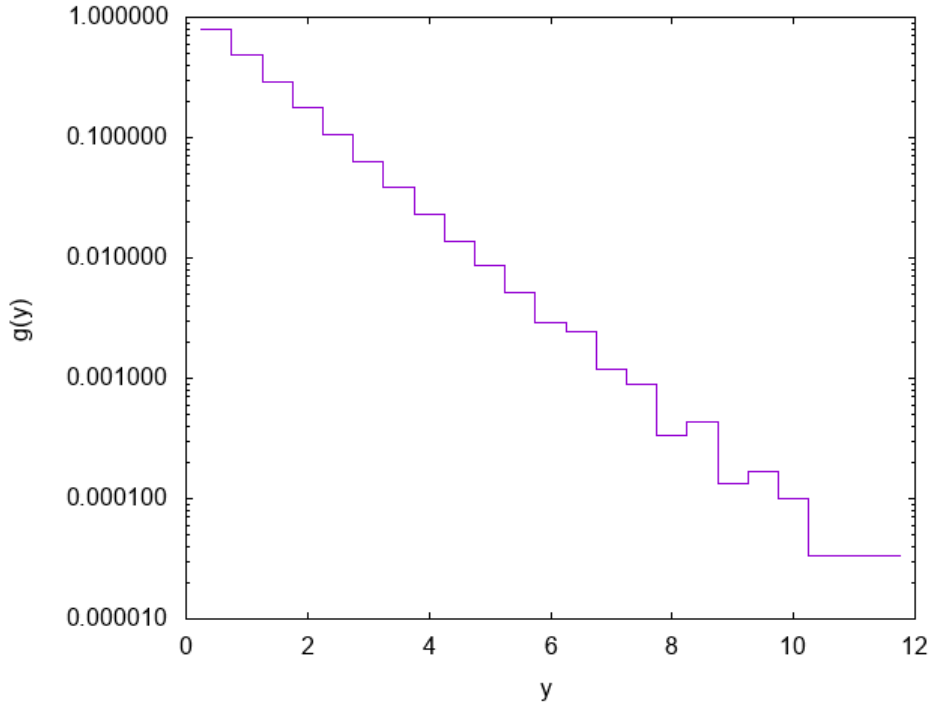

図 4: 指数分布のヒストグラム

なお真の値は

$$V_n = \frac{\pi^{n/2}}{\Gamma(n/2+1)}$$

によって与えられる．

例えば次元数と試行回数を標準入力から与える形式のプログラムの実行結果は以下のようになる．この例のように誤差は試行回数 m に対して， $1/\sqrt{m}$ に比例して減少する．

```
2 1000    # キーボード入力 (2 次元，試行回数 1000 回)
approximation  =       0
      value    =       0
        error  =       0

2 100000  # キーボード入力 (2 次元，試行回数 100000 回)
approximation  =       0
      value    =       0
        error  =       0
```

第9章　モジュールと構造型

これまでに既に 関数とサブルーチン ではプログラムの開発を容易にするための手段として，関数やサブルーチンといったサブプログラムを用いる方法を学んだ．これらサブプログラムは機能を分割し，1 つの独立したプログラム単位として扱われる．ところがプログラムの規模が大きくなってくると，関数やサブルーチン群を用いるだけでは必ずしも十分とは言えなくなって来る．そのような場合に便利になってくる **モジュール** という機能について学ぼう．

モジュールも独立したサブプログラムであるが，関数やサブルーチンなどよりも **1 段階大きなプログラム単位** として考えることができる．すなわち，複数の関連する機能を提供する関数やサブルーチン群を 1 つのモジュールにまとめて提供することが出来る．さらに関数やサブルーチンと大きく異なり，モジュール内部に宣言された変数に外部からアクセスすることも出来るため，複数の変数群をまとめる役割も果たす．またモジュールと共に用いると便利な **構造型** の使い方も身につけよう．構造型はいくつかのデータを 1 つにまとめて，新しいユーザー定義型を提供する機能である．

参考:

- sample1.f90 : 定数や変数の参照
- sample2.f90 : 内部手続き
- sample3.f90 : 総称名
- sample4.f90 : アクセス制限
- sample5.f90 : 構造型と演算子オーバーロード

この章の内容

9.1 モジュールの基本

既に述べたようにモジュールは 1 つの独立したプログラム単位である．その特徴は以下の様な点である．

- モジュール内で変数宣言が出来る．宣言された変数はモジュール内部からはもちろん他のモジュールやメインプログラムから使用することが出来る．
- 複数の関数やサブルーチンをモジュール内で内部手続きとして定義することが出来る．内部手続きはモジュール内部の他の内部手続きや，モジュール外部から呼び出すことも出来る．

モジュールの定義は以下のような形で行う．

```
module name_of_module
  implicit none

  ! 変数宣言など

contains

  ! 内部手続きの定義

end module name_of_module
```

モジュールの構造はメインプログラムと非常に似ており，module で始まり，end module で終わる．また内部手続きは contains から end module の間に定義する．ただしメインプログラムに記述されたコードは上から順に実行されていくのに対して，モジュール内 (implicit none から contains までの間) には実行文は記述せず，変数宣言などを行うだけである．内部手続きについても明示的に呼び出されない限りは実行されることは無い．

定義したモジュールはメインプログラムや，サブルーチン，関数，または他のモジュールから参照して用いることが出来る．使い方は implicit none を記述する前に use によって使いたいモジュールを記述するだけである．

例えばメインプログラムからモジュールを使用するには

```
program name_of_program
  use name_of_module
  implicit none

  ! メインプログラムの処理

end program name_of_program
```

というような形となる．

9.2 変数や定数の参照

サンプルコード参照

モジュールを用いると変数や定数の宣言を共通化し，異なるモジュールやメインプログラムから利用することが出来る．大規模プログラムで共通の変数が複数のモジュールなどから参照される場合には変数宣言を共通化しておくと良い．具体的には以下の例の様になる．

```
! モジュールの定義
module mod_variable
  implicit none

  ! 定数の定義
  real(8), parameter :: light_speed = 2.998e+08 ! 光速 [m/s]
  real(8), parameter :: kboltzmann  = 1.381e-23 ! Boltzmann 定数 [J/K]
  real(8), parameter :: hplanck     = 6.626e-34 ! Planck 定数 [J s]

  ! 変数
  real(8), save :: x, y, z

end module mod_variable

! メインプログラム
program sample
  use mod_variable
  implicit none

  ! 定数の値は参照のみ可能
  write(*, '(a20, ":", e12.4)') 'speed of light', light_speed
  write(*, '(a20, ":", e12.4)') 'Boltzmann constant', kboltzmann
  write(*, '(a20, ":", e12.4)') 'Planck constant', hplanck

  ! これはできない (コンパイルエラー)
  !light_speed = 1.0_8

  ! 変数の値は変更可能
  x = 1.0
  y = 0.0
  z = 0.0

  stop
end program sample
```

プログラム実行中に常にどこからでもアクセスできる変数を **グローバル変数**，サブルーチンや関数の内部でしか用いない変数を **ローカル変数** などと呼ぶことがある．上の例ではモジュールの内部変数がグローバル変数として用いられている．この例のように，一般にはモジュール変数をグローバル変数として用いるには save 属性を付けておく方が良い．例えば上の例で変数 x，y，z の宣言を

```
real(8) :: x, y, z
```

としてしまうと，複数のサブルーチンや関数などから use で参照される場合に，その都度これらの値が書き換えられてしまう (初期化される) 可能性がある．(例えばメインプログラムから 1 度だけ use で参照される場合にはこのような問題は生じない．) 最近のコンパイラは自動でモジュール内変数に save 属性が指定されたものと扱う場合が多いようなので，これは必ずしも必須ではないかもしれない．ただしコンパイラ依存性を無くし，移植性の高いプログラムとするためには指定しておいた方が無難であろう．なお，いずれにせよ定数変数については参照されるだけなので save 属性は必要ない．

一般に，プログラムが複雑化して来ると **グローバル変数がバグの原因** になりやすくなるため，使わない方が良いとされている．グローバル変数を一切使わない場合にはメインプログラムで全ての変数を宣言し，必要な変数を各サブルーチンや関数へ全て引数として渡せば良い．この場合にはデータの受け渡しが明示的に行われるので，意図せずデータが変更されるのを防ぐことが出来る．ただしこれはあくまで一般論であり，いつでもグローバル変数の使用を避けるべきというわけではない．比較的単純で，データの受け渡しを間違いそうに無いようなプログラム (比較的小規模の数値シミュレーションコードはこれに当てはまる場合が多い) であればグローバル変数を用いた方がスッキリ書けるような場合も多い．ただし，この場合でもグローバル変数にしても問題無い変数と，ローカル変数にすべき変数はよく考えて区別しておいた方が良い．何でもかんでもグローバル変数にしてしまうと汎用性の無いプログラムになってしまい，仕様変更に伴うプログラムの修正が非常に困難になる．

例えば以下の例を考えよう．ここでは変数 i をモジュール mod_global 内で定義されたグローバル変数として用いている．メインプログラムの内部手続きとして定義されたサブルーチン sub 内の do ループでも，メインプログラムでも変数 i をループ変数として用いている．gfortran でこのプログラムをコンパイルして実行すると，無限ループになってしまうようである (この動作はコンパイラに依存するかもしれないが，いずれにせよ「意図した通り」には動かない)．これはメインプログラムから sub を 3 回呼び出すつもりでも，sub 内部で変数 i の値が更新され，メインプログラムの do ループの反復が正しく終了しないためであろう．これは極端な例ではあるが，特にループ変数のように不用意に使ってしまいそうな名前の変数はローカル変数にして，必要な場合にその都度宣言して用いる方が安全である．

```
module mod_global
  implicit none

  ! グローバル変数
  integer, save :: i

end module mod_global

program main
  use mod_global
  implicit none

  ! グローバル変数 i でループを回す
  do i = 1, 2
     call sub() ! この中で i が更新されてしまう!!
  end do

  stop
```

(次のページに続く)

(前のページからの続き)

```
contains
  subroutine sub()
    implicit none

    ! ここでもグローバル変数 i でループを回す
    do i = 1, 3
       write(*,*) i
    end do

  end subroutine sub
end program main
```

9.3 内部手続き

サンプルコード参照

メインプログラムと同様に，モジュールにも内部手続きを定義することが可能であり，またモジュールの内部手続きからモジュール内で定義された変数には自由にアクセスすることが出来る (これもメインプログラムの内部手続きと同様である)．use でモジュールの使用を宣言すると，モジュールの変数だけでなく内部手続きも同様に用いることが出来る．このようにモジュールは関連する変数と手続きをまとめることが出来るため，サブルーチンや関数よりも大きなプログラム単位を提供することになる．

例えば以下のモジュール mod_integrator は予めサンプリングされた関数値の配列と刻み幅を受け取り，**数値積分** で扱った台形公式および Simpson の公式を用いて数値積分する関数を内部手続きとして実装したモジュールである．このように関連するサブプログラムをまとめてモジュールの内部手続きとして実装しておけば，use mod_integrator するだけで (外部手続きの時のように interface による準備をしなくても) 安全に利用することが出来る．これから分かるように，**内部手続き** で外部手続きを非推奨としたのは，単純に外部手続きを何らかのモジュールの中に入れて (内部手続きとして定義して) しまえば良いからである．これによって外部手続きの抱える問題は全て解決する．

```
module mod_integrator
  implicit none

contains
  !
  ! 台形公式による数値積分
  !
  function trapezoid(f, dx) result(ret)
    implicit none
    real(8), intent(in) :: f(:)
    real(8), intent(in) :: dx
    real(8) :: ret

    integer :: i, n
```

(次のページに続く)

(前のページからの続き)

```
    n = size(f)

    ret = 0.5_8 * (f(1) + f(n))
    do i = 2, n-1
       ret = ret + f(i)
    end do
    ret = ret * dx

  end function trapezoid

  !
  ! Simpson 公式による数値積分
  !
  function simpson(f, dx) result(ret)
    implicit none
    real(8), intent(in) :: f(:)
    real(8), intent(in) :: dx
    real(8) :: ret

    integer :: i, n

    n = size(f)

    ! 端点を含めた配列サイズが奇数でなければエラー
    if( mod(n, 2) == 0 ) then
       write(*,*) 'array size must be odd'
       stop
    end if

    ret = f(1) + f(n)
    ! even
    do i = 2, n-1, 2
       ret = ret + 4.0_8 * f(i)
    end do
    ! odd
    do i = 3, n-2, 2
       ret = ret + 2.0_8 * f(i)
    end do
    ret = ret * dx / 3.0_8

  end function simpson
end module mod_integrator
```

9.4 総称名

サンプルコード参照

これまでは何も意識せずに単精度で宣言された x に対しても，倍精度で宣言された x に対しても sin(x) のように型の精度を気にせず組み込み関数を呼び出しをしてきたことと思う．しかし自分で定義した関数やサブルーチンについては，正確に宣言した型を引数として呼び出す必要があった．実はその昔の Fortran 77 の時代には単精度に対して sin(x)，倍精度に対しては dsin(x) というように，組み込み関数にも精度ごとに異なる関数が用意されていて，手動で使い分ける必要があった (ここで d は倍精度を表す double の意味である)．これでは明らかに不便である．

Fortran 90 以降では，この問題を解決するために，内部手続きに対して総称名 (オーバーロード) という便利な機能を用いることが出来るようになった[1]．これを用いると，呼び出し形式 (引数の数や型) が異なる複数の関数やサブルーチンを同じ名前で呼び出すことが出来る．先ほどの sin(x) の例で言えば，引数 x が単精度実数であれば単精度版を，倍精度であれば倍精度版の関数を自動的に選択して呼び出すことになる．自分で定義した関数やサブルーチンについても，この総称名の機能を用いることが出来る．

これにはモジュールの変数宣言部分で

```
interface generic_procedure
  module procedure specific_procedure1, specific_procedure2
end interface generic_procedure
```

のように interface を用いて総称名を宣言すれば良い．個別名としては呼び出し形式の異なる (形式が同じだとコンパイラが判別出来ない!) 複数の関数やサブルーチンをカンマで区切って記述する．これによって複数のルーチンを単一の名前で呼び出すことが出来る．(上の例の場合は総称名 generic_procedure によって specific_procedure1 や specific_procedure2 を呼び出す．) コンパイラは総称名で呼び出されたルーチンについて，その呼び出し形式に応じて自動的に適切なものを選択する．具体的な使い方は以下の例を見て欲しい．

```
! 面積を計算するモジュール
module mod_area
  implicit none

  real(8), parameter :: pi = 4*atan(1.0_8)

  ! 総称名を定義
  interface triangle
     module procedure triangle1, triangle2
  end interface triangle

contains

  ! 底辺と高さが与えられた時の面積の計算
  function triangle1(a, b) result(area)
    real(8), intent(in) :: a, b
```

(次のページに続く)

[1] 実は総称名はモジュールに限らず，メインプログラムの内部手続きでも用いることが出来る．その場合あまりありがたみは感じないかもしれないが．

(前のページからの続き)

```
    real(8) :: area

    area = a * b / 2

  end function triangle1

  ! 3 つの頂点の座標が与えられた時の面積の計算
  function triangle2(x1, y1, x2, y2, x3, y3) result(area)
    real(8), intent(in) :: x1, y1, x2, y2, x3, y3
    real(8) :: area

    area = abs((x2-x1)*(y3-y1) - (x3-x1)*(y2-y1))/2

  end function triangle2

end module mod_area
```

この例では三角形の面積を底辺と高さが与えられた時と 3 つの頂点の座標が与えられた時のいずれも同じ関数名で呼び出すことが出来るように総称名 `triangle` を宣言している．2 つの違いは呼び出し時の引数だけなので，呼び出される時の引数の個数や型によってコンパイラが自動的に適切な方を呼び出すことが出来る．なお，総称名を用いると全く別の機能を実装したものであってもまとめることが出来てしまうのだが，このような使い方は混乱の元になるだけであろう．総称名を使うのは意味的に同じ機能を持った関数やサブルーチンをまとめる時にのみにしておいた方が良い．

9.5 アクセス制限

サンプルコード参照

モジュールを用いるために `use` すると，モジュール内で定義された変数，関数，サブルーチンに自由にアクセスできるが，これは一般的にはあまり好ましいことでは無い．例えば，**変数や定数の参照** では不用意にグローバル変数を作るとバグの元になることを示した．これはモジュールの利用者がモジュール内部の詳細を知らないために起こる問題である．

しかし，そもそもモジュールを利用する側はモジュール内部の詳細について知らないことが一般的であるし，そうあってしかるべきである．すなわち，モジュールを提供する側はそのモジュールと外部のインターフェースのみを提供し，内部の実装の詳細については公開しないという立場を取る方が懸命である．これには主に以下の 2 つの理由が挙げられる．

- モジュールを利用する立場からは，モジュール内で定義された変数名などで名前空間が汚染されてしまい，同じ名前の変数やルーチンを宣言出来なくなる．
- モジュールを提供する立場からは，モジュール内部で用いている変数などが不用意に変更されてしまう可能性がある．例えば何らかの状態を保持する変数が利用者から意図せず変更されてしまうと動作がおかしくなるかもしれない．

関数とサブルーチン で学んだことは，それらをブラックボックスとして用いることで間違いを減らすことが出来るということであった．モジュールについても基本的に考え方は同じであって，せっかく機能を分割してモジュールを実装したのならば利用する時にはその中身のことは忘れたい．特に規模が大きなプロ

グラムを複数人体制で開発する際には他人が実装したモジュールの中身など知る由も無いし，知りたくも無いであろう．いたずらに守備範囲を広げてエラーするくらいなら，狭くても良いから自分の守備範囲だけは死守する方が守りは固くなるのである．

9.5.1 参照先からのアクセス制限

まずは参照先 (モジュールの利用者の側) からのアクセス制限について学ぼう．一部の変数やルーチンへのアクセスしか必要無い場合には，use 宣言の際に only を用いてそれ以外の名前を参照先からは無効にする (見えないようにする) ことが出来る．例えば **変数や定数の参照** の mod_variable から light_speed だけを用いたい場合には

```
use mod_variable, only : light_speed
```

のように only : に続けて必要な変数名やルーチン名をカンマで区切ってリストすれば良い．また light_speed を別名で使いたい場合には

```
use mod_variable, only : c => light_speed
```

のようにすることで，light_speed の代わりに c という名前でアクセスすることが出来る．(ただしあくまで別名なので実態は ``light_speed`` のままである．) これによって，例えばモジュール内部の変数と同じ名前の変数を参照先で使いたい場合に，名前の競合を避けることが出来る．

9.5.2 参照元からのアクセス制限

参照先からのアクセス制限は言わば性善説の立場に立ったアクセス制限である[2]．それに対して，性悪説の立場に立った，モジュールを提供する側からのアクセス制限も可能である．

モジュール内部で宣言された変数やルーチンには public や private などの属性を与えることが出来，この属性によってアクセス制限をすることが出来る．すなわち，private 属性が指定された変数やルーチンはモジュール外からは直接見えず，内部からのみアクセスが可能になる．一方で，public 属性が指定されたものは公開され，外部から自由にアクセスすることが出来る．Fortran のモジュールでは public がデフォルトである．

原則としてモジュールの内部でしか用いられないものは外部には公開しない方が良い．例えば以下のモジュールを考えよう．

```
module mod_sample
  implicit none

  integer :: l, m, n

end module mod_sample
```

[2] モジュール利用者がモジュール内部の処理に悪影響を与えないように振る舞ってくれればこれでも良いのだが，過度の期待は禁物である．

いかにも他で使いそうな l，m，n という変数をモジュール内で宣言している．例えばメインプログラムからこのモジュールを利用する際に，他の用途に使おうと思ってこれらと同じ名前の変数を宣言するとコンパイルエラーとなってしまう．実はコンパイルエラーを出してくれればまだ良い方なのであって，メインプログラムで変数宣言を忘れた場合にはこれらの変数が普通に使えてしまう．モジュールの変数だと意識して使っていれば問題は無いのだが，そんなことはお構いなしに全く違う用途に使って値を書き換えてしまうと，モジュール内部でこれらの変数に依存しているようなコードは動作がおかしくなってしまうかもしれない．公開する必要が無いものは予め非公開にしておけば，このような不用意なバグの混入を未然に防ぐことが出来るのである．public と private の指定方法はいくつかあって，個別に指定する場合は

```
integer, private :: l, m, n
```

のように変数宣言時に属性を指定することが出来る．または

```
integer :: l, m, n

private :: l, m, n
```

のように別に指定することも可能である．なお，内部手続きの公開設定についても上の 3 行目のような形で手続名を並べれば良い．デフォルトで非公開としたい場合にはモジュール宣言の最初 (implicit none の後) に private を指定しておけば，明示的に public 属性を付けない限りは非公開となる．実用的なプログラムではデフォルトを非公開の設定にし，必要な物だけに public 属性を指定することを強く推奨する．

以下は単位変換付きの物理定数モジュールの例である．デフォルトで private にすることでモジュール内部の変数には直接アクセス出来ないようし，その代わり必要な定数の値を返す関数を public にしてある．アクセスする手段 (インターフェース) を敢えて限定することで，単位系のモード (unit) に応じて物理定数の値が自動的に切り替わるようになっている[3]．

```
! 物理定数モジュール
module mod_const
  implicit none
  private ! デフォルトで非公開

  ! 単位選択フラグ: 1 => MKS, 2 => CGS
  integer, save :: unit = 1

  real(8), parameter :: pi  = 4*atan(1.0_8)
  real(8), parameter :: mu0 = 4*pi * 1.0e-7_8

  ! MKS => CGS への変換ファクター
  real(8), parameter :: T = 1.0e+0_8
  real(8), parameter :: L = 1.0e+2_8
  real(8), parameter :: M = 1.0e+3_8

  ! MKS で定義
  real(8), parameter :: mks_light_speed       = 2.997924e+8_8
```

(次のページに続く)

[3] このように内部データを敢えて隠ぺいする (外から見えないようにする) ことをカプセル化 (encapsulation) と呼ぶ．これは現代的なオブジェクト指向プログラミングの基本的な概念である．

(前のページからの続き)

```
  real(8), parameter :: mks_electron_mass     = 9.109382e-31_8
  real(8), parameter :: mks_elementary_charge = 1.602176e-19_8

  ! これらのみ公開
  public :: set_mks, set_cgs
  public :: light_speed, electron_mass, elementary_charge

contains

  ! MKS モード
  subroutine set_mks()
    implicit none

    unit = 1
  end subroutine set_mks

  ! CGS モード
  subroutine set_cgs()
    implicit none

    unit = 2
  end subroutine set_cgs

  ! 光速
  function light_speed() result(x)
    implicit none
    real(8) :: x

    if( unit == 1 ) then
       x = mks_light_speed
    else if ( unit == 2 ) then
       x = mks_light_speed * 1/T
    else
       call unit_error(unit)
    end if

  end function light_speed

  ! 電子質量
  function electron_mass() result(x)
    implicit none
    real(8) :: x

    if( unit == 1 ) then
       x = mks_electron_mass
```

(次のページに続く)

(前のページからの続き)

```
    else if ( unit == 2 ) then
       x = mks_electron_mass * M
    else
       call unit_error(unit)
    end if

  end function electron_mass

  ! 素電荷
  function elementary_charge() result(x)
    implicit none
    real(8) :: x

    if( unit == 1 ) then
       x = mks_elementary_charge
    else if ( unit == 2 ) then
       x = mks_elementary_charge * light_speed() * sqrt(mu0/ 4*pi) * M * L * T**2)
    else
       call unit_error(unit)
    end if

  end function elementary_charge

  ! エラー
  subroutine unit_error(u)
    implicit none
    integer, intent(in) :: u

    ! 標準エラー出力へ
    write(0,'(a, i3)') 'Error: invalid unit ', u

  end subroutine unit_error

end module mod_const
```

9.6 構造型

サンプルコード参照

これまで扱ってきた `integer` や `real` のような組込み型だけで無く，新しいデータ型を自分で定義して用いることもできる．これを **構造型** と呼ぶ[4]．構造型は組込み型やその配列はもちろん他の構造型を要素に持つことも出来る．

[4] Fortran 2003 以降では派生型と呼ばれるらしい．(オブジェクト指向プログラミングをサポートしたからであろう．)

9.6.1 定義と使い方

構造型は以下の様な形式で定義される．

```
type :: name_of_type
  !型名 :: 要素名
  !の形で任意の変数を定義する
end type name_of_type
```

組み込み型と同様に，定義した構造型の変数を以下のように宣言することによって使うことが出来る．

```
type(name_of_type) :: name_of_variables
```

例えば以下の例では，倍精度実数型の x，y を要素に持つ新しい構造型 vector2 を定義して用いている．

```
! 2次元のベクトル
type :: vector2
  real(8) :: x, y
end type vector2

! 構造型の変数を宣言
type(vector2) :: a

! '%' を用いて各要素にアクセスが出来る
a%x = 1.0_8
a%y = 0.0_8
```

構造型の各要素にアクセスするには上の例のように"% "を用いれば良い．構造型を用いると，常にセットで必要になるような複数のデータをまとめて保持することが出来るので，上手く利用すればプログラムが非常に見やすくなる．例えば非常に多くの引数を必要とするサブルーチンでも，構造型を利用してデータをまとめることで引数の数を減らして，スッキリとした形に書き換えることが出来るだろう[5]．

9.6.2 ユーザー定義演算子

構造型の機能として特筆すべきは，構造型に対する演算子を自分で定義することが出来るという点である．これをユーザー定義演算子と呼ぶ．演算子の定義は以下のように総称名の場合とほぼ同様である．

```
interface operator (.operator.)
   module procedure specific_procedure1, specific_procedure2
end interface operator (.operator.)
```

演算子記号は +，-，*，/ という組込み型に対して定義されている演算子か，または .operator. のように両側をピリオドで挟んだ名前のいずれかである．例として，ベクトル同士の和を各要素同士の和として + 演算子を定義しよう．以下の関数は 2 つのベクトルを引数に受け取り，要素同士の和を計算したものを結果として返す．

[5] もっともこれは好みの問題であって，引数が山ほどあるサブルーチンの方が分かりやすいという人もいるかもしれない．

```
! + 演算子の中身
function add2(a, b) result(ret)
  implicit none
  type(vector2), intent(in) :: a, b
  type(vector2) :: ret

  ret%x = a%x + b%x
  ret%y = a%y + b%y
end function add2
```

これを interface を用いて

```
interface operator (+)
   module procedure add2
end interface operator (+)
```

のように宣言しておくことで type(vector2) の変数 a，b の和を a + b のように記述することが出来る．この + 演算子の計算の実態は上の add2 という関数である．

また演算子についても総称名を用いることが出来る．すなわち add2 は引数が 2 つとも type(vector2) であったが，例えばどちらか片方が実数型の場合の処理も定義して interface 宣言に加えておくことで，+ 演算子を総称名として用いることが出来る．これを演算子のオーバーロードと呼ぶ．

例えば，先ほどの add2 に加えて add2_scalar1，add2_scalar2 の 2 つの関数を interface に加えておく．

```
interface operator (+)
   module procedure add2, add2_scalar1, add2_scalar2
end interface operator (+)
```

ここで add2_scalar1，add2_scalar2 はそれぞれ以下のように定義されたものとする．

```
! + 演算子の中身: vector2 + scalar
function add2_scalar1(a, b) result(ret)
  implicit none
  type(vector2), intent(in) :: a
  real(8), intent(in) :: b
  type(vector2) :: ret

  ret%x = a%x + b
  ret%y = a%y + b
end function add2_scalar1

! + 演算子の中身: scalar + vector2
function add2_scalar2(a, b) result(ret)
  implicit none
  real(8), intent(in) :: a
  type(vector2), intent(in) :: b
```

(次のページに続く)

(前のページからの続き)

```
  type(vector2) :: ret

  ret%x = a + b%x
  ret%y = a + b%y
end function add2_scalar2
```

このようにしておけば，vector2 型と倍精度実数型の + 演算が可能になる．この時，a + 1.0_8 のような演算では add2_scalar1 が，0.5_8 + a のような演算では add2_scalar2 が自動的に呼び出されることになる．

9.6.3 ユーザー定義代入文

代入文 (=) に関しては，ユーザー定義演算子とは少し事情が異なるのでここで触れておく．まず代入文は，同じ構造型同士であればデフォルトで使用可能である (例えば type(vector2) 型の変数同士で a = b としても良い)．この場合は，構造型の各要素で単純に代入文が実行される．異なる型同士での代入には，ユーザー定義代入文の定義が必要である．(明示的に指定しない限り，コンパイラには何が正しい代入動作か判断出来ないため．)

ユーザー定義演算子と異なり，代入文の実態はサブルーチンを用いて定義する．

```
! = 中身
subroutine assign2(a, b)
  implicit none
  type(vector2), intent(out) :: a ! intent(out) に注意
  real(8), intent(in)        :: b ! intent(in)  に注意

  ! どちらも同じ値
  a%x = b
  a%y = b
end subroutine assign2
```

このサブルーチンを代入文として用いるには以下の様な interface 宣言を行う．

```
interface assignment (=)
   module procedure assign2
end interface assignment (=)
```

これによって a = 0.0_8 のように，= の右辺と左辺が異なる型の変数の場合であっても代入を行うことが出来るようになる．これにも総称名を用いてオーバーロードすることが可能である．

9.7 第 9 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例
- 課題 5 解答例

9.7.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

9.7.2 課題 2

サンプルプログラム sample1.f90 を参考にして，数学定数を定義するモジュールを作成せよ．メインプログラムも作成して，動作を確認すること．特に use によるモジュール参照， only の使い方などを理解し，定数値の書き換えが出来ない (コンパイルエラーとなる) ことを確かめよ．数学定数としては例えば π, $\sqrt{\pi}$, e などを定義すれば良い．

9.7.3 課題 3

sample5.f90 のモジュール mod_vector で定義されている 2 次元のベクトル構造型 type(vector2) を拡張し，以下の演算子を定義せよ．(以下で a, b は共に type(vector2) で宣言されたものとする．)

- - 演算子: 倍精度実数と type(vector2) の間で以下のような演算が出来るようにする．

```
b = a - 1.0_8 ! b%x = a%x - 1.0_8; b%y = a%y - 1.0_8; と同値になるように
b = 1.0_8 - a ! b%x = 1.0_8 - a%x; b%y = 1.0_8 - a%y; と同値になるように
```

- * 演算子: ベクトル積を返す演算子とする．(従って以下の 2 行が同じ結果となる．)

```
write(*,*) a * b
write(*,*) a%x * b%y - a%y * b%x
```

- = 演算子: 以下のような代入文を実行できるようにする．

```
a = 1.0_8             ! x, y に同じ値 (1.0) を代入
b = (/2.0_8, 1.0_8/) ! x には 2.0, y には 1.0 を代入
```

ここで b = (/2.0_8, 1.0_8/) の右辺は長さ 2 の倍精度実数型の配列である．

9.7.4 課題 4

mod_vector をさらに拡張し， x, y, z を要素に持つ 3 次元ベクトルを表す構造型 type(vector3) を新たに定義せよ．また，この型に対する +, -, *, = などの演算子を定義せよ．(ただし, * 演算子は通常のベクトル積を返すものとし，それ以外は type(vector2) の自然な拡張となるようにすること．)

さらにサブルーチン show をオーバーロードし， type(vector2), type(vector3) のどちらの変数も引数に取れるようにせよ．

これによって，例えば

```
type(vector2) :: a, b, c
type(vector3) :: x, y, z

write(*, fmt='(a)') '--- vector2 ---'

a = (/1.0_8, 0.0_8/)
b = 1.0_8
c = a + b

call show(a + b)
call show(a - b)

write(*, fmt='("a * b = ", f12.4)') a*b

write(*, fmt='(a)') '--- vector3 ---'

x = (/1.0_8, 0.0_8, 0.0_8/)
y = 1.0_8
z = x * y

call show(x)
call show(y)
call show(z)
call show(x+y)
call show(x-y)
```

のような記述が可能になるはずである．

9.7.5 課題 5

有理数を表す構造体を扱うモジュールを作成せよ．有理数同士の和差積商の演算子もそれぞれ定義すること．さらに，代入演算子で長さ 2 の整数配列を受け取れるようにし，また有理数を標準出力に表示するサブルーチン (例えば show) も作成せよ．(有理数の分子および分母は整数であるので 2 つの整数型を持つ構造体として定義すればよい．)

```
type(rational) :: a, b

a = (/1, 4/)
b = (/2, 5/)

write(*, fmt='(a)', advance='no') 'a     = '
call show(a)

write(*, fmt='(a)', advance='no') 'b     = '
call show(b)

write(*, fmt='(a)', advance='no') 'a + b = '
call show(a+b)

write(*, fmt='(a)', advance='no') 'a - b = '
call show(a-b)

write(*, fmt='(a)', advance='no') 'a * b = '
call show(a*b)

write(*, fmt='(a)', advance='no') 'a / b = '
call show(a/b)
```

例えば上のようなプログラムをコンパイルして実行した結果が以下のようになればよい．約分も忘れずにすること．最大公約数を求める関数もしくはサブルーチンを用いると良い．(絶対値に注意すること．)

```
a     =      1        4
b     =      2        5
a + b =     13       20
a - b =              20
a * b =      1       10
a / b =      5        8
```

第10章　付録

参考:

- sample1.f90 : 分割コンパイル用サンプル
- sample1a.f90 : 分割コンパイル用サンプル (モジュール A)
- sample1b.f90 : 分割コンパイル用サンプル (モジュール B)
- sample2.f90 : GUI ライブラリの使い方
- sample3.f90 : 二分法モジュール (mod_bisection) の使い方
- bisection.f90 : 二分法モジュール (mod_bisection) の実装
- sample4.f90 : 計算時間の測定
- sample5.f90 : ポインタの使い方
- sample6.f90 : リストモジュール (mod_list) の使い方
- list.f90 : リストモジュール (mod_list) の実装
- sample7.f90 : プリプロセッサの使い方

この章の内容

10.1 大規模なプログラム開発

10.1.1 分割コンパイル

これまでは基本的に単一のソースファイルからなるプログラムを扱ってきたが，大規模なプログラムを扱う際にはソースファイルをいくつかに分割して扱うと開発の見通しが良くなる．複数のソースファイルからなるプログラムを開発するにあたって，まずは単一のソースファイルから実行形式のファイルを作る手順を復習しよう．

これまでは例えば

によってソースファイルから実行ファイル a.out を作成してきた．これは実際には **コンパイル** と **リンク** という 2 つのステップを同時に行うものである．

コンパイルとはソースファイルを解釈し，機械語に変換する作業である．機械語に変換されたファイルを **オブジェクトファイル** と呼び，通常は拡張子 .o が用いられる．オブジェクトファイルを生成するには

のように -c オプションを付けてコンパイルする．この結果，この例では sample.o が生成される．生成されたオブジェクトファイルは機械語になってはいても，実際にプログラムを実行可能な形式にはなっておらず，関数やサブルーチンなどの単位で個別に機械語になっているだけである．実行形式に変換するには，それらの自分で実装したものに加え，組み込み関数 (read , write なども実際には組み込み関数の 1 種である) などを連結する作業が必要である．組み込み関数などのまとまりを標準ライブラリと呼び，この連結作業をリンクと呼ぶ．

オブジェクトファイルをリンクするには

とすればよい．この例では sample.o と標準ライブラリがリンクされ，実行可能なファイル (この例では a.out) が生成される．

次に複数のファイルからなるプログラムの場合を考えよう．例えばメインプログラムが定義されている sample1.f90 が sample1a.f90 に，また sample1a.f90 が sample1b.f90 に依存している場合には

のようにすれば，コンパイルとリンクを 1 つのコマンドで同時に実行することができる (順番に注意)．これは実際には以下のように各ファイルを個別にコンパイルし，最後に全体をリンクする作業に相当する．

リンク時には与えられた全てのオブジェクトファイルに加えて，常に標準ライブラリがリンクされる．

このように分割コンパイルを用いる利点は，コンパイル時間の短縮にある．個別にコンパイルが出来るので，ソースファイルを更新した場合には，その必要なファイルだけを再コンパイルすれば良い．なお，リンクにかかる時間は通常短く，コンパイル時間に対しては無視できる．従って，頻繁に修正をしながらテストする場合などでは，実行ファイルの作成にかかる時間を短縮できることになる．

なお，Fortran でモジュールを用いる場合は gfortran -c によるコンパイルで .mod という拡張子のファイル (例えばモジュール名が test なら test.mod) が生成される．別ファイルのモジュールを利用する際には，この .mod ファイルが無いとコンパイル出来ないので注意しよう[1]．また他のディレクトリに .mod

[1] これは Fortran の規格というわけではないようだが，多くのコンパイラがこのような仕様になっている．

ファイルがある場合には，-I オプションを用いてコンパイラが .mod ファイルを探す場所を指定することが出来る[2]．

```
```

のように実行すれば，カレントディレクトリに加えて ../module にある .mod ファイルも参照することになる．規模が大きなプログラムではいくつかのディレクトリにファイルを分散して配置するようになるため，-I オプションを頻繁に用いることになるだろう．

10.1.2 ライブラリの利用方法

一般に，関数やサブルーチン群をひとまとまりにしたものをライブラリと呼ぶ．これまでにも組み込み関数などの標準で用意されている便利な機能を利用してきたが，これは標準ライブラリが提供するものである．これと同じように，他の誰かが開発した便利な関数やサブルーチンを自分のプログラムから呼び出して利用することも出来る．

ライブラリは通常 (Unix 系の OS では) libABC.a や libXYZ.so といったファイル名になっている (これらをアーカイブとも呼ぶ)．基本的には実行形式のファイルを作成するリンク時にこれらのライブラリともリンクするように指定してやれば良い．例えば

```
```

とすれば libABC.a，libXYZ.so の両方とリンクすることが出来る．このようにライブラリとリンクするには，拡張子とファイル先頭の lib を除いたライブラリ名を -l オプションに渡せば良い．ただし，ライブラリファイルがあるディレクトリがカレントディレクトリや標準の場所 (通常 /usr/lib や /usr/local/lib など) 以外の場合にはその場所を -L オプションで明示的に指定してやらなければならない．以下の例では libABC.a が ../lib にある場合に，それを明示的に指定してリンクを実行する．

```
```

10.1.3 GUI プログラムの作成 †

10.1.4 Make の使い方

To be written.

[2] C 言語のインクルードパスの指定と同じである．

10.2 関数やサブルーチンの引数渡し

関数やサブルーチンの引数として，関数やサブルーチンを渡すことが出来る．これには以下のように引数として渡す関数やサブルーチンの形式を interface を用いて指定する．

```
subroutine writefunc(f, x)
  implicit none
  real(8), intent(in) :: x

  ! 引数として受け取る関数の形式
  interface
     function f(x) result(y)
       real(8), intent(in) :: x
       real(8) :: y
     end function f
  end interface

  write(*,'("f(", e12.5, ") = ",  e12.5)') x, f(x)

end subroutine writefunc
```

より実用的な例として二分法のアルゴリズムを実装した以下の様なモジュール(抜粋)を考えよう．bisection というサブルーチンは関数 $f(x)$ を引数として受け取り，$f(x) = 0$ の解を求める．このようなモジュールを定義しておけば，$f(x)$ の具体的な形が変わってもメインプログラムで bisection の引数として渡す関数を変更するだけで良い．このように，汎用性の高いモジュールを作成しておくことで再利用が非常に簡単になる．

なお，このサブルーチンでは最大の反復回数や許容誤差は optional 属性の引数となっていることにも着目して欲しい．これらが与えられない場合にはモジュール内で定義されたデフォルトの値を用いるようになっている．

```
module mod_bisection
  implicit none
  private

  integer, parameter :: default_maxit = 50
  real(8), parameter :: default_tol   = 1.0e-8_8

  public :: bisection

contains
  ! 二分法により与えられた方程式の解を求める
  subroutine bisection(f, x1, x2, error, status, maxit, tol)
    implicit none
    real(8), intent(inout) :: x1, x2
    real(8), intent(out) :: error
    integer, intent(out) :: status
```

(次のページに続く)

(前のページからの続き)

```
    integer, intent(in), optional :: maxit
    real(8), intent(in), optional :: tol
    ! 引数として関数を受け取る
    interface
       function f(x) result(y)
         real(8), intent(in) :: x
         real(8) :: y
       end function f
    end interface

    integer :: i, n
    real(8) :: x, y, sig, tolerance

    ! 最大の反復回数
    if (.not. present(maxit)) then
       n = default_maxit
    else
       n = maxit
    end if

    ! 許容誤差
    if (.not. present(tol)) then
       tolerance = default_tol
    else
       tolerance = tol
    end if

    !
    ! 以下略
    !

  end subroutine bisection

end module mod_bisection
```

なお bisection.f90 で上記モジュールが定義されており，sample3.f90 がこれを用いるメインプログラムである．これをコンパイルするには

```

```

もしくは

```

```

とすれば良い．

10.3 計算時間の測定

プログラム全体の実行時間はシェルコマンドで計測できる．例えば time コマンドを用いて以下のように実行すれば良い[3]．real の行が実際の実行時間を示している．大雑把には全実行時間のうち，user が自分のプログラムの処理が動いていた時間，sys は OS の処理が動いていた時間を表す．

```
        0                    0                    0
```

アルゴリズムによる実効速度の違いを検証したり，プログラムのチューニングをするようになってくると，プログラムのある特定の部分の実行時間を測定する必要が出てくる．ここでは cpu_time という組込みサブルーチンを使った時間計測のサンプルプログラムを以下に示す．

実行結果は以下のようになる．

```
          [    ]      0
```

サンプルプログラムでは 9-12 行目の処理にかかる時間を計測している．cpu_time の呼び出しによって引数に現在の時刻が秒単位で代入されるので，計測したい部分の前後でこの値の差をとればよい．ただし正確に計算時間を計測する際には以下のようにいくつか注意が必要である．

- write や read のような I/O(入出力) は避ける．これらは非常に時間のかかる処理であるので，入出力があると純粋な計算時間を正確に測ることが出来ない．
- 計測対象がある程度時間のかかる処理 (例えば $\gtrsim 1$ 秒) であること．cpu_time の返す時刻の精度は $1\mu s$ 程度である．
- 何度か同じ計測をしてばらつきを見ること．現在のほぼ全ての計算機はマルチタスク OS であり，多くの処理を同時に行なっているため，ユーザーが実行しているプログラムの処理に全てのリソースが使われるわけではない．たまたま他のプログラムが走っているタイミングで実行された場合にはパフォーマンスがでない可能性がある．

なお，細かいことを言うと経過時間 (elapsed time)，CPU 時間 (cpu time)，システム CPU 時間 (system cpu time)，ユーザー CPU 時間 (user cpu time) で意味が少しずつ異なることに注意しよう．cpu_time で計測されるのは CPU 時間である．

[3] 単に time とするとシェルの組込みコマンドになってしまうので，/usr/bin/time と絶対パスを指定している．シェル組込みの time でも問題は無いが，出力が多少異なるであろう．

10.4 ポインタとデータ構造 †

ポインタとは簡単に言えば別名である．即ち，ポインタ変数は他の変数や配列によって保持されているデータ領域のメモリ上のアドレス (番地) を指し示す変数になっている．例えば，ポインタ変数に対する処理として記述しておけば，ポインタ変数の指し示すアドレスを変更するだけで異なるデータに対する処理を行っていることになる．

しかし，ポインタはこれまで出てきたような処理には敢えて用いる必要が無いので，あまりありがたみを感じられないかもしれない．実際にはポインタはリストやスタック，キューなどのより複雑な **データ構造** を記述するには必須であるが，そうでなければ必ずしも必要にはならない．ポインタは初心者には少し難しいかもしれないので，分からなければとりあえずは無視しても良い．

10.4.1 ポインタ変数

ポインタを用いるには変数宣言に pointer 属性を指定すれば良い．またポインタが指し指すことの出来る変数 (ターゲット変数) には target 属性を指定する必要がある．以下の例を考えよう．

```
integer, pointer :: iptr
integer, target  :: i, j

i = 5
j = 9

! 出力は iptr = 5, i = 5, j = 9
iptr => i
write(*,'(" iptr = ", i3, ", i = ", i3, ", j = ", i3)') iptr, i, j

! 出力は iptr = 9, i = 5, j = 9
iptr => j
write(*,'(" iptr = ", i3, ", i = ", i3, ", j = ", i3)') iptr, i, j

! 出力は iptr = 0, i = 5, j = 0
iptr = 0
write(*,'(" iptr = ", i3, ", i = ", i3, " j, = ", i3)') iptr, i, j
```

8 行目の `iptr => i` によって iptr は i のアドレスを指すことになる (iptr が i の別名となる) ので，9 行目の出力では"iptr = 5, i = 5, j = 9"となる．同様に 12 行目の `iptr => j` によって iptr は j のアドレスを指すことになる．また，16 行目の `iptr = 0` は iptr の指すアドレスへの代入なので，この結果 iptr だけでなく j の値も変更されることになる．この代入文のように，ポインタ変数に対しても通常の変数のように任意の演算が可能である[4]．

なお nullify によってポインタ変数とターゲット変数との結合を解除することが出来る．またポインタは **無名領域** (他の変数によって指し示されていない領域) との結合も可能である．これは動的配列の場合と同様に allocate によって行い，この場合の結合の解除は (nullify では無く) deallocate によって行う．

[4] C 言語のポインタと異なり，=> でポインタのアドレスを指定する以外は普通の変数のように使える．これによってポインタであることを意識せずに用いることが出来る．個人的には逆に分かりにくいように思うのだが．

また，結合状態を検査するための組込み関数 associated も用意されている．この関数はポインタが結合状態であれば真，そうでなければ偽を返す．従って例えば以下のように使うことが出来る．

```
if( associated(iptr) ) then
  nullify(iptr)
end if

allocate(iptr)
iptr = 1

deallocate(iptr)
```

10.4.2 ポインタ配列

ポインタ配列はターゲット配列と結合させることが出来る．ただし両者の次元は同じでなければならない．ポインタ配列は配列全体を指したり，部分配列を指したりすることが出来る．

```
integer :: i, j
integer :: lb(2), ub(2)
integer, pointer :: rptr(:,:)
integer, target :: x(9,9)

do j = 1, 9
   do i = 1, 9
      x(i,j) = i + (j-1)*9
   end do
end do

! 部分配列へ結合
rptr => x(2:4:2,6)

lb = lbound(rptr) ! (/1, 1/)
ub = ubound(rptr) ! (/3, 5/)

do j = lb(2), ub(2)
   do i = lb(1), ub(1)
      write(*, fmt='(i7)', advance='no') rptr(i,j)
   end do
   write(*,*)
end do
```

上の例では 13 行目において rptr を x の部分配列と結合させている．ポインタ変数の場合と同様に，rptr は x への別名となり，このポインタ配列に対するアクセスは x の対応する要素へのアクセスと等しくなる．

またポインタ配列に対しても allocate による無名領域への結合が出来るので，動的配列と同様に用いる

ことも可能である[5].

10.4.3 データ構造

これまでの例では，ポインタは複雑な割にはありがたみが少ない．実際にポインタを使う必要性は特に感じられないだろう．ポインタが特に重要となってくるのは柔軟なデータ構造を扱う場合である．実は配列もデータ構造の一つである．配列はサイズが固定で，全ての要素が同じ型のデータの集まりなので比較的簡単に扱うことが出来る．すなわち，配列の各要素はメモリ上に連続的に配置されているので，任意の要素への直接アクセス (ランダムアクセス) が可能という長所を持つ (任意の要素のアドレスが簡単に計算出来る)．その一方で，要素をある特定の位置に挿入したい場合にはそれ以降の要素を全てずらさなければならないし，サイズの変更が簡単には出来ないという短所がある．

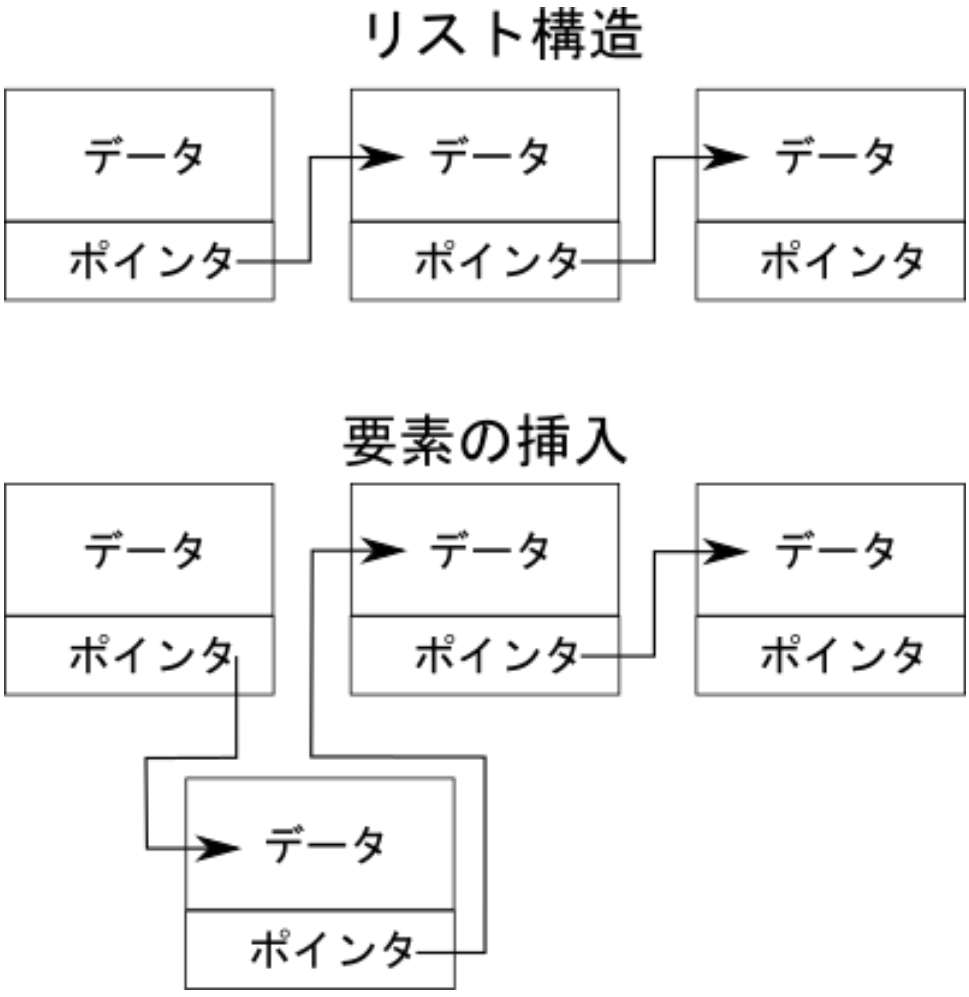

図 1: List 構造.

配列とよく対比されるデータ構造として，リスト (list) が知られている．図に示すようにリスト構造は各要素が他の要素へのポインタ (アドレス) を保持する．従って，リスト構造では任意の要素 (例えば先頭から N 番目の要素) へのアクセスをしようと思ってもそのアドレスが分からない．すなわち，常に各要素に対して順番にアクセス (シーケンシャルアクセス) をしなければならない．これは配列に対するリスト構造の短所である．その一方で，任意の場所への要素の挿入や削除をするにはポインタアドレスの付け替えをする

[5] ただしポインタ配列の場合は `allocatable` とは異なり，スコープから外れた場合に自動で `deallocate` されないようである．

だけで実現出来る．また，リストの末尾に要素を追加するには新しい要素を生成 (allocate) して，その要素に対するポインタを指定してやれば良い．このような特徴があるため，シーケンシャルアクセスしか必要とされない代わりに，頻繁にサイズ変更，要素の挿入・削除があるような用途にはリストが有用となる．

例として，各要素の値が整数であるリストは以下のように構造型を用いて定義することになる．

```
type :: list_type
   type(list_type), pointer :: next
   integer :: value
end type list_type
```

リストの実装 (list.f90) は少し長くなるのでここには掲載しないが，地道にポインタを使ってアドレスを付け替えたりするだけである．以下の例は list.f90 で実装したモジュール mod_list の使い方を示している．リストの伸長は append，要素の挿入は insert，削除は remove を用いてなどのモジュール内部手続きによって行うことが出来る．なお 10 行目ではモジュール内で定義した代入演算子を用いてリストを配列によって初期化している．

```
program sample
  use mod_list
  implicit none

  type(list_type), pointer :: a

  nullify(a)

  write(*, fmt='(a20)', advance='no') 'initialize : '
  a = 1
  call show(a)

  write(*, fmt='(a20)', advance='no') 'append 2 : '
  call append(a, 2)
  call show(a)

  write(*, fmt='(a20)', advance='no') 'append 3 : '
  call append(a, 3)
  call show(a)

  write(*, fmt='(a20)', advance='no') 'insert -1 at 1 : '
  call insert(a, 1, -1)
  call show(a)

  write(*, fmt='(a20)', advance='no') 'insert -2 at 3 : '
  call insert(a, 3, -2)
  call show(a)

  write(*, fmt='(a20)', advance='no') 'remove at 1 : '
  call remove(a, 1)
```

(次のページに続く)

(前のページからの続き)

```
  call show(a)

  write(*, fmt='(a20)', advance='no') 'remove at 3 : '
  call remove(a, 3)
  call show(a)

  write(*, fmt='(a20)', advance='no') 'delete : '
  call delete(a)
  call show(a)

end program sample
```

このプログラムの実行結果は以下のようになる．正しくリストの操作が出来ていることが分かるかと思う．

```
          List = [    1 ]
2         List = [    1     2 ]
3         List = [    1     2     3 ]
1         List = [          1     2     3 ]
3         List = [          1           2     3 ]
1         List = [    1           2     3 ]
3         List = [    1           3 ]
          List = []
```

ここではリストの実装例を見たが，他にもキュー (queue) やスタック (stack)，ツリー (tree)，ハッシュ (hash) などがよく用いられるデータ構造の例として挙げられる．(これらはあくまでも大まかな分類であり，それぞれに対して更に細かい種類がある．例えば単にリストと言っても，片方向リストや双方向リスト，線形リストや循環リストなどの種類が存在する．)

ただし，大規模な数値計算においてこのようなデータ構造が用いられることはあまり多くない．なぜなら一般に柔軟なデータ構造を用いる処理は単純な配列に比べて効率が悪いためである．大規模数値シミュレーションではパフォーマンスが非常に重要となってくるため，複雑なデータ構造は極力使わずに実装される．しかし，配列では実装しづらい複雑なアルゴリズムを用いる必要がある場合にはデータ構造の知識も重要になってくるかもしれないので，興味のある人は勉強してみて欲しい．なお C++を始めとする多くの言語において，**データ構造を扱う標準ライブラリ** (組込み関数のようなもの) が存在している．従って，複雑なデータ構造の扱いが必要な場合には大人しく Fortran を使うのは諦めて，他の言語を用いることを推奨する．パフォーマンスが必要な場合には C++など，そうでないなら Python などを使うことで開発効率が格段に向上するであろう．

10.5 デバッグのテクニック †

10.5.1 プリプロセッサ

プログラムの開発中 (デバッグ中) には一時的にソースコードの一部分をコメントアウトしてその影響を調べたいことが多々あるかと思う．このような時にプリプロセッサと呼ばれる機能を用いると便利である．プリプロセッサとは C/C++のコンパイラが自動的に行う前処理のことであるが，Fortran コンパイラでも多くの場合オプションによってプリプロセッサを有効にすることが出来る．なお，プリプロセッサを使うには gfortran であれば -cpp オプション[6] を付けてコンパイルすれば良い．または拡張子を .f90 では無く .F90 とすればオプションを指定しなくても自動的にプリプロセッサが有効になる．

プリプロセッサの機能は様々であるが，特に簡単で便利なのは以下の様な使い方である．

```
```

一時的に実行させたく無い処理については，上の例のように #if 0 から #endif によって囲むことでコンパイラにその部分を無視させることが出来る．コンパイラに認識させたい時には 0 を 1 に変更するだけで良いので，特定の処理が結果に与える影響を簡単に調べることが出来る．さらに

```
#if 1
  処理 A
#else
  処理 B
#endif
```

としておけば，1 の時には処理 A，0 の時には処理 B をそれぞれコンパイルさせる事が出来る．

プリプロセッサは基本的に C 言語で用いられるものと同じなので，**マクロ** という機能も用いることが出来る．例えば

```
#ifdef _DEBUG
  デバッグ処理
#endif
```

となっている場合に

```
```

とすれば"デバッグ処理"がコンパイルされる．ここで -D_DEBUG は _DEBUG というマクロを定義するという意味である．マクロを用いたプリプロセッサには様々な機能があり，より複雑な指定も可能である．

```
#if   _DEBUG_MODE == 0
  デバッグモード 0
#elif _DEBUG_MODE == 1
```

(次のページに続く)

[6] ifort では -fpp.

(前のページからの続き)

```
  デバッグモード 1
#elif _DEBUG_MODE == 2
  デバッグモード 2
#endif
```

のような場合には，_DEBUG_MODE という名前のマクロの値[7] によってコンパイルする処理を切り替えることが出来る．マクロの値もコマンドラインから明示的に与えることが出来る．例えば

```
                                =1
```

のように実行すれば，マクロ _DEBUG_MODE の値が 1 に定義され，"デバッグモード 1"の処理がコンパイルされる．

10.5.2 printf デバッグ

デバッグの基本中の基本テクニックは printf デバッグと呼ばれる．printf とは C 言語で標準出力に表示するための関数名で，要するに間違っていそうな箇所に当たりをつけて，途中結果を出力して確認するという古典的な手法である．Fortran の場合にはひたすら write(*,*) で怪しい変数を出力してみれば大抵の場合 (すくなくとも演習の課題レベルの場合) には自ずとバグの原因が見えてくる[8]．

このデバッグ手法はプリプロセッサと組み合わせると便利である．例えば

```
#ifdef _DEBUG
#define DEBUG_PRINT(a) write(*,*) (a)
#else
#define DEBUG_PRINT(a)
#endif
```

としておくと，任意の場所で

```
DEBUG_PRINT(出力したい変数)
```

のように DEBUG_PRINT というマクロを関数 (サブルーチン) のように用いることが出来る．この時，コンパイル時にオプション -D_DEBUG を付けると DEBUG_PRINT の引数に与えた変数の値が出力されるが，-D_DEBUG を付けなければ何も出力されない．つまり，この形式でデバッグメッセージを埋め込んでおけば，ソースコードを一切修正することなく，コマンドラインのみでデバッグモードに切り替えることが出来る．(これは _DEBUG というマクロが定義されている場合には，DEBUG_PRINT(x) が write(*,*) (x) に変換され，それ以外の場合は単なるホワイトスペース (空白) に変換されるためである．)

このようなテクニックは非常に有用で，開発効率を大幅に高めてくれるハズなので，少しずつ身につけていくと良い．

---

[7] 変数のように見えるが，プリプロセッサで (コンパイルされる前に) 処理されるのでプログラム中の変数にはなっていない．

[8] 演習中にディスプレイの前でフリーズしている人を良く見かけるが，考えているだけでは何も進まない．とりあえずヒントを探すという意味で値を出力するのは非常に重要である．

10.5.3 gdb

To be written.

10.6 C 言語とのリンク †

To be written.

10.7 Fortran 77 について †

To be written.

10.8 第 10 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例

10.8.1 課題 1

サンプルプログラムをコンパイル・実行して動作を確認せよ．さらに，適宜修正してその実行結果を確認せよ．

10.8.2 課題 2

任意の 1 変数関数 $f(x)$ およびその微分値を返す関数 $f'(x)$ を引数として受け取り， $f(x) = 0$ の近似解を Newton 法により求めるサブルーチンを実装し，その動作を確認せよ．ただし以下の条件を満たすこと．

- 初期値を引数として受け取る．
- 無限ループにならないように最大の反復回数を引数として受け取る．
- 許容誤差を引数として受け取り, 反復による近似解の変化が許容誤差以下となったら収束したとみなす．
- 収束判定のステータスを返す．
- 求まった近似解を返す (収束しなかった場合は最後の反復後の解の近似値).

(二分法のモジュール `bisection.f90` の `mod_bisection` を参考にすればよい．)

10.8.3 課題 3 †

list.f90 で定義されているリストを実装したモジュール mod_list へ以下の様な機能追加を試みよ．以下では a はリスト (type(list_type)) のポインタであるとする．

- 代入演算子が配列を受け取れるようにせよ．即ち，

```
a = (/0, 1, 2/)
```

のようにリストに配列を代入すると元のリストを破棄し (正しく deallocate し)，与えられた配列でリストを初期化すること．

- サンプルでは append によって単一の値をリストの終端に追加しているが，配列を与えた場合にはその各要素をリストに追加するように拡張せよ．ただしオーバーロードを用いることで，どちらも同じサブルーチン名 append で行うようにせよ．

```
call append(a, 3)
call append(a, (/4, 5/))
```

のようなコードが実行可能となるはずである．

- 同様に insert でも配列を追加できるようにせよ．

```
call insert(a, 1, -1)
call insert(a, 1, (/-3, -2/))
```

のようなコードが実行可能となるはずである．