Fortranでシミュレーションをしよう

(第1.0版)

摂南大学 理工学部 電気電子工学科

田口 俊弘

2011 年 6 月 24 日

はじめに

本書は，これからコンピュータシミュレーションを勉強しようとする学生さんたちのために，コンピュータ言語である Fortran を使ってプログラムを作成するための基礎的な文法を説明したものです。最近の学部における計算機の講義では C 言語で教えるところが増えてきて Fortran は影が薄くなっていますが，コンピュータシミュレーションの分野では Fortran の方がよく使われています。これは，Fortran が科学技術計算用言語として発展してきているので，数値計算に便利な書式が多数採用されているからです。例えば，複素数計算や行列計算などは非常に簡単に書くことができます。また，C 言語のようにフレキシブルな書き方ができないのでタイプする文字の量が多くなりがちですが，そのおかげでスーパーコンピュータなどで実行する時の最適化がしやすいという利点もあります。

しかし，逆にこのことが Fortran を近づきにくいものにしたようです。というのは，数値計算に特化するということはユーザーの専門分野が限られるということであり，その結果コンパイラの値段があまり下がらず，気軽に使うことができなくなってしまったのです。C 言語はその誕生時点から比較的安価に供給され，これが広く使われている理由の一つです。C 言語はそもそも OS を記述するために開発された言語であるため，計算機のハードウェア寄りの書式が多く，必ずしも初心者向きの言語だとは思えないのですが，プログラミングの専門家からすれば Fortran も C 言語も大して変わらないので，安い方にシフトしたのはある意味やむを得ないことだと思います。

しかし，Linux の世界では有志が開発したフリーの Fortran(`gfortran` や `g95`) が使えるようになり，この Windows 版や Mac OS 版も開発されて，インターネット経由でダウンロードすればパソコンでも Fortran が自由に使えるようになりました。コンピュータシミュレーションのプログラムを書くには断然 Fortran の方が書きやすいし，完成したプログラムをそのまま大型コンピュータで高速に実行させることも可能です。是非 Fortran でプログラムを書きましょう。

ただし，コンピュータシミュレーションは複雑なので，どうしてもプログラムが長くなります。このため，単に文法を覚えてそれに従って書くだけでは不十分です。全体的なプログラム書式の統一性を考慮して書かないと，プログラムのチェックや修正をする時に苦労します。このため，多少冗長になっても，読みやすく後でデバッグしやすいように書くべきです。本書はこの点を考慮して解説したので，必ずしも説明通りに書く必要はない部分もあります。しかしそういった部分はプログラムが大きくなるにつれて御利益が現れるので，横着せずに解説の通り書くことをお勧めします。最初は面倒に感じても慣れてくればそれほどの手間ではありません。

そもそもプログラムというのはコンピュータを動作させるための命令手順を書くものです。このため，コンピュータが計算しやすいように書いてあげれば，より高速に計算させることができます。コンピュータシミュレーションは計算速度が重要なポイントなので，本書ではこの点も考慮し，コンピュータの内部構造で知っておいた方が良い部分も簡単に説明しました。孫子曰く,「彼を知り己を知れば百戦殆うからず」と。

最近の Fortran には配列と配列を直接計算することができるような，この解説書では述べていない便利な機能が色々含まれています。本書で基礎的な計算機プログラミング手法を学んだら，このような便利な文法を勉強して，さらに高度なプログラムへと発展させてもらえればと思います。

目　　次

第 1 章　Fortran プログラミングの基礎

本章では，Fortran でプログラムを書くのに必要な最低限の文法を説明します。これらの知識を使って，とりあえず色々な計算問題をプログラムしてみましょう。

Fortran コンパイラは大文字・小文字を区別しないので，どちらでプログラムを書いても同じです。小文字で問題なのは数字の 1 とローマ字の l（エル）の区別が付きにくいことです。

1.1　メインプログラムの開始と終了

メインプログラムとは，プログラムの実行を開始した時に最初に動き出す部分である。Fortran ではいきなりプログラムを書くと。それがメインプログラムと仮定されるのだが，それでは第 2 章で説明するサブルーチンと区別しにくいので，program 文を用いてプログラムの名前を書くようにする。

```
program プログラム名
```

メインプログラムの終了は end 文で指定する。このため，メインプログラムは次のような構造になる。

```
program code_name
    .......
    .......
    .......
    .......
end program code_name
```

プログラム名にはアンダースコア「_」も使えるので，これをスペースの代わりに使えば単語をつないだ長いプログラム名を付けることもできる。なお，end 文はこの例のように program 文の情報を書かず,「end」だけでも良いのだが，プログラムの境界を明確にするためにこの形で書くことを心がけよう。

program 文と end 文の間に Fortran の文法に従った文を使って動作させたい計算手順を記述する。動作手順を記述する文を「実行文」という。しかし，プログラムに記述するのは実行文だけではない。計算途中で必要となる変数領域を確保するための宣言文もプログラム中に書かねばならない (1.2.3 参照)。このような計算動作に直接携わらない文を「非実行文」と言う。Fortran では，非実行文をプログラムの最初に集約して，実行文はその後に書く。このため，動作の開始は program 文の次の文ではなく，最初の実行文である。

完結したメインプログラムの一例を下に示す。

```
program test_code
  implicit none
  real x,y,z
  x = 5
  y = 100
  z = x + y*100
  print *,x,y
  print *, ' z = ' ,z
end program test_code
```

このプログラムにおける実行文・非実行文の区分と，動作開始から終了までの実行の流れを図 1.1 に示す。実行文は基本的に上から順に一行づつ実行される。do 文を使った繰り返し (1.4) や，if 文による条件分岐 (1.5) など，動作指定によっては所定の位置にバックしたり，条件に応じて実行するかどうかの選択ができるが，それでもそれぞれの文が終了した後に次の文が実行されるという基本的動作は変わらない。

これは計算機が一回に一つの動作しかできないためである[1]。プログラム全体を見渡して実行するのではなく，一行一行を順に実行していくのである。このため，バックするような動作指定がなければ，下方に書いた実行文は上方に影響を及ぼさない。プログラムはこのことを念頭に置いて書かなければならない。

[1]最近はマルチコア CPU などを使って一度に複数の動作を実行できるコンピュータも存在するが，それを利用するには複数処理を意識した手続きが必要である。プログラムは基本的に上から順に一行づつ実行するものと考えて書かなければならない。

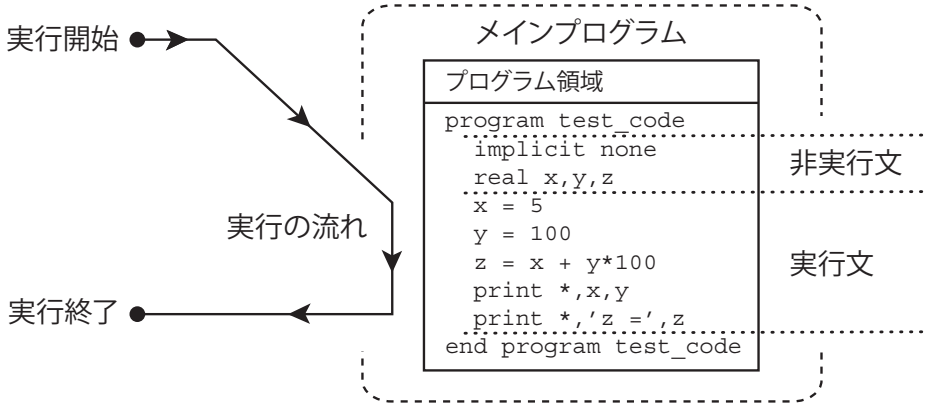

図 1.1　プログラム開始から終了までの流れ

なお，if 文を使って条件によって途中でプログラムを終了するときや，サブルーチン内で終了するときには stop 文を用いる。

```
program fluid_code
  implicit none
  real x,y
  integer n,m
    .......
  if (m<0) stop
    .......
end program fluid_code
```

この例だと，m<0 の時にプログラムは終了し，それ以降は実行しない。

1.2　数値と演算

1.2.1　プログラムにおける基本的な数式

プログラムとは，計算機に一連の計算をさせる手順を与えるものである。このため，基本的には計算を実行した結果をその後の計算式で使えるように保存する必要がある。このため，プログラムにおける数式の基本形は

```
変数 = 計算式
```

である。この式は，右辺の計算式で計算した結果を左辺で示した変数に代入することを意味する。数学で方程式を表す

```
計算式 = 計算式        ! これはエラー
```

という形は使えない。Fortran において「= (イコール)」という記号は,「左辺と右辺が等しい」という意味ではなく,「右辺の計算結果を左辺で指定した変数メモリに保存する」という計算機の動作を表すからである。

例えば,

```
x = 4 + 2
y = 9 - x
y = y + 1
```

のようなプログラムを考えてみよう。プログラムは上から順に実行されるので，まず 4+2 の結果 6 が変数 x に代入される。次に，9 から変数 x に保存されている数値を引いた値が変数 y に代入されるので，y には 3 が代入される。プログラムに慣れない人は，最後の文で「y はいったいいくつなんだ？」ととまどうかもしれない。答えは 4 である。イコール (=) が方程式ではなく，あくまでも「y+1 の計算結果を y という変数に代入する」と理解できれ

ば，この式は不思議でもなんでもない。y はその前の行で 3 が代入されているから，y+1 の結果は 4。よって，最後の行の実行結果は，変数 y に 4 が代入される，となる。変数というのは，数値を入れる箱だと考えればいいだろう。

四則演算などの基本演算は以下のような記号で表す。

演算記号	演算の意味	使用例	使用例の意味
+	足し算	x+y	x + y
-	引き算	x-y	x − y
*	かけ算	x*y	x × y
/	割り算	x/y	x ÷ y
**	べき乗	x**y	x^y

プログラム命令によっては，変数に保存しないで直接計算結果の値を利用することもできる。例えば，関数の引数に計算式を記述する場合がこれに当たる。すなわち，

```
x = 3.14
y = sin(2*x+1)
```

のように，関数 sin(x) の引数に計算式 2*x+1 を使えば，その計算結果を引数とした関数値を計算する。

もう一つよく使うのは，出力文である。計算結果を画面に出力するには print 文を使う。

```
x = 3.14
print *,x,sin(2*x)
```

文頭の print に続く文中で，最初の「*」は出力形式が標準形式であることを意味するので出力はされない。その後に「,」で区切った変数 x と sin(2*x) の値が横に並んで出力される。なお，print 文など，出力文の詳細は 3.1 に説明してあるが，この書式を使えば取りあえず計算結果を出力することができる。

1.2.2 数値の型

コンピュータという機械が取り扱う「数値」は 2 進数で表されていて，基本的には「整数」である。例えば，1byte の数は，8bit，つまり 0 か 1 のどちらかを選択できる数が 8 個並んだものなので，10 進数では 0〜2^8-1(=255) までの整数を表すことができる。しかし，シミュレーションをする場合には，10^{10} とか，1.6×10^{-19} とかの大小様々な数を扱うため，この形式では不便である。

さらに，整数は小数点以下の表現ができないので，計算機で割り算をすると全て「切り捨て」になる。これを忘れてプログラムを書くと必ず間違いを起こす。

例えば，

```
x = (5/2)**2
y = x**(3/2)
t = 1/2*y*x
```

と書くと，x，y，z はいくつになるか？答えは x=4，y=4，t=0 である。

そこで，整数で表される計算機本来の型式「整数型」の他に，1.6×10^{-19} のように，$A\times10^B$ という形の数を取り扱うことができる。これを「実数型」といい，A の部分を「仮数部」，B の部分を「指数部」と呼ぶ。数値計算は，全てこの実数型で計算しなければならない。

実数型は少なくとも 2 種類用意されている。単精度実数と倍精度実数である。単精度実数とは 4byte(=32bit) で表される実数のことで，有効数字は 7 桁程度である。これに対し，倍精度実数とは 8byte (=64bit) で表される実数のことで，有効数字は 15 桁程度である。7 桁あれば問題ないと思うかもしれないが，シミュレーションでは大量の数の合計を計算することが多いため，有効数字が計算回数の増加につれて徐々に減ってくる。このため，通

常は倍精度実数を用いなければならない。Fortran でのデフォルトの実数型は単精度であるが，最近のコンパイラはオプションでデフォルトの実数を倍精度にする自動倍精度化機能を持っているので，本書では単精度実数と倍精度実数を使い分ける手法の説明は省略する[2]。よって，特に単精度で計算する必要がなければコンパイル時に自動倍精度化オプションを付加して倍精度で計算することを忘れないようにしよう。多くの計算機は倍精度実数計算を高速で行えるハードウェアを内蔵しているので，倍精度計算をしてもさほど実行速度が遅くなることはない。

整数型の定数と，実数型の定数は「小数点の有無」で区別する。例えば，`100` とか，`-12345` と書けば，これらは「整数型」である。これに対し，`3.` とか，`1.3` とか，`-0.0314` とか書けば実数型になる。さらに，1.6×10^{23} を入力したい時には，`1.6e23` と書く。即ち，$A\times10^{B}$ は AeB と書くのである。なお，B が負の場合でも `1.6e-19` のように `e` の後に続けて書く。

例えば，

$a = 3.141592 \times r^2 + 6.5 \times 10^{-5} \times x - 10 \times 10^5$

をプログラムで書けば以下のようになる。

```
a = 3.141592*r**2 + 6.5e-5*x - 10e5
```

なお，`z**2` とか `x**5` のように整数べき乗の指数 (2 とか 5) は整数型を使う方が良い。

先ほど整数型を使ったら予想どおりの結果が出ない例を挙げたが，実数を使えば正しい結果が得られる。

```
x = (5.0/2.0)**2
y = x**(3.0/2.0)
y = 0.5*y*x
```

Fortran で便利な機能の一つは，複素数が使えることである。複素数にも単精度と倍精度があるがやはり倍精度複素数型を使うべきである。複素数の定数は

```
(0.0,1.0)
(1e-5,-5.2e3)
(-3200.0,0.005)
```

のように，2 個の実数をコンマでつないで括弧でくくる。前半が実部，後半が虚部である。つまりこの例は，それぞれ，i，$10^{-5}-5.2\times10^{3}i$，$-3200+0.005i$ を表した複素定数である。

なお，ここまで読むと整数型は必要がないように思うかもしれないが，そうではない。1.3 で説明する配列要素を指定する数は整数型でなければならないし，1.4 で説明する `do` 文で用いるカウンタ変数やオン・オフを表すだけの変数など「順番」や「区別」を示すときは整数型を用いる。また実数の整数部を取り出したいときや，剰余 (あまり) を計算するときなども整数型を利用する。数値計算でも整数型の使い道は多い。

計算式中に異なる数値型が混在した時には，精度の高い方の型に合わせて計算をし，その型の値となる。例えば，実数型と整数型の計算は整数型を実数型に変換して実数型と実数型の計算にし，実数型の結果になる。複素数型と実数型の計算の場合にはその実数型を実部とした複素数型にして計算し，結果は複素数型になる。ただし，計算は基本的に左から行うので，注意が必要である。最初の整数型を使った計算例で，

```
t = 1/2*y*x
```

が 0 になるのは，最初に計算されるのが，`1/2` という整数型÷整数型だからである。

[2]自動倍精度化オプションについては付録 B を参照。

1.2.3 変数の宣言

以上は，定数の話である。これに対し，数値を代入した記号で計算する場合には「変数」を使う。変数の名称は，頭文字が a〜z のどれかであれば，後は a〜z，0〜9 が使える。例えば，abc とか k10 等である。

定数と同様に，変数にも「整数型」や「実数型」などの型がある。このため，計算結果を変数に保存する時には型を合わせる必要がある。変数の型を決める文を「宣言文」という。使用する変数を用意する文と考えればいいだろう。計算に用いる変数は，全て宣言しなければならない。宣言は一度しかできず，途中で変更することはできない。

ところが，Fortran には「暗黙の宣言」があり，通常は宣言しなくても文法的な間違いではないので，タイプミスなどで思わぬエラーが発生する可能性がある。これを防ぐため，プログラムの 1 行目で必ず一度次の implicit 宣言をする。この文を挿入しておけば宣言せずに使用した変数があるとコンパイルエラーとなるので，タイプミスのチェックができる。

```
implicit none
```

数値計算は基本的に実数で行うが，実数型変数は次のように宣言する。

```
real 変数名，変数名，...
```

これに対し，整数型の変数は，次のように宣言する。

```
integer 変数名，変数名，...
```

宣言文は，非実行文であり，全てプログラムの先頭に書かなければならない。

```
program test1
  implicit none
  real x,y,z,omega,wave,area
  integer i,k,n,i1,k2
    .......
```

ただし，real や integer 等の宣言文は何行書いても良いし，順番も無関係である。

Fortran の暗黙宣言では，その頭文字が a〜h と o〜z の変数は実数型で，それ以外，即ち頭文字が i〜n の変数は整数型となる。このため，整数型変数の頭文字を i〜n にする習慣がある。全てを宣言する以上，基本的に変数名の付け方に制限はないのだが，整数型は用途が限られているので，頭文字を限定する方が良いだろう。実数型の名称はあまり頭文字にこだわる必要はないが，原則として整数型をイメージする i,j,k,l,m,n の 1 文字変数は使わない方が無難である。

なお，変数名の長さに決まりはないが，プログラム全域にわたって効力を持つ変数に 1 文字程度の簡単な名前を付けることは避けなければならない。これは，短い変数名を使うとプログラムが大きくなるにつれてどこでその変数を使っているかを検索するのが難しくなるからである[3]。

複素数変数の宣言文は

```
complex 変数名，変数名，...
```

である。複素数は，習慣として頭文字を c か z にすることが多い。

変数に数値を代入する時は，右辺の計算結果を左辺の変数の型に変換して代入する。このため，実数の計算結果を整数型の変数に代入すると小数点以下は切り捨てになる。

例えば，

[3] プログラムにおいて数値を変数に保存する意義は大別して二種類ある。一つはプログラム全域にわたって共通してその値を利用するためであり，もう一つは限定された範囲で一時的に保存するためである。この二つは意識して使い分けてプログラムを作成しなければならない。特に前者の大域的な変数には長めで意味のある名前を付けるべきである。

```
real x
integer n
x = 3.14
n = x + 6
```

の結果，n に代入される値は 9 である。

また，複素数型の計算結果を実数型の変数に代入すると，その複素数の実部が代入される。

例えば，

```
real x
complex c
c=(1.0,-2.0)
x=c**2
```

とすると，x=-3 になる。

逆に，複素数型の変数に実数型の数を代入すると，実部に結果が代入され虚部は 0 となる。例えば，

```
real x
complex c
x=5
c=x**2
```

とすると，c=(25.00000,0.00000) になる。

1.2.4 数学関数

数値計算上よく使う関数はあらかじめ用意されている。これらを「組み込み関数」という。代表的な数学関数を次表に示す。これらの組み込み関数は型宣言をする必要がない。また，引数の型に応じて計算する「総称名」機能があるので，引数 x が複素数でも使える。次表中の必要条件と関数値の範囲は引数が実数型の場合である。もし必要条件を満たさない実数型値を与えるとエラーになる。しかし複素数型を与えた場合には必ずしもエラーにはならない。

基本的な数学関数 (必要条件と関数値の範囲は x が実数型の場合)

組み込み関数	名　称	数学的表現	必要条件	関数値の範囲
sqrt(x)	平方根	$\sqrt{x}$	$x \geq 0$	
abs(x)	絶対値	$\|x\|$		
sin(x)	正弦関数	$\sin x$		
cos(x)	余弦関数	$\cos x$		
asin(x)	逆正弦関数	$\sin^{-1} x$	$-1 \leq x \leq 1$	$-\frac{\pi}{2} \leq \sin^{-1} x \leq \frac{\pi}{2}$
acos(x)	逆余弦関数	$\cos^{-1} x$	$-1 \leq x \leq 1$	$0 \leq \cos^{-1} x \leq \pi$
atan(x)	逆正接関数	$\tan^{-1} x$		$-\frac{\pi}{2} \leq \tan^{-1} x \leq \frac{\pi}{2}$
atan2(y,x)	逆正接関数	$\tan^{-1} \frac{y}{x}$		$-\pi \leq \tan^{-1} \frac{y}{x} \leq \pi$
exp(x)	指数関数	e^x		
log(x)	自然対数	$\log_e x$	$x > 0$	
log10(x)	常用対数	$\log_{10} x$	$x > 0$	

1.2.1 で示したように，x という引数に数式を書いても良いし，数学関数を使った式も書ける。

例えば，

```
c = sin(10*x+3) - 2*tan(-2*log(x))**3
```

のように書くことができる。

なお，$\sqrt{2}$ を計算したくて，sqrt(2) と書くと文法エラーになる。なぜなら，2 は整数型であり，整数型の平方根は用意されていないからである。sqrt(2.0) と書かなければならない。

1.3 配列

1.3.1 配列宣言

変数 abc と書けば，変数は 1 個である。これは「1 個の数値を記憶しておくことができるメモリ」を意味する。しかし，シミュレーションでは，100 万個の粒子の位置と速度を保存して，それらを全部変化させる，なんていうのが当たり前のように出てくる。このときそれぞれに名前を付けて，a,b,c... なんてことをしていては名前を考えるのも大変だし，宣言文だけで膨大になってしまう。そこで，数列 $a_1, a_2, ...$ というように変数に添字を付けて区別するように，数字で区別した変数を作ることができる。これを「配列」と言う。

配列も一種の変数なので，型宣言文を使って宣言する。単一の変数と異なるのは，宣言時に添字の範囲を示す数値を付加することである。例えば，次のように宣言する。

```
real a(10),sum(20,30)
complex cint(10,10)
integer node(100)
```

ここで，添字が 1 個の配列を 1 次元配列，2 個の配列を 2 次元配列という。3 次元以上の配列を作ることもできる。配列の名前の付け方，頭文字の選択などの原則は変数名の付け方と同じである。

宣言した配列の各部の名称は以下の通りである。

real a(10) と宣言した配列について

記号	名称
a	配列名
a(3) など	配列要素
かっこ中の数字	要素番号，添字

配列の要素番号は，1 から宣言文で指定した数値までが使用できる。上例の a(10) の宣言では a(1) から a(10) までの配列要素が使用可能である。もし，a(100) のように範囲外の要素番号を指定して使用すると，コンパイルエラーにはならないが実行時にエラーになる可能性が高いので注意しなければならない。

1.3.2 メモリ上での配列の並び

配列はコンピュータ内部において連続したメモリ領域で実現されている。例えば,「real a(10)」で宣言された配列は，図 1.2 の左図で示されるように，a(1),a(2)...a(10) の順で並んだ実数型のメモリである。例えば，a(3) とは，a(1) から数えて 3 番目のメモリという意味である。図に示したように配列を配列名 a で代表させることもできる。これは配列をサブルーチンの引数に与える時などで利用する。

最初の要素番号を 1 にしているのは，単なるコンパイラの仕様である。これを変更できれば，上限だけでなく下限も指定した範囲指定ができる。Fortran ではこれが可能である。例えば，0 や負数の要素番号を使いたいときは次のように「:」を使って下限を指定した宣言をする。

```
real ac(-3:5)
```

この時は，ac(-3) から ac(5) までが使用可能となる。この場合，図 1.2 の右図のようにメモリ上での先頭要素は ac(-3) である。

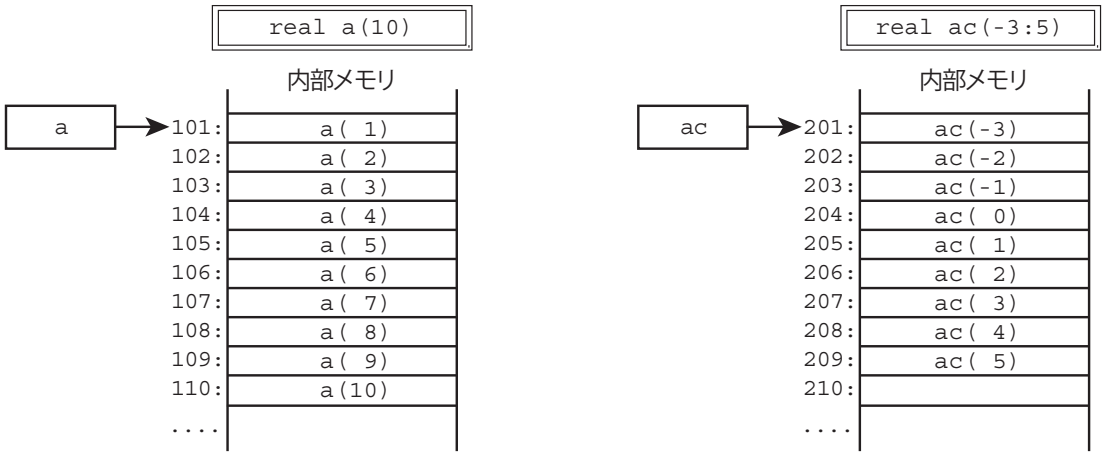

図 1.2　1 次元配列のメモリ並び

2 次元以上の配列の場合は，左の方の要素番号から先に進むようにメモリ上で並んでいる。
例えば

```
real b(3,2)
```

と宣言した場合，メモリ上での並びは図 1.3 のようになる。2 次元以上の配列も配列名で代表させることができる。

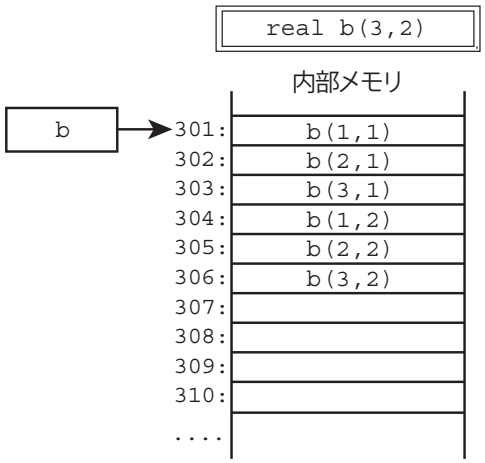

図 1.3　2 次元配列のメモリ並び

2 次元と 3 次元の配列の並び方を正確に表現すると

```
real b(m,n)       の配列宣言で b(i,j) という要素は先頭から数えて i+m*(j-1) 番目
real b(l,m,n)     の配列宣言で b(i,j,k) という要素は先頭から数えて i+l*(j-1+m*(k-1)) 番目
```

となる。この並び方は憶えておいた方が良い。また，どちらも一番右側の要素数 n には依存しない。

1.4 手順の繰り返し — do 文

プログラムは原則的に一番上から順に実行される。しかし，それだけだと同じ手順を繰り返す時に，必要な回数だけ同じ文を書かねばならない。そこで，ある範囲の手順を必要な回数だけ繰り返し行わせるときは do 文を使う。do 文の基本形は，

```
do m = m1, m2
  .......
  .......
enddo
```

である。do 文と enddo 文に囲まれた範囲の文が繰り返し実行される。この範囲を「do ブロック」といい[4]，do ブロック内の手順を一回実行する毎に変数 m が変化する。この m をカウンタ変数と呼び，整数型の変数を使う。カウンタ変数 m は m1 から始まって，1 ずつ増加しながら m=m2 になるまで do ブロック内の手順を繰り返す。繰り返し終了後は enddo 文の次の文に移る。

例えば，

```
do m = 1, 10
  a(m) = m
enddo
```

と書けば，a(1)〜a(10) までの配列要素に，1〜10 までの数値がそれぞれ代入される。

次のようにもう一つの整数 m3 を指定することで増加量を m3 にすることもできる。

```
do m = m1, m2, m3
  .......
  .......
enddo
```

このとき m は m1 から開始して，m3 ずつ増加しながら do ブロック内の手順を繰り返す。そして，m に m3 を加えた結果が m2 を越えた時に終了する。

例えば，m が 10 以下の奇数の時のみ計算したい時は，

```
do m = 1, 10, 2
  .......
  .......
enddo
```

と書く。この時，終了値 m2 は 10 であるが，10 は奇数ではないので計算をしないことに注意しよう。

m3<0 でもよい。m3<0 の時には m に m3 を加えた結果が m2 より小さくなった時に終了する。例えば，100 から順に下って 1 まで繰り返す時には以下のように書く。

```
do m = 100, 1, -1
  .......
  .......
enddo
```

なお，m3 を省略するか m3>0 のときに m1>m2 ならば，do ブロック内を一回も実行せずに処理が enddo 文の次へ移る。m3<0 の時は m1<m2 ならば一回も実行されない。

do 文のカウンタ変数を増加するタイミングは，enddo 文実行時である。do 文の動作がわかりにくい人は，一度手動で展開してみると良い。

例えば，

[4] プログラムの流れが循環するという意味で，do ループとも言う。

```
do m = 1, 3
  a(m) = m**2
enddo
```

という文は,

```
m = 1
a(m) = m**2
m = m + 1
a(m) = m**2
m = m + 1
a(m) = m**2
m = m + 1
```

と等価である[5]。この展開からわかるように，do ブロックを終了した時，カウンタ変数は終了値より大きい値になる。上例では，do ブロック終了後，m は 4 (= 3+1) になっている。do ブロック終了時の m の値を使用する時は，このことを考慮する必要がある。

do 文のカウンタ変数は整数型でなければならないので，実数を一定間隔で変化させたい時には次のように書く必要がある[6]。

```
do m = 0, 10
  x = 0.1*m
  y(m) = sin(x+3)
enddo
```

この場合，x は 0.0 から開始して，1.0 まで計算した後に終了する。

do ブロックの中に別の do ブロックを入れることもできるが，カウンタ変数は異なるものを使う必要がある[7]。

```
do k = 1, 100
  a(k) = k**2
  .......
  do m = 1, 10
    b(m,k) = m*k
    .......
  enddo
  .......
enddo
```

よく使うので覚えておく必要があるのは，配列に代入された数値の総和を求めるプログラムである。一次元配列，a(1)，a(2)，....，a(n) 中に数値データが入っている場合に，これらの合計を求めるにはどうすればいいだろうか。最後の配列要素番号 n がわかっていて小さければ，a(1)+a(2)+a(3) のように計算することもできるが，n が変数の場合や，非常に大きい場合には使えない。これは，do 文を使って以下のように書く。

```
sum = 0
do m = 1, n
  sum = sum + a(m)
enddo
```

この文がちゃんと合計を計算していることを理解するには，やはり以下のように展開してみると良いだろう。

[5]実際には m=m+1 の後で m と m2 を比較する条件文が入るのだがここでは省略している。

[6]昔の Fortran では do 文のカウンタ変数として実数を使うこともできたのだが，Fortran95 から廃止された。この名残で，多くのコンパイラでは警告を出すだけで使えないわけではないようである。しかし廃止された以上は使うべきではない。

[7]これは，プログラムの動作がややこしくなるためではなく，Fortran では do ブロック内でカウンタ変数を変更することが禁止されているからである。

```
sum = 0
m = 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
   ......
```

ここで重要なことは，do 文の前であらかじめ変数 sum を 0 にしていることである。これがないとうまく働かない。このようなプログラムならではの書き方はパターンとして覚えておくと良い。

1.5 条件分岐 — if 文

条件に応じて異なる手順を行わせるプログラムを書く時には if 文を使う。もっとも単純に，一つの条件に応じて一つの文を実行するかしないかを決めるだけの時は単純 if 文を使う。単純 if 文は以下の形式で与える。

```
if（条件）実行文
```

この文はかっこ内の条件が満足されれば，その右の実行文を実行し，条件が満足されなければ何もしないで次の文に動作が移る。

例えば，

```
a = 5
if (i < 0) a = 10
b = a**2
```

と書くと，i が負の場合には a=10 となり，それ以外は a=5 のままなので，それに応じて b に代入される値が異なるという動作になる。しかし単純 if 文は実行文が一つしか書けないので，実行したい文が複数必要なときには使えない。また，条件に合った時の動作指定はできるが，合わなかった時の動作指定はできないので，条件に合った時と合わなかった時で全く違う動作をさせるには不便である。

そこで，通常は単純 if 文ではなく，ブロック if 文を使う。if 文の実行文のところを then にして if 文を終了し，続けて条件を満足した場合に実行する文を書く。複数行書いても良い。実行する範囲は最後の実行文の次に endif 文を書くことで指定する。例えば，次のように書く。

```
a = 5
b = 2
if (i < 0) then
   a = 10
   b = 6
endif
c = a*b
```

この場合，i が負の場合には a=10，b=6 となり，それ以外は変化がない。つまり a=5，b=2 のままである。if 文から endif 文までの範囲を「if ブロック」という。

もっとも，この例では，i≥0 の時，すなわち i<0 という条件を満足しないときにはあらかじめ，a=5，b=2 という代入をする必要がない。このように「条件を満足しないとき」に別の動作をさせたいときには if ブロック中に else 文を挿入する。else 文から endif 文までが条件を満足しないときに実行される。

例えば上記のプログラムは，

```
if (i < 0) then
   a = 10
   b = 6
 else
   a = 5
   b = 2
endif
c = a*b
```

と書くことができる。

また，条件を満足しない場合に，さらに別の条件を指定したいときには elseif 文を使う。elseif 文も条件はカッコで指定し，その後に then を書く。

```
if (i < 0) then
   a = 10
   b = 6
 elseif (i < 5) then
   a = 4
   b = 7
 else
   a = 5
   b = 2
endif
c = a*b
```

この場合，i<0 の場合には a=10，b=6 を実行，0 ≤i< 5 の場合には a=4，b=7 を実行，それ以外は a=5，b=2 を実行する。

elseif 文による新たな条件は if ブロック中にいくつ入れてもかまわない。その場合は，その elseif 文より以前の条件を全て満足しない場合に，その条件を満足すれば，という意味になる。これに対し，else 文は最後の一回しか使えない。

数値計算でよく使う代表的な条件を以下に示す。

```
  ==     左辺と右辺が等しいとき
  /=     左辺と右辺が等しくないとき
   >     左辺が右辺より大きいとき
  >=     左辺が右辺以上のとき
   <     左辺が右辺より小さいとき
  <=     左辺が右辺以下のとき
```

これらの条件は以下の記号で論理的につなぐこともできる。

```
  .and.     左辺かつ右辺のとき
  .or.      左辺または右辺のとき
```

例えば，

```
if (i > 0 .and. i <= 5) then
  a = 10.0
 else
  a = 0.0
endif
```

と書くと，i が 0 より大きく 5 以下の時，すなわち，0 <i≤ 5 のときに a=10，それ以外は a=0 となる。横着して，この if 文を

```
if (0 < i <= 5) then      !  これはエラー
```

と書くことはできないので注意しよう。

1.6 無条件ジャンプ — goto 文，exit 文，cycle 文

プログラムというのは基本的に上から順番に実行していくものである。do 文を使えば，do ブロックで指定した範囲のプログラムを一定の回数繰り返すことができるが，繰り返す範囲は固定されているし，繰り返しを終了する条件はカウンタ変数の増加だけで決まる。

これに対し，より一般的にプログラムの流れを変えたい時，例えば途中で計算を中断してプログラムの最初からやり直す，とか，最後の文に一気に移動して終了する，とかいう時には goto 文を使う。goto 文を使えば指定した行へ強制的に移動することができる。計算機的に言えば「無条件でジャンプする」のである。goto 文とは以下のように，goto の後に整数値を指定した形の文である。

```
   goto 整数値
```

この goto の後の整数値がどの行にジャンプするかを指定するための数値で,「文番号」と呼ばれている。文番号はジャンプ先の行に書かれた実行文の「前に」スペースを 1 個以上空けて書く。

例えば,

```
   cd = 10
   goto 100
   cd = 50
   ab = 20
   ij = 1
100  ab=1000
```

と書く。最後の ab=1000 の前の 100 が文番号である。この例では，最初の cd=10 の実行後，cd=50 から ij=1 までの文は実行されず，直ちに ab=1000 が実行される。

次例のように戻ることも可能である。この場合，指定した文番号の行と goto 文の間の動作を繰り返し実行する。

```
   cd = 50
100  ab = 200
   cd = cd + ab - ef
   ef = 10
   goto 100
   ij = 1
```

ただし，この例ではいつまでたっても goto 文の次の ij=1 が実行されない。こういうのを「無限ループ」と呼んで，プログラムエラーの一つである。計算結果に応じて goto 文より下の行へジャンプする別の goto 文やプログラム自体を終了させる stop 文を挿入しなければならない。

なお，一般の Fortran の教科書には「ジャンプ先の文は continue 文にしておく方が良い」と書いてあることが多い。continue 文とは「何もしない」文である。例えば，上記のプログラムは,

```
   cd = 50
100 continue
   ab = 200
   cd = cd + ab - ef
   ef = 10
   goto 100
   ij = 1
```

と完全に同じである[8]。

goto 文は多用するとプログラムの流れがわかりにくくなるし，無限ループに陥る可能性もあるので使用はできるだけ避けた方がよい。基本的なループや条件に応じたジャンプは do ブロックと if ブロックでほとんどすべて書くことができる。

do ブロックを使って繰り返し計算をしている時に，条件によって途中で繰り返しを終了したい場合がある。この時，原理的には goto 文を使わなければならないが，goto 文の使用を極力避けるという観点から exit 文が用意されている。

例えば，

```
   do m = 1, n
     sum = sum + a(m)
     if (sum > 100) exit
   enddo
   sum = sum/n
   ......
```

のように書いたプログラムは，次の goto 文を使ったものと同じである。

```
   do m = 1, n
     sum = sum + a(m)
     if (sum > 100) goto 10
   enddo
10 sum = sum/n
   ......
```

すなわち，exit 文を実行すると do ブロックの外に飛び出して enddo 文の直後から実行を開始する[9]。

また，条件を満足した時に do 文の残りの部分をスキップして，カウンタ変数を進める時には cycle 文を使う。

例えば，

```
   do m = 1, n
     sum = sum + a(m)
     if (sum > 100) cycle
     sum = sum + a(m)*2
   enddo
      ......
```

のように書いたプログラムは，次の goto 文を使ったものと同じである。

```
   do m = 1, n
     sum = sum + a(m)
     if (sum > 100) goto 10
     sum = sum + a(m)*2
10 enddo
   ......
```

すなわち，cycle 文を実行するのは実効的に enddo 文にジャンプしたのと同じことになる。enddo 文にジャンプすれば，カウンタ変数 (この例では m) を増加して m>m2 かどうかをチェックした後，条件を満足しなければ再び do ブロックを最初から実行する。

ただし，cycle 文は do 文内部の制御なので次のように if ブロックを使って同じ動作をさせることができる。

[8]筆者が思うにこの習慣はプログラムを 1 行あたり 1 枚のカードに書いていた時代の名残である。プログラムを画面上で編集することができるエディタで書く今日，continue 文にあまりこだわることはないと思う。

[9]このプログラムは a(1)〜a(m) までを合計して，その合計が 100 を越えたら終了し，その合計を n で割った値を計算するのが目的なのでこれでいいのだが，n で割った値ではなく，a(1)〜a(m) の平均値を計算したい場合には単に n の代わりに m で割るように書き換えるだけでは不完全である。なぜだか考えてみよう。

```
do m = 1, n
  sum = sum + a(m)
  if (sum <= 100) then
     sum = sum + a(m)*2
  endif
enddo
......
```

この方が，条件に応じた動作を明示している点で好ましい。goto 文を多用しない方がよい，という意味合いと同じで，cycle 文を使うとプログラムがわかりにくくなるのであまり使わない方が良いと思われる。

1.7 プログラムのチューンアップ

大規模なシミュレーションプログラムを書くときには，計算効率や精度を常に念頭におく必要がある。ここでは，これらを高めるいくつかの方法を示す。プログラム作りに慣れてきたら，できるだけ心がけよう[10]。

1.7.1 演算の速度を考える

コンピュータの計算動作は加算が基本である。減算は負数に変換して加算するだけなので加算とそれほど実行時間は変わらないが，乗算は加算の繰り返し動作のため，加減算よりかなり遅い。除算にいたっては，減算の繰り返しを条件付きで行うのだからさらに遅い。速さの比較を書けば次のようになる。

```
加減算　>>　乗算　>>>>　除算
```

このため，割り算はできるだけ避ける方がよい。例えば，

```
x = a/b/c
```

と書くより，

```
x = a/(b*c)
```

と書く方が速い。

do ブロック内で何度も同じ割り算をするときには，逆数を掛けるように書き換える。例えば，

```
do i = 1, 100
  a(i) = b(i)/c
  x(i) = a(i)/10.0
enddo
```

と書くより，

```
d = 1.0/c
do i = 1, 100
  a(i) = b(i)*d
  x(i) = a(i)*0.1
enddo
```

と書く方が速い。もっとも，プログラムがわかりにくくなる欠点もあるので，割り算の箇所が少なく，繰り返しの回数もさほど多くない時にはそれほど神経質に変形する必要はない。

べき乗算はもっと遅いので，2 乗・3 乗程度のときは，掛け算にする方が速い。例えば，

[10]最適化のオプションを付けてコンパイルすると，この節で述べる高速化技法の一部は自動的に行われる可能性がある。このため実際にはそれほど高速化の効果は出ないかもしれない。しかし最適化はコンパイラの性能に依存するので，どこまで当てにできるかは不明である。手動で高速化できるものは手動で書き換えた方が確実である。

```
x = a**2 + b**3
```

と書くより，

```
x = a*a + b*b*b
```

と書く方が速い。ただし，指数が大きいときには意味がないし，プログラムもわかりにくくなる。

べき乗の中で，$\frac{1}{2}$ 乗に関しては，組み込み関数 sqrt を利用する方が速い。例えば，

```
y = x**0.5
z = y**1.5
```

と書くより，

```
y = sqrt(x)
z = sqrt(y)**3
```

と書く方が速い。

1.7.2 多項式を計算する手法

多項式を計算するときは，掛け算の回数ができるだけ少なくなるように考える。一番良いのは horner 法である。

例えば，

```
y = a0 + a1*x + a2*x*x + a3*x*x*x + a4*x*x*x*x
```

と計算すると，乗算が，10 回必要である。これを

```
y = a0 + (a1 + (a2 + (a3 + a4*x)*x)*x)*x
```

と書けば，乗算は 4 回で良い。この計算手法を horner 法と言う。一般的に，horner 法は 0 の係数が少ない多項式の計算に有効である。

もし，a0,a1,a2,a3,a4 が，a(0),a(1),a(2),a(3),a(4) という配列に入っているなら，次のような do 文を使って計算できる。

```
n = 4
y = a(n)
do i = n-1, 0, -1
  y = a(i) + x*y
enddo
```

1.7.3 同じ計算は繰り返さない

例えば，

```
do i = 1, 100
  a(i) = b(i)*c*f
  b(i) = sin(x)*a(i)
enddo
```

と書くと繰り返し毎に c*f や sin(x) を計算する。これらは do ブロック内では変化しないのだから，あらかじめ計算しておく方が速い。

例えば，

```
d = c*f
s = sin(x)
do i = 1, 100
  a(i) = b(i)*d
  b(i) = s*a(i)
enddo
```

のように変形する。

1.7.4 do 文を入れ子にするときの順序

配列を使った繰り返し計算をするときは，メモリをできるだけ連続的に参照していく方が速い。このため，2 次元以上の配列計算をするときには左の要素から順に進める方が速くなる。

例えば，

```
real b(10,100)
integer m,n
do n = 1, 100
  do m = 1, 10
    b(m,n) = m*n
  enddo
enddo
```

のように，2 次元配列 b(m,n) に代入するときは m に関する do ブロックを内側に持ってくる方が高速である。

1.7.5 桁落ちに気を付ける

コンピュータの内部精度は倍精度でも 15 桁程度である。よって，接近した 2 個の実数の引き算をするときは気を付ける必要がある。

例えば，

2000.06 - 2000.00 = 0.06

なので，計算結果 0.06 は元の 2000.06 と比べると有効数字が 5 桁も落ちている。よって，こういう差の計算はできるだけ避けねばならない。

例えば，2 次方程式

$$ax^2 + bx + c = 0 \tag{1}$$

の 1 根は，

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

であるが，$b > 0$ かつ $b^2 \gg |4ac|$ の時，b と $\sqrt{b^2 - 4ac}$ がほぼ等しいので，$-b + \sqrt{b^2 - 4ac}$ の計算をすると桁落ちする可能性がある。そこでこの根を計算する時は，分子の有理化をする。即ち，分母分子に $-b - \sqrt{b^2 - 4ac}$ を掛ければ，

$$x_1 = \frac{\left(-b + \sqrt{b^2 - 4ac}\right)\left(-b - \sqrt{b^2 - 4ac}\right)}{2a\left(-b - \sqrt{b^2 - 4ac}\right)} = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{3}$$

となるが，最後の式を使えば桁落ちする心配はない。2 根とも計算する時は，2 根の積が c/a であることを利用して，まず桁落ちしない方の根 x_1 を計算した後，もう一方の根を $c/(ax_1)$ で計算すると良い。

大きな数に小さい数を加えると，小さい数が桁落ちして情報が失われる可能性がある。例えば，1 次元配列 a(i) の合計を計算するプログラムは，

```
s = 0.0
do i = 1, n
  s = s + a(i)
enddo
```

で良いのだが，n が非常に大きい場合には，合計である s がだんだん大きくなるので，後の方で加えた a(i) の情報が失われる可能性がある。倍精度実数を使うよう指導している理由の一つがこれである。これを防ぐには，たとえば，2 個ずつ加えてそれらを別の配列に入れ，その後それらを同様に 2 個ずつ加えていって，... とするのが良いのだが，これではプログラムが煩雑になってしまう。

ひとつの比較的簡単な解決法は，補助変数 r を使って，誤差を別に評価しておく方法である。

```
s = 0.0
r = 0.0
do i = 1, n
  r = r + a(i)
  t = s
  s = s + r
  t = s - t
  r = r - t
enddo              .....（参考文献，数値計算の常識，伊理正夫・藤野和建，共立出版）
```

倍精度計算をしていれば必ずしもこの方法にする必要はないが，精度の良い合計計算が必要になった時のために憶えておいても良いだろう。

桁落ちしそうな引き算は可能な限り避けるようにする。例えば，正の数の dt(i) という 1 次元配列が与えられていて，これから

```
t(1) = 0.0
do i = 2, n
  t(i) = t(i-1) + dt(i)
enddo
```

により t(i) という 1 次元配列を作ったとする。この t(i) を使って t(i+1)-t(i) や t(i+1)-t(i-1) の値が必要な時には，もし dt(i) が保存されているのであれば，直接計算をしないで dt(i) や dt(i)+dt(i-1) を使うべきである。

演 習 問 題 1

(1–1)　2 次方程式の根 (その 1)

3 個の実変数 a，b，c に適当な値を代入し，それから，

$$ax^2 + bx + c = 0$$

の 2 根を計算して出力するプログラムを作成せよ。さらに，その根を $ax^2 + bx + c$ に代入して 0 に近い値になることもプログラムして確かめよ。また，a，b，c の値が不適切だと，エラーが出ることも確認せよ。(例えば，$a = b = c = 1$ にしてみよ)

(1–2)　ヘロンの公式

三角形の 3 辺の長さを a，b，c とすると，その三角形の面積 S は次式（ヘロンの公式）で表される。

$$S = \sqrt{s(s-a)(s-b)(s-c)}$$

ただし，$s = \dfrac{a+b+c}{2}$ である。

a，b，c に適当な数値を与え，この公式を使って三角形の面積を計算するプログラムを作れ。さらに，一辺 a の正三角形の面積を別の方法で計算するプログラムを作成し，ヘロンの公式から求めた面積と一致するかどうか確かめよ。

(1–3)　面積，体積

1 個の実変数 a と整変数 n に適当な数値を与え，これから，半径 a の円の面積，半径 a の球の体積，一辺 a の正 n 角形の面積を計算するプログラムを作成せよ。ただし，π の値はできるだけ正確な値（15 桁程度）を使うこと。

(1–4)　ビオ・サバールの法則

線分 PQ に電流 I が流れている時，その線分から距離 r の点 A にできる磁束密度 B は次の公式で与えられる。

$$B = \frac{\mu_0 I}{4\pi r}(\cos\theta_P - \cos\theta_Q)$$

ここで，μ_0 は真空の透磁率（$4\pi \times 10^{-7}$），θ_P，θ_Q はそれぞれ点 P，Q から点 A をみた見込み角である。

この公式で，I，r，θ_P，θ_Q を入力して磁束密度 B を計算するプログラムを作れ。ただし，角度は「度」で与えて，内部でラジアンに換算するようにせよ。

次に，一辺 a の正三角形の導線に電流 I が流れている時，中心にできる磁束密度を計算するプログラムを作れ。

さらに，一辺 a の正方形の導線に電流 I が流れている時，中心にできる磁束密度を計算するプログラムを作れ。

(1–5)　回路計算

3 個の抵抗 R_1，R_2，R_3 がある時，この 3 個を直列にした時の合成抵抗，3 個を並列にした時の合成抵抗，R_1 と R_2 を並列にしたものと R_3 を直列にした時の合成抵抗を計算するプログラムを作れ。

(1–6)　繰り返し出力

i=1,2,3...,n の整数に対し，i，i^2，$\frac{1}{i}$，$\sqrt{i}$ の値を並べて出力するプログラムを作れ。なお，整数と実数の変換に注意すること。

(1–7)　3 次元ベクトルと電磁気学

3 次元ベクトル $\mathbf{A} = (A_1, A_2, A_3)$ を要素数 3 の 1 次元配列 `a(3)` で表すとする。まず電場ベクトル $\mathbf{E}$，磁場ベクトル $\mathbf{B}$ を配列で表して適当な数字を代入する。次に荷電粒子の速度ベクトル $\mathbf{v}$ を配列で表して適当な数字を代入し，これらの配列から，内積 $\mathbf{v} \bullet \mathbf{E}$，および外積ベクトル $\mathbf{v} \times \mathbf{B}$ を計算するプログラムを作れ。外積ベクトルも配列にすること。

(1–8)　統計計算

1 次元配列，`a(n)` に，1,2,3... のデータを順に `n` 個代入し，

$$\text{平　均}\ \ \bar{A} = \frac{1}{N}\sum_{i=1}^{N} A_i$$

$$\text{標準偏差}\ \ \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(A_i - \bar{A})^2}$$

を計算するプログラムを作成せよ。

次に，それに続いて `a(n)` に 1,3,5... と奇数を順に `n` 個入れて同様に平均と標準偏差を計算するプログラムを追加せよ。

(1–9)　定積分

等間隔 h 毎に並んだ座標点 $x_1(= a)$，x_2，$\cdots$，$x_n(= b)$ に対して，関数値，$f(x_1)$，$f(x_2)$，$\cdots$，$f(x_n)$ があるとする。このとき $f(x)$ の定積分を

$$\int_a^b f(x)dx = h\left(\frac{1}{2}f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right)$$

と近似的に計算する方法を台形公式という。以下の関数と積分範囲の定積分値を計算し，分割数 n が大きくなるほど正確な値に近づくことを確かめよ。

1. $f(x) = \sin(x)$，$a = 0$，$b = \pi$
2. $f(x) = (x-2)^3$，$a = 1$，$b = 5$

(1–10)　テーラー展開

テーラー展開を利用して指数関数，三角関数を計算するプログラムを作成せよ。但し，テーラー展開の n 次まで計算するとして，変数値 x は適当に決め，n が大きくなるほど，正確な関数値に近づくことを確かめよ。

$$\begin{aligned}
e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}\cdots &&= \sum_{n=0}^{\infty}\frac{x^n}{n!} \\
\cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!}\cdots &&= \sum_{n=0}^{\infty}(-1)^n\frac{x^{2n}}{(2n)!} \\
\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!}\cdots &&= \sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{(2n+1)!}
\end{aligned}$$

(1–11)　漸化式

次の漸化式で計算される数値を，それぞれ，配列 `a(n)`，`b(n)` に代入し，平均と標準偏差を計算せよ。

$$a_{n+1} = 3a_n + 2, \qquad a_1 = -1$$

$$b_{n+1} = \frac{2}{3}b_n - 4, \qquad b_1 = 3$$

演 習 問 題 2

(2–1)　2 次方程式の根 (その 2)

3 個の実変数 a，b，c に適当な値を代入し，それから，

$$ax^2 + bx + c = 0$$

の根を計算して出力するプログラムを作成せよ。今回は，根の判別式，$D = b^2 - 4ac$，を計算して，2 実根になった時，重根になった時，虚根になった時でそれぞれ正しい結果を出力するようにせよ。さらに，その根を $ax^2 + bx + c$ に代入して 0 に近い値になることを判別式の値に応じて手法を変えて確かめよ。

(2–2)　非線形方程式の解法 1

$x_0 = 0$ として，次のように次々に代入していくと，x_n は徐々に $\cos x = x$ の解に近づくことがわかっている。

$$\begin{aligned} x_1 &= \cos x_0 \\ x_2 &= \cos x_1 \\ x_3 &= \cos x_2 \\ &\vdots \\ x_{n+1} &= \cos x_n \end{aligned}$$

最初に，この代入を 100 回繰り返し，その後で 101 回目の値と 100 回目の値の差を計算するプログラムを作成せよ。

それができたら，収束条件を $|x_{n+1} - x_n| < \epsilon$ とし，ϵ を適当に小さい値に定めて (例えば，10^{-7})，収束したら x_n を出力して終了するプログラムにせよ。このとき，収束するまでの回数も出力せよ。

なお，この問題では配列を使わないこと。

(2–3)　非線形方程式の解法 2: Newton 法

次の公式を繰り返し用いて方程式 $f(x) = 0$ の根を近似的に求めるアルゴリズムを Newton 法という。

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

以下の関数 $f(x)$ に対し，適当な初期値 x_0 を与えて Newton 法で根を求めるプログラムを作成せよ。また収束条件を $|x_{n+1} - x_n| < \epsilon$ とし，ϵ は適当に定めよ。このとき，収束するまでの回数も出力せよ。

$$\begin{aligned} f(x) &= x - \cos(x) \\ f(x) &= x^3 - 2 \end{aligned}$$

それぞれの問題に対し，得られた結果を $f(x)$ に代入して答えがどの程度 0 に近いかを調べよ。

(2–4)　常微分方程式 (初期値問題) の解，サイクロトロン運動

一様な磁場中での荷電粒子の速度 $\mathbf{v}(t) = (u(t), v(t))$ は次の運動方程式を満足する。この運動方程式をオイラー法で解け。ただし，サイクロトロン周波数 ω_c は一定とする。

$$\begin{aligned} \frac{du}{dt} &= \omega_c v \\ \frac{dv}{dt} &= -\omega_c u \end{aligned}$$

ここで1回の時間ステップを Δt として $f_n = f(n\Delta t)$ とした時に，オイラー法とは n ステップ目の値 f_n と $n+1$ ステップ目の値を使って $\dfrac{df}{dt} \simeq \dfrac{f_{n+1} - f_n}{\Delta t}$ という近似で時間微分を評価する方法である。例えば，x 方向は

$$\frac{u_{n+1} - u_n}{\Delta t} = \omega_c v_n$$

と近似できるから，これを変形して，$u_{n+1} = u_n + \omega_c v_n \Delta t$ となる。v_{n+1} も同様に変形すれば，オイラー法とは次の連立の漸化式を使って (u_n, v_n) から (u_{n+1}, v_{n+1}) を計算することである。

$$u_{n+1} = u_n + \omega_c v_n \Delta t$$
$$v_{n+1} = v_n - \omega_c u_n \Delta t$$

1 周期 $\dfrac{2\pi}{\omega_c}$ を適当な回数 N で分割して Δt を決定し，初期条件 $u_0 = 1$, $v_0 = 0$ から出発して，オイラー法で N 回計算するようなプログラムを作成せよ。そして，その結果得られる 1 周期後の速度，u_N，v_N と u_0，v_0 を比較せよ。また，10 周期後の速度，u_{10N}，v_{10N} とも比較せよ。

次に，オイラー法を少し修正したシンプレクティック法，(v_{n+1} の式に注意！)

$$u_{n+1} = u_n + \omega_c v_n \Delta t$$
$$v_{n+1} = v_n - \omega_c u_{n+1} \Delta t$$

のプログラムを作成し，オイラー法の結果と比較せよ。

なお，以上のプログラムは 配列を使わないで作成すること。

(2–5)　1 次元移流方程式

次の関数 $f(x,t)$ に関する 1 次元移流方程式を，CIP 法を用いて解析せよ。

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} = 0$$

但し，CIP 法とは，1 回の時間ステップを Δt として，格子点 i 上で定義された，n ステップ目の関数値 f_i^n とその x 偏微分 $g(x,t) = \dfrac{\partial f}{\partial x}$ の値 g_i^n が与えられているとき，$n+1$ ステップ目の関数値を

$$f_i^{n+1} = a_i X^3 + b_i X^2 + g_i^n X + f_i^n$$
$$g_i^{n+1} = 3a_i X^2 + 2b_i X + g_i^n$$

で近似する方法である。ここで，$X = -u\Delta t$ である。また，a_i，b_i は次式で与えられる。

$$a_i = \frac{g_i^n + g_{i+1}^n}{D^2} + \frac{2(f_i^n - f_{i+1}^n)}{D^3}$$
$$b_i = \frac{3(f_{i+1}^n - f_i^n)}{D^2} - \frac{2g_i^n + g_{i+1}^n}{D}$$

ただし，D は格子点間隔 Δx である。初期条件は，グリッド数 `imax` と，その中間の値，i_1，i_2 を適当に決めて，

$$\begin{cases} f_i^0 = 1 & (i_1 < i < i_2) \\ f_i^0 = 0 & (\text{それ以外}) \\ u = -1, \ \Delta x = 1 & \end{cases}$$

とせよ。

最初に，n ステップ目の格子点データ f_i^n と g_i^n を 1 次元配列 `f1(i)` と `g1(i)` で表し，これらを用いて，$n+1$ ステップ目の格子点データ f_i^{n+1} と g_i^{n+1} を表す 1 次元配列 `f2(i)` と `g2(i)` を計算するプログラムを作成する。そして，これらを適宜交換しながら繰り返すことで時間発展をするプログラムを作成すればよい。

なお，配列要素番号の最大値 `imax` では a_i，b_i を計算することができないので，`f2(imax)=0`，`g2(imax)=0` で良い。

第 2 章　サブルーチン

第 1 章で説明した基本的な Fortran 文法を使えば，原理的にはどんなプログラムでも作成可能です。そもそもプログラムとは計算機に仕事をさせるための動作手順を記述するものですから，それほどバラエティはありません。繰り返し手順を書くための do 文や条件分岐の if 文があればそれだけで計算機の能力のほとんどを使いこなすことができます。

しかし，計算機シミュレーションのように複雑な現象の記述が入り交じった長いプログラムをメインプログラムだけで書くと，見通しが悪くなって修正するのに手間がかかったりエラーが起こりやすくなります。そこで役に立つのがサブルーチンです。サブルーチンはメインプログラムと同じレベルの完結したプログラムのことですが，違いはメインプログラムがプログラム起動時に動作を開始するのに対して，サブルーチンはメインプログラムやその他のサブルーチンから呼び出されて初めて動作することです。本章ではこのサブルーチンの作り方と使い方について述べます。

2.1　サブルーチンを使う目的

「サブルーチン」の「ルーチン」とは，それ自体で完結した一連のプログラム手順を示す用語であり，サブルーチンに対応してメインプログラムのことを「メインルーチン」とも呼ぶ[11]。それぞれのルーチンは独立しているので，ルーチン毎にファイルを作成して別々にコンパイルし，必要に応じて結合 (リンク) することも可能である。また，プログラムだけでなくルーチン内で宣言した変数も独立しているため，ルーチン間でデータのやりとりをするには，所定の手続きが必要である。起動時に動作を開始するメインプログラムと違って，サブルーチンは単独で動作することはできない。このため，他のプログラムから呼び出されて初めてその機能を発揮する。

サブルーチンを利用する主な目的は 3 つある。

(1) プログラム中で同じ計算手順を複数の場所で使用するため
(2) 方程式の解法や行列式の計算などのような汎用性のあるプログラムを作るため
(3) 長いプログラムを構成要素毎に分割してプログラムの見通しを良くするため

(1) がサブルーチンの基本的な使用目的である。ある一連の手順があってそれを複数の場所で使う時，その手順が簡単ならばそれぞれで同じプログラムを書けばいいのだが，複雑になるとプログラムが長くなるし修正をする時に手間がかかる。こういう時にはその部分をサブルーチンにして手間を省くのである。

(2) は (1) を発展させたものと考えられる。よく使う公式や数値計算アルゴリズムをサブルーチンにしておけば，新しいプログラムを作る度に作り直す必要がなくなる。この場合，完成したサブルーチンはその機能だけが必要なので，できるだけブラックボックス化して数値の受け渡し部だけ公開するようにするのが望ましい。

しかし，シミュレーションプログラムにおいては，(3) の目的でサブルーチンを作成することが多い。シミュレーションに含まれる様々な物理過程の計算や入出力に関する作業などを機能ごとに分割してサブルーチンにするのである。このため，メインプログラムには分割したサブルーチンを呼び出す文と，必要ならそれらを繰り返すための do 文だけ書いておけばよい。シミュレーションプログラムは，サブルーチンという部品を使って組み立てるものだと考えればいいだろう。

サブルーチンに分割しておけば，プログラムのメンテナンスも楽である。なぜなら，それぞれのルーチンが独立しているので，プログラムを修正した時の影響やエラーが発生する原因がルーチン内に限定されるからである。シミュレーションプログラムというのは一度に完成するのではなく，使っているうちに改良や機能追加をして徐々に長くなるのが普通である。新しい機能を付け加える必要が出てきたら新しいサブルーチンを作るという考えで良いだろう。

[11] メインプログラムやサブルーチンのように，それぞれが完結しているプログラムを「プログラム単位」という。ルーチンだけでなく，2.6 で述べる module などもプログラム単位である。

2.2 サブルーチンの宣言と呼び出し

サブルーチンは subroutine 文で開始を宣言し，end 文で終了する。すなわち，次のような構造を持つ。

```
subroutine subr1
  implicit none
  real a,b
  integer i
    .......
    .......
end subroutine subr1
```

subroutine の後に指定した文字列 (この例では subr1) を「サブルーチン名」と呼ぶ。サブルーチンの最後を示す end 文には，subroutine という単語とサブルーチン名を指定する。すなわち，program 文と同じ構造である。サブルーチン内のプログラムの書き方も基本的にはメインプログラムと同じで，subroutine 文の次に implicit none を入れ，非実行文を上方に集約して，その後に実行文を書く。サブルーチンは独立した存在なので，実行文中で使う変数や配列は基本的にその内部で宣言しなければならない[12]。

上記のサブルーチンのように，subroutine 文にサブルーチン名しかないものを「引数なしサブルーチン」という。これに対し，引数ありのサブルーチンとは，以下のようにサブルーチン名の後に変数のリストをかっこで付加したものを言う。

```
subroutine subr2(x,y,n,m)
  implicit none
  real x,y
  integer n,m
    .......
    .......
end subroutine subr2
```

この例では，x,y,n,m が「引数」である。引数はサブルーチンとそのサブルーチンを使うルーチンの間で数値の受け渡しをするのに用いる。引数も変数であるから，それぞれの型に応じた宣言をしなければならない。

サブルーチンの機能を使うには，他のルーチンの中でそのサブルーチンを指名する。これにより，計算機の動作は指名をしたルーチンからサブルーチンに移る。そしてサブルーチン内に書かれた手順を実行して動作が完了すると，元のルーチンに動作が戻る。

サブルーチンを指名するには call 文を用いる。このため，サブルーチンを指名することを,「サブルーチンを呼び出す」とか「コールする」という。例えば，次のようにコールする。

```
real z
integer m
z = 200
call subr1          ........................  (1)
m = 21
call subr2(10.0,z**2,100,m*5+1)          .....  (2)
```

引数なしサブルーチンは (1) のように名前だけをコールする。これに対し，引数ありサブルーチンは (2) のように引数に応じた型の数値を引数と同じ順番で並べてコールする。この例のように，引数の場所に計算式を使ってもよい。その場合には計算結果が引数に与えられる。

サブルーチンは単独では意味をなさないので，メインプログラムが別途必要である。例えば次の様なセットを構成して初めて一つのプログラムが完成する。

[12] 2.6 で述べる module 中で宣言するグローバル変数は別である。

```
program stest1
   implicit none
   real x,y
   x = 5
   y = 100
   call subr(x,y,10)
   print *,x,y
end program stest1

subroutine subr(x,y,n)
   implicit none
   real x,y
   integer n
   x = n
   y = y*x
end subroutine subr
```

ここで，メインプログラムを先に，サブルーチンを後に書いたが，順序はどちらでも良い。サブルーチンが複数存在する場合でも，ルーチンを記述する順番は実行結果とは無関係である。

プログラムの実行の流れを図 2.1 に示す。すなわち，call 文で呼び出すとサブルーチンの一番最初の実行文に動作が移る。そして，サブルーチン内の動作が全て完了したら呼び出した側に戻って，call 文の次の文を実行する。

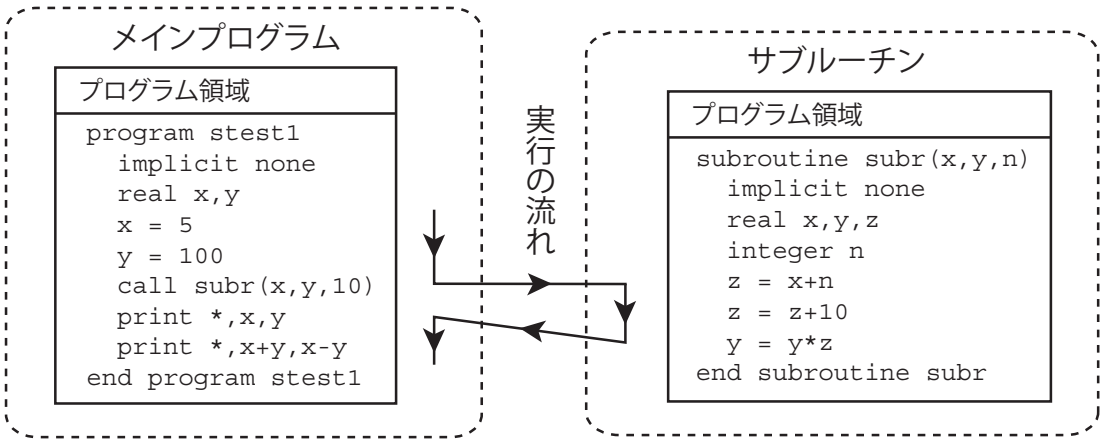

図 2.1　サブルーチンの呼び出しと実行の流れ

もし条件に応じて途中でサブルーチンの処理を打ち切って戻るときには return 文を用いる。

```
subroutine subr(x,y,n,m)
  implicit none
  real x,y
  integer n,m
    .......
  if (m<0) return
    .......
end subroutine subr
```

この例では，m<0 の時にサブルーチンの処理を終了し，呼び出したルーチンに戻る。ここで return 文の代わりに stop 文を用いると，プログラムそのものが終了する。

2.3 サブルーチン内の変数と引数

サブルーチン内で宣言した変数や配列は，メインプログラムや他のサブルーチンから独立している。すなわち，同じ名前を使っても全く別の変数である。

例えば，

```
program stest1
  implicit none
  real x,y
  x = 10
  y = 30
  call subr
end program stest1

subroutine subr
  implicit none
  real x,y
  print *,x,y
end subroutine subr
```

と書いても，サブルーチンの print 文によって出力される x と y はメインプログラムで代入した 10 と 30 ではなく，まるきり無関係な数字となる (このサブルーチンは無意味である)。逆に言えば，他のルーチンでの宣言を気にしないで変数や配列の名前を決めることができる。こういう，ルーチン内でのみ有効な変数を「ローカル変数」という。

上記のプログラムを期待通りに働かせるには引数を使う。引数は，関数 $f(x)$ における独立変数 x と同じような役割をする。すなわち，上記のプログラムを期待通り働かせるには，

```
program stest2
  implicit none
  real x,y
  x = 10
  y = 30
  call subr(x,y)
end program stest2

subroutine subr(x,y)
  implicit none
  real x,y
  print *,x,y
end subroutine subr
```

と書く。なお，この例では call 側の引数と subroutine 側の引数の変数名を同じにしたが，必ずしも同じである必要はない。次のようなサブルーチンに置きかえても全く同じ動作をする。

```
subroutine subr(a,b)
  implicit none
  real a,b
  print *,a,b
end subroutine subr
```

外部からの影響を受けたり，外部に影響を与える，という特徴を持つことを除けば，引数もローカルな変数なのである。関数 $f(x)$ の変数 x に値を与える時に，$f(10)$ と数字を書いたり，$f(a)$ と異なる文字を書いたり，$f(y^3)$ と数式を書いたりしても良いのと同じと考えればよい。

引数を持つサブルーチンを作る時に重要なポイントは，

(1) 引数の数
(2) 対応する引数の型

が call 文側と subroutine 文側で一致していなければならないことである。

例えば,

```
program stest3
  implicit none
  real z
  integer m
  z = 200
  m = 21
  call subr(10.0,z**2,100,m*5+1)
end program stest3

subroutine subr(x,y,n,m)
  real x,y
  integer n,m
  print *,x,y,n,m
end subroutine subr
```

というプログラムでは,

call 文側	subroutine 文側	変数型
10.0	x	実数
z**2	y	実数
100	n	整数
m*5+1	m	整数

という対応になっている。特に注意して欲しいのは第 1 引数の x は実数型なので実定数 10.0 を与え，第 3 引数の n は整数型なので 100 という整定数を与えていることである。もし，第 1 引数に，同じ意味だろうと思って 10 という整定数を与えると，エラーになる。

サブルーチン側で，引数の変数に数値を代入すると，call 側で引数に与えた変数に代入される。

例えば,

```
program stest4
  implicit none
  real x,y,p
  x = 10
  y = 30
  call subr(x+y,20.0,p)
  print *,x,y,p
end program stest4

subroutine subr(x,y,z)
  implicit none
  real x,y,z
  z = x*y
end subroutine subr
```

と書くと，subroutine 内の x は program 側の x+y，すなわち 40 であり，subroutine 内の y は program 側の 20.0 なので，subroutine 内の z は 40×20=800 となる。この時，z が引数なので，program 内の変数 p にも 800 が代入されて，call 文の次の print 文では x，y，p としてそれぞれ 10,30,800 が出力される。

このようにサブルーチン側で引数に値を代入することで call 側の変数に値を代入するものを「戻り値」という。引数は，call 側→ subroutine 側という一方通行なのではなく，subroutine 側→ call 側へのリターンも可能なのである。

ただし，call 側での引数の与え方で，定数を与えたり計算式を書いてもよい，という話をしたが，このような戻り値を指定する引数は別である。必ず変数にしなければならない。理由を次節で説明しよう。

2.4 ルーチン間のデータの引き渡しと間接アドレス

ここでは，ルーチン間のつながりをもう少し詳しく考えてみよう。各ルーチンは基本的に独立した構造をしていて，プログラム領域とデータ領域から構成されている。プログラム領域とは文字通りプログラムの手続きが書いてある領域のことであり，データ領域とはルーチン内で使われている変数や配列のために用意されたメモリ領域のことである。プログラム的には，変数や配列の宣言文がデータ領域の指定を意味する。

変数や配列は各ルーチン毎に宣言しなければならないことからわかるように，データ領域は各ルーチンそれぞれに付随して存在している。このため，異なるルーチンで同じ名前の宣言をしても別の変数として扱われるのである (図 2.2)。これがローカル変数である。

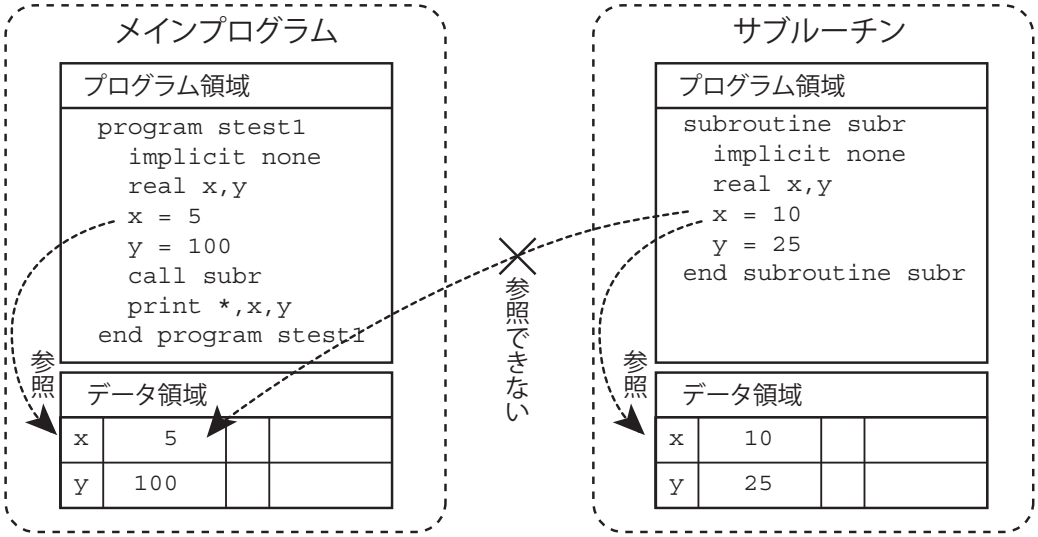

図 2.2 プログラム領域とデータ領域

このため，あるルーチンで計算した結果を別のルーチンで使うには特別な手続きが必要となる。これが「引数」である。引数というのは，サブルーチンを呼び出す側 (call 側) からデータをサブルーチンに伝える手段である。さて，それでは引数は具体的にどうやってデータをやりとりしているのであろうか？

もっとも単純に考えられるのは，引数の数値を直接サブルーチンに渡す方法である。図 2.3 を見てみよう。

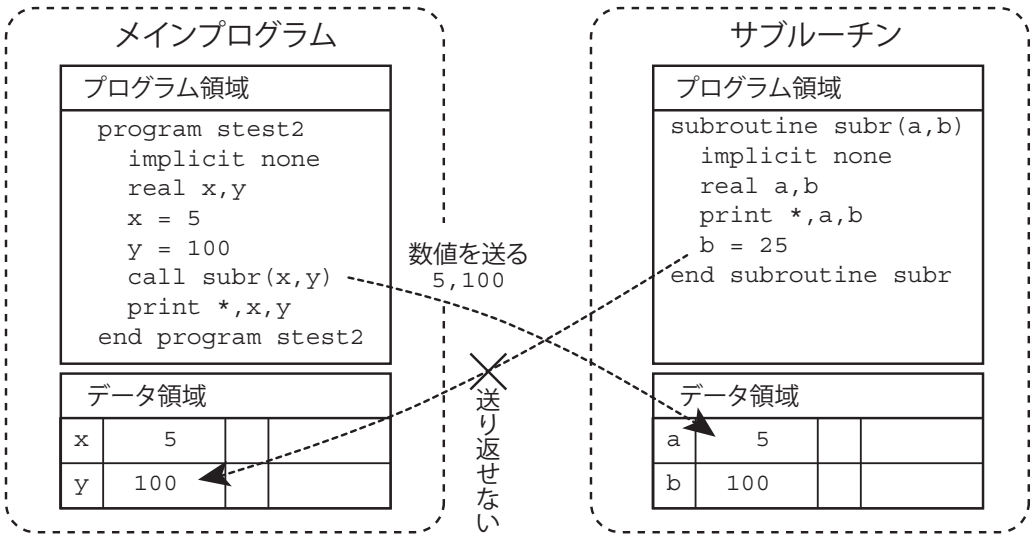

図 2.3 データの直接受け渡し (call by value)

図のように，call 側の引数 x と y の内容 (この例では 5 と 100) が，サブルーチンの引数として宣言された変数，a と b に代入されている。この方法は，引数の値を直接引き渡す，という意味で「call by value」と呼ばれている。

この方法は，呼び出し側→サブルーチン側への値の引き渡しについては何の問題もない。実際，C 言語はこの方法をとっている。

しかし，逆，すなわちサブルーチン側で作成したデータを call 側に戻すことはできない。サブルーチンは，call 側から送られてきた値，この例では 5 と 100，はわかるのだが，それ以上の情報がないので call 側のどこにデータを戻せばいいのかわからないからである。図 2.3 のように，サブルーチン側の変数 b に値を入れても (b=25)，それはあくまでも，サブルーチンのデータ領域の b に代入するだけなので，メインプログラム側には一切影響がない。

この問題を解決するために，Fortran は call by value ではなく，call by reference という方法を採用している。call by reference とは，変数の内容ではなく，変数の存在場所 (アドレス) をサブルーチンに伝える方法である (図 2.4)。

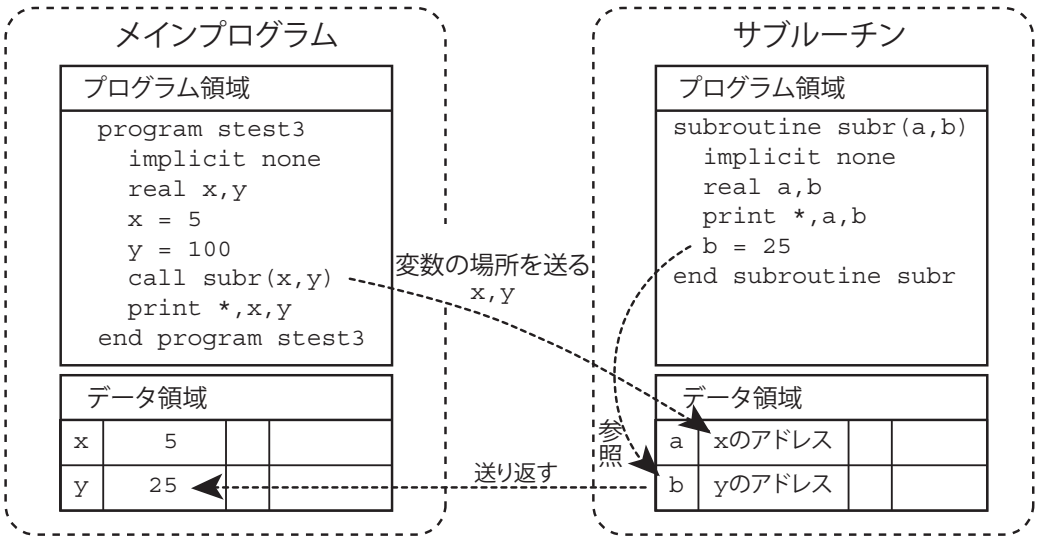

図 2.4　データの間接受け渡し (call by reference)

コンピュータには指定したアドレスのメモリから直接データを読み書きする方法の他に，指定したアドレスのメモリに入っている「メモリアドレスデータ」を使ってそのアドレスのメモリから間接的に読み書きする方法がある。これを間接アドレスという。Fortran のサブルーチンの引数は，この間接アドレスを使って call 側にデータを返すことができるのである。

ハードウェア的にメモリには「番地」と呼ばれる番号が付いている。アドレスとはこの番地のことである。メモリデータを参照するときはそのメモリ番地を指定する。例えば，簡単な代入文

```
a = 15000
b = a
```

を考えよう。もちろん結果は b=15000 であるが，代入式 b=a において，a（番地が [104] とする）という変数からデータを取って来て b に入れるという流れは

```
a の番地指定　　→　　a 番地 [104] のメモリ内のデータ（15000）の呼び出し
　　→　　b に代入
```

となる。流れを図に描けば図 2.5 のようになる。この方式を「直接アドレス」と言う。

これに対し，指定した番地のメモリ内に入っているのが，計算のための数値ではなく，別の場所のメモリ番地を表す数値が入っていて，その番地データが指定するメモリの中に入っている数値を参照することを「間接アドレス」と言う。

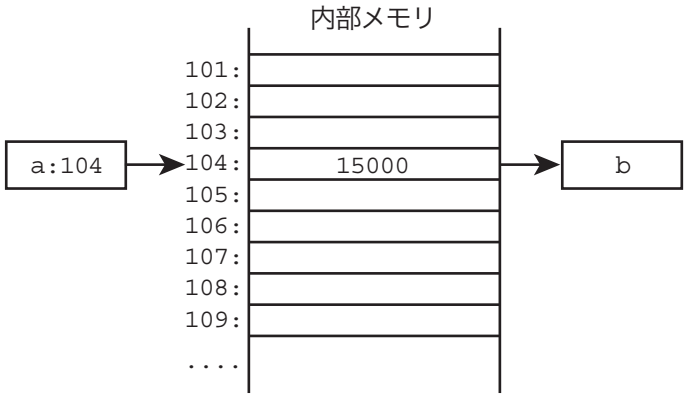

図 2.5　直接アドレスによるメモリ参照

ややこしい方法であるが，Fortran におけるサブルーチンの引数は，この間接アドレスなのである。例えば，

```
program stest2
  implicit none
  real a
  a = 15000
  call subr(a)
end program stest2

subroutine subr(b)
  implicit none
  real b,c
  c = b
end subroutine subr
```

と書くと，サブルーチン subr の中の代入文 c=b によってやはり，c=15000 になるが，この場合，b は a の内容ではなく，a の番地データが入った変数なのである。そして c=b という代入文で変数 b（番地が [109] であるとする）からデータを参照するのが間接アドレスである。流れとしては

```
b の番地指定      →    b 番地 [109] にあるメモリの番地データ（a:104）の呼び出し
                  →    a 番地 [104] にあるデータ（15000）の呼び出し
                  →    c に代入
```

となる。図示すれば図 2.6 のようになる。

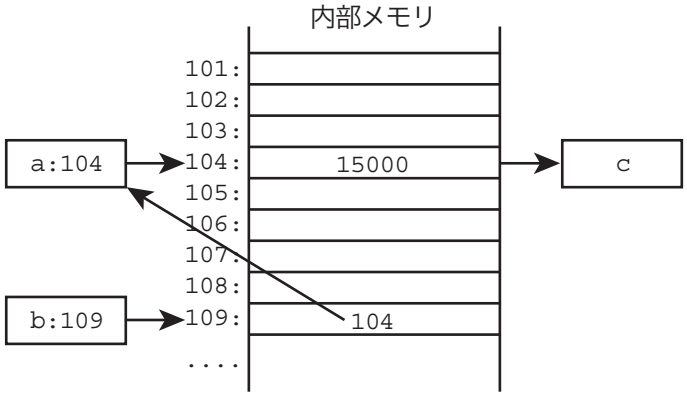

図 2.6　間接アドレスによるメモリ参照

引数が間接アドレスであることの意義は，この逆の動作にある。つまり，前節で述べたように引数の変数に数値を代入すると，そのデータが call 側に影響する。例えば

```
program stest3
  implicit none
  real a
  call sub(a)
  print *,a      ..... (1)
end program stest3

subroutine sub(x)
  implicit none
  real x
  x = 10.0
end subroutine sum
```

とすると，サブルーチン内で引数 x に代入された値 10.0 は，間接アドレスを経由してメインプログラムの変数 a に代入される。この結果，(1) の print 文で 10.0 が出力されるのである。

このように，戻り値を使う時は対応する call 側の引数に値を代入するアドレスを与えなければならない。これが変数または配列を指定しなければならない理由である。定数や計算式では実行時エラーとなる。また，変数の型も合っていなければならない。

2.5 配列を引数にする場合

配列を引数にするときは，配列名を引数にすることも，配列要素を引数にすることもできる。配列名を引数にしたときはその配列の第 1 要素を引数にしたことと同じ意味を持つ。即ち，a という 1 次元配列に対して，

```
call sub(a)   と   call sub(a(1))
```

は同じ動作をする。

これを受けるサブルーチン側の宣言はサブルーチンでそのデータをどう使うかで決める。例えば次の 2 種類が考えられる。

Case A

```
subroutine sub(x)
  implicit none
  real x
  x = 10.0
  ......
end subroutine sub
```

Case A では引数 x の配列宣言をしていない。このため，call 文で指定した a(1) だけしかサブルーチン内では利用できない。これは，call sub(a) という，配列名を与えた場合でも同じである。

Case B

```
subroutine sub(x)
  implicit none
  real x(10)
  integer i
  do i = 1, 10
      x(i) = i
  enddo
  ......
end subroutine sub
```

これに対し，Case B の場合には a という配列が x という名に変わっただけで，あとは同じように使用できる。このとき，サブルーチン側の配列宣言 x(10) の要素数 10 は call 側と同じ数にする必要はない。これは 2.4 で述べたように「引数は間接アドレス」だからである。サブルーチンの引数はあくまでもアドレスを示す変数に過ぎないのだが，このことは単変数でも配列でも同じである。配列はメモリ上で並んでいることがわかっているので，その先頭アドレスさえサブルーチンに引き渡せば，それ以外のデータは「先頭アドレスから何番目」と参照する

ことができる。よって，サブルーチン内で必要な情報は配列の次元と変数の型だけである。

一般的に，サブルーチン内で宣言するときの引数配列は必ずしも call 側ルーチンの宣言と要素数が一致している必要はなく，それどころか次元さえ一致している必要はない。このため，明示する必要がない要素は数字ではなく「*」を使って宣言することができる。

例として，要素数 n の配列 a のデータを同じ長さの配列 b にコピーするプログラムを考えよう。最も単純な答えは，

```
subroutine copy(a,b,n)
  implicit none
  real a(*),b(*)
  integer n,i
  do i = 1, n
    b(i) = a(i)
  enddo
end subroutine copy
```

である。つまり，サブルーチン側の配列宣言は「*」でよい。もちろん，コピーする要素数 n が元の配列の宣言範囲を越えていれば動作は保証されないが，これは「使う側の責任」である。さらに，このプログラムは a(10,10) の様な 2 次元配列でも使用できる。なぜなら 1.3.2 で述べたように，2 次元配列もメモリ上では 1 次元的に並んでいるからである。ただし，a(10,10) なら要素数 n には 10×10=100 を指定する必要がある。

ただし,「*」を 2 次元以上の配列に対して用いるときは注意が必要である。例えば，a(*,*) という形の宣言はできない。なぜなら，2 番目の要素を進めるのは 1 番目の要素が最大値，つまり宣言した上限数に達したときだから，1 番目の要素が不定では情報不足だからである。よって，1.3.2 で述べたような，配列の並び番号に影響しない最後の要素数のみ「*」にすることが可能である。例えば，a(3,*) と宣言すると，第 1 要素の数が 3 の 2 次元配列として計算される。

2 次元配列に対し，要素数が何であっても自由に対応できるようにする時は「整合配列」を用いる。整合配列とは引数の中の整変数で引数配列を宣言することである。

```
subroutine copy2d(a,b,n,m)
  implicit none
  real a(n,m),b(n,m)
  integer n,m,i,j
  do j = 1, m
    do i = 1, n
      b(i,j) = a(i,j)
    enddo
  enddo
end subroutine copy2d
```

上例は，a，b にどんな要素数の 2 次元配列を代入しても，n，m の指定をその call 側での配列宣言通りに指定すれば使える。この整合配列という機能は，Fortran の優れた文法の一つである。

なお，配列宣言の際にはコロンを付加することで下限を指定することができるが (1.3.1 参照)，これをサブルーチンに引き継ぐにはサブルーチン側でも同様の下限を指定する必要がある。

```
program stest4
  implicit none
  real a(0:100)
  call sub(a,100)
end program stest4

subroutine sub(x,n)
  implicit none
  real x(0:*)
  integer n
  x(10) = 10.0
end subroutine sum
```

この例のようにサブルーチン側でもメインプログラムと同じ下限指定をすると，x(10) に代入された値がメインプログラムの a(10) に代入されるが，サブルーチンの宣言を real x(*) とすると，下限が 1 になるので，a(9) に値が代入される。ただしこれは必ずしもエラーではない。サブルーチン使用者が意識して使えば良いことである。

2.6 module と use 文

2.3 で，サブルーチン内で宣言した変数は，その内部でしか意味を持たない「ローカルな」変数であるとの話をした。ローカル変数の反対を「グローバル変数」という。グローバル変数とは，メインプログラムやどのサブルーチンから参照しても同じ値を持つ変数のことである。Fortran では，module と use 文の組み合わせでグローバル変数を実現している[13]。

module とは，言ってみれば，変数の集合を一つのプログラム単位にしたものである。つまり，メインプログラムやサブルーチンなどのルーチンと同じレベルである。

```
module data1
   integer m,n
   real t0,a(20)
end module data1
```

このように，先頭が module とその名前（この例では data1），最後が end module 文となる。サブルーチンと同じように implicit none 文を先頭に書くこともできるが，変数宣言だけならば必ずしも書く必要はない。

この module 中の変数を使いたい時には，使いたいルーチンの先頭に use 文で module 名を指定する。use 文は implicit 文よりも前に書かなければならない。一例として次のようになる。

```
module global
   real x,y
end module global

program stest5
   use global
   implicit none
   x = 5
   y = 100
   call subr
   print *,x,y
end program stest5

subroutine subr
   use global
   implicit none
   print *,x,y
   y = 25
end subroutine subr
```

この例に示したように module はメインプログラムおよび全てのサブルーチンより前に書かねばならない。このプログラムのイメージを図 2.7 に示す。

module を使う欠点としては，各ルーチン内で陽に宣言されていない変数が存在することになるので，プログラムのチェックに若干手間がかかることである。これを軽減するために，グローバル変数はできるだけ意味のある長めの名前を付けるのが良い。そうすれば検索もしやすいし，タイプミスチェックもしやすい。

module はさらにサブルーチンを含めることも可能で，その内蔵サブルーチンは単独で存在するサブルーチンにはない引数配列のチェック機能や引数の型に応じて異なるサブルーチンを起動するオーバーロード機能など，様々な拡張機能を持っている。

[13] module は Fortran90 から採用された比較的新しい仕様である。Fortran77 以前では common 文でグローバル変数を実現していた。しかし，common 文は不便な点が多いので本書では省略する。

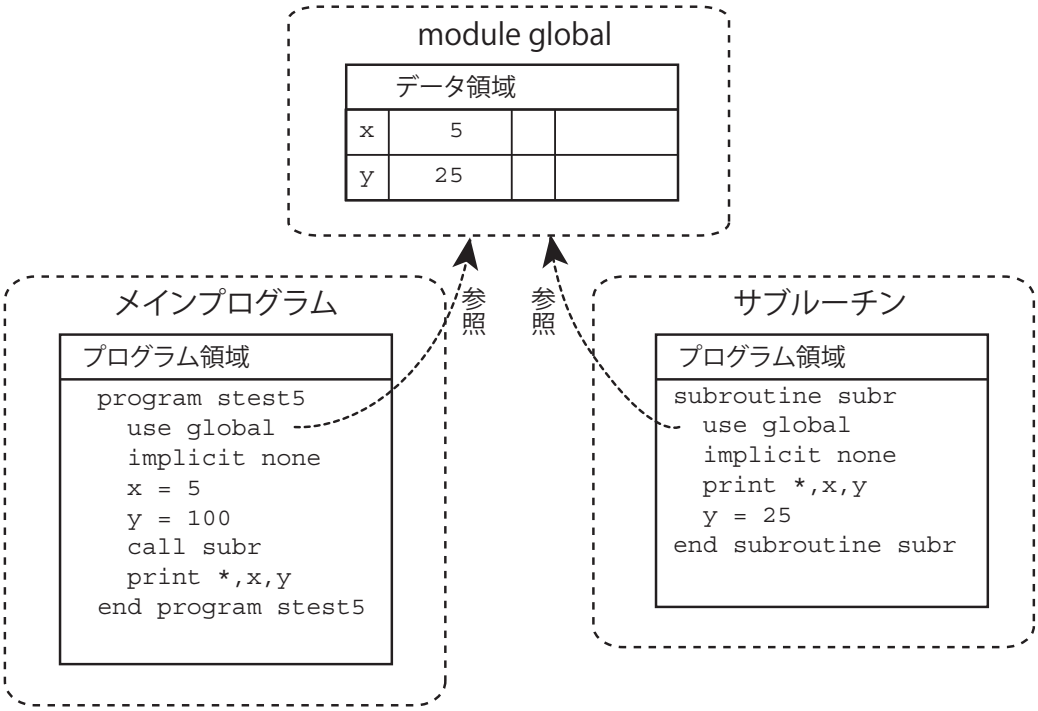

図 2.7　module を使ったデータの受け渡し

2.7　関数副プログラム

sin(x) のような数学関数を自作することもできる。これを関数副プログラムという。関数副プログラムはサブルーチンとほとんど同じ構造をしているが，引数以外に 1 個の戻り値を持たせることができるところが異なる。この戻り値が関数値となるので，計算式中に直接使用することができる。

例えば，与えられた実数 x の二乗を計算する関数副プログラムは以下のようになる。

```
function square(x)
  implicit none
  real square,x
  square = x*x
end function square
```

この例のように subroutine の代わりに function にすると関数副プログラムになる。サブルーチンと違うのは，関数名（上例では square）を戻り値変数として値を代入しなければならないことである。このため，関数名も変数のように型宣言をする必要がある。

関数副プログラムの「関数名」も変数の一種であるため，関数副プログラムを使う時も関数名の型宣言が必要である。上例の square を使うには，

```
program ftest1
   implicit none
   real x,y,square
   x = 5
   y = square(x) + x**3
   print *,x,y
end program ftest1
```

のように square という関数名を関数副プログラムと同じ型宣言にする。function 文での関数名の型と，その関数を使用するルーチンでの関数名の型が異なるとエラーになる。

ただし，次節で述べる組み込み関数については関数名の型宣言をする必要はない。

2.8 組み込み関数

1.2.4 で説明したように，sin や sqrt のように数値計算上よく使う関数はあらかじめ用意されている。これを「組み込み関数」という。組み込み関数には，引数の型に応じて計算を変える「総称名」機能があるので，関数副プログラムのような型宣言は不要である。数学関数以外で数値計算上必要と思われるものには，この他に型変換や余りの計算などがある。

複素数型を使ったプログラムを作成する時は，実数と複素数との型変換が必要となることがある。そのいくつかを次表に示す。ここで，z は複素数型，x，y は実数型とする。

複素数変換関数

組み込み関数	名　称	結果の数値型	数学的表現
real(z)	複素数の実数部	実数型	$\mathrm{Re}(z)$
imag(z)	複素数の虚数部	実数型	$\mathrm{Im}(z)$
cmplx(x,y)	複素数化	複素数型	$x + yi$
conjg(z)	共役複素数	複素数型	$\bar{z}$

ただし，複素数型の数 z を実数型の変数 x に x=z と代入すれば，z の実数部が x に入るので，real(z) は必ずしも必要ではない。

整数型の関数は使う機会が少ないが，意外によく使うのが「余り」を計算する関数 mod である。

整数関数

組み込み関数	名　称	結果の数値型	数学的表現
mod(m,n)	剰余	整数型	m/n の余り

どこで使うかというと，do 文で繰り返しの計算をするときに「n 回毎に出力する」という時である。

例えば，シミュレーションで繰り返し計算をするときに，10 回に 1 回出力するには

```
do n = 0, 10000
......
  if (mod(n,10) == 0) then
     print *,x,y,z
  endif
......
enddo
```

のように書く。

第 3 章　データの入出力

シミュレーションプログラムは計算をするだけではだめで，計算結果を表示したり保存したりしなければなりません。これを「出力する」と言います。これまでの例題中に出てきた print 文は画面に結果の数値を出力するための文で，取りあえずはこれで充分です。しかし，出力データが増えると画面に出力することが無意味になるので，その時はファイルに出力します。ファイルに出力しておけば，プログラム実行後にその出力ファイルを使って別のソフトでデータ解析をすることも可能です。

また，print 文を使って実数を標準形式で出力すると内部表現の有効数字 (15 桁程度) で表示するので，あまり多くの数値を横に並べることができません。このようなときには出力形式を指定して，桁数を短くしたり数字をそろえて出力します。

プログラムが完成してくると，計算の初期値を変更して再コンパイルするのが煩わしくなってきます。そこで，必要な数値をプログラムの実行中に与える「入力する」方法も知っておくと便利です。

本章では，ファイルへの出力方法や出力形式の指定，および入力文について述べます。

3.1　データ出力の各種方法

3.1.1　出力文の一般型

もっとも単純な出力文は，print 文である。

```
print form, 数値 1 , 数値 2   ...
```

print 文で出力すると，画面に数値が出力される[14]。*form* には出力形式を指定するのだが,「*」を与えておけば，それぞれの数値の型に応じた標準形式で出力をしてくれる。

```
print *, x, y(i), x**2+5, 10
```

この例のように「数値」の位置には，変数や配列を与えても良いし，計算式や定数でも良い。ただし，標準出力形式では実数が 15 桁程度の有効数字で出力されるので，あまりたくさん横に並べることはできない。出力する数値の桁数を減らすには，*form* で出力形式の指定を行う (3.1.3 参照)。

画面ではなく，ファイルに出力するときには write 文を用いる。print 文との違いは，出力先と *form* をかっこで指定することである。

```
write(nw, form) 数値 1 , 数値 2   ...
```

nw には整数を与え，この数値で出力先を指定する。*nw* を装置番号という。なお，$nw \geq 10$ にすること[15]。

nw に適当な整数を与えて実行すると,「fort.*nw*」という名のファイルに出力される[16]。

例えば,

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

と書けば，x,y,z の値が fort.20 という名前のファイルに出力される。print 文と同じく，*form* に「*」を与えれば，標準形式で出力する。ただし，標準出力では 1 行の出力文字数に制限があるので，出力数が制限を超えると自動的に改行されてしまう。このため，幅広く出力する時も出力形式 *form* の指定が必要である。

---

[14] 正確には標準出力に出力される。

[15] 6 以下は予約されているので使えない。例えば 6 を指定すると print 文と同じ標準出力になる。それ以外の一桁の数字は使っても間違いではないのだが，数字をそろえるという意味でも二桁の整数にしておいた方が良いだろう。

[16] 「fort」の部分はコンパイラに依存するが fort が多いようである。

3.1.2 配列の出力，do 型出力

配列は「数値」の位置に配列名を書けば出力される。

```
real a(5)
do i = 1, 5
   a(i) = i
enddo
print *,a
```

この場合，出力は a(1),a(2),...,a(5) のように，配列要素の順に横に出力される。

しかし，要素数が小さい時は良いのだが，要素数が大きくなったり，多次元配列の時には不便である。

例えば，2 次元配列を

```
real a(3,3)
do j = 1, 3
   do i = 1, 3
      a(i,j) = i*j
   enddo
enddo
print *,a
```

のように出力すると，a(1,1),a(2,1),a(3,1),a(1,2),...,a(3,3) という具合に 9 個の要素が横に出力される。

そこで，次のように do 文で出力する方がよい。

```
real a(3,3)
do j = 1, 3
   do i = 1, 3
      a(i,j) = i*j
   enddo
enddo
do j = 1, 3
   do i = 1, 3
      print *,a(i,j)
   enddo
enddo
```

しかし，この方法では 1 行に 1 数値なので，この例だと 9 行必要で，出力行が多くなる。そこで，1 行に行列の 1 行を出力すれば見やすいであろう。Fortran の print 文は，出力後に必ず改行をするので，複数の print 文で，横に続けて書くことはできない。このため，次のように書かなければならない。

```
real a(3,3)
do j = 1, 3
   do i = 1, 3
      a(i,j) = i*j
   enddo
enddo
do i = 1, 3
   print *,a(i,1),a(i,2),a(i,3)
enddo
```

しかし，このように行列の数がわかっている時は良いのだが，変数で指定したい時や，列数が多くなってくると不便である。この時は次のような do 型出力を使う。

```
real a(3,3)
do j = 1, 3
   do i = 1, 3
      a(i,j) = i*j
   enddo
enddo
do i = 1, 3
   print *,(a(i,j),j=1,3)
enddo
```

この例は，a(i,1),a(i,2),a(i,3) と書くことと同等である。つまり，do 文のように print 文の中に繰り返し文を書けるのである。

do 型出力は二重になっても良い。

```
real a(3,3)
do j = 1, 3
   do i = 1, 3
      a(i,j) = i*j
   enddo
enddo
print *,((a(i,j),j=1,3),i=1,3)
```

この場合，内側の繰り返し (この例では j=1,3) が先に進む。a(i,j) の部分には計算式を書いても良い[17]。

3.1.3 出力形式の指定（format）

標準の出力形式 (*指定) で実数を画面に出力すると，15 桁くらいの数字が出てきて，見にくいし，1 行あたりに出力できる数値が少ない。また，行毎に小数点の位置が違うので，大量に出力すると非常に見にくい。

この問題を解決するには出力形式 (format) を指定する。つまり，これまで「*」を指定していた *form* の部分に format を指定することで，小数点の桁数を指定したり，必要に応じて数字と数字の間にスペースを空けたり，改行を入れることができる。

format の指定方法は 2 通りある。まず，format 文による指定である。

```
real x,y
integer n
x = 1.5
y = 0.03
n = 100
print 600,x,y,n
  600 format('  x = ',f10.5,'   y = ',es12.5,'   n = ',i10)
```

format 文には，必ず先頭に文番号を付け，その文番号を print 文や write 文の *form* に指定する。この例では，600 が *form* 指定である。*form* の数字は文番号なので，一つのルーチン内では重複しないようにしなければならない。別の print 文が同じ format 文を指定するのはかまわない。次のように，write 文との併用も可能である。

```
real x,y,u,v
integer n,k
x = 1.5
y = 0.03
n = 100
print 600,x,y,n
write(20,600) u,v,k
  600 format('  x = ',f10.5,'   y = ',es12.5,'   n = ',i10)
```

format を指定するもう一つの方法は，*form* に文字列として format の内容を書くことである。例えば，上記の format 文の代わりに以下のような書き方ができる。

[17] この例では，print *,a と書いたのと同じ 9 個の要素が横に並んで出力されるが，出力の順序は異なる。なぜか考えてみよう。

```
real x,y
integer n
x = 1.5
y = 0.03
n = 100
print "(' x = ',f10.5,'   y = ',es12.5,'   n = ',i10)",x,y,n
```

このように，*form* のところに，「"」で囲んだ format 文の内容を指定するのである。この時，両端かっこも必要なので忘れないように。なお，Fortran では，文字列の指定は「'」で囲むのが普通だが，format 内部にも「'」が入っている時には「"」で指定する。

format を print 文の中に書き込む方法は，文番号を必要としない点は良いのだが，print 文が長くなるし，同じ format を何度も使う時には不便である。この時には，文字列変数を使うことができる (4.3.2 参照)。

3.1.4 format の記述方法

format は，コンマで区切った「編集記述子」の並びである。前節の format の意味を示す。

```
print "(' x = ',f10.5,'   y = ',es12.5,'   n = ',i10)",x,y,n
```

まず，format 中の文字列はそのまま出力される。この例では，' x = 'や' y = 'は，スペースも含めてそのまま出力される。

次に，print 文中の数値それぞれに対応して出力形式を指定する編集記述子を記述する。この例では，

```
x  →  f10.5
y  →  es12.5
n  →  i10
```

という対応で出力指定されていて，実行したときの出力は以下のようになる。

```
 x =    1.50000    y =  3.00000e-02    n =         100
```

出力形式を指定する編集記述子の主要なものを以下に示す。この表で，イタリック文字 w や m は整数 (10 とか 3 とか) を指定する。

基本的な編集記述子

編集指定	数値の型	編集の意味
Iw	整数	幅 w で整数を出力する
I$w.m$	整数	幅 w で整数を出力する 出力整数の桁が m より小さい時には，先頭に 0 を補う ($w \geq m$)
F$w.d$	実数	幅 w で実数を固定小数点形式で出力する d は小数点以下の桁数 ($w \geq d+3$)
E$w.d$	実数	幅 w で実数を浮動小数点形式で出力する d は小数点以下の桁数 ($w \geq d+8$) 仮数部の 1 桁目は 0 になる
ES$w.d$	実数	幅 w で実数を浮動小数点形式で出力する (d は E 編集と同じ) 0 以外の数値を出力すると，仮数部の 1 桁目は 1 から 9 となる
EN$w.d$	実数	幅 w で実数を浮動小数点形式で出力する (d は E 編集と同じ) 0 以外の数値を出力すると，仮数部の整数は 1000 未満となり， 指数部の数は 3 で割り切れる数となる

基本的な編集記述子 (続)

編集指定	数値の型	編集の意味
G$w.d$	実数	幅 w で実数を固定小数点または浮動小数点形式で出力する どちらになるかは，実数の指数部の大きさで決まる d は小数点以下の桁数
A	文字列	文字列をそれ自身の長さの幅で出力する
Aw	文字列	幅 w で文字列を出力する

ここでは，編集指定の文字 (F や ES など) を指定の数 (w など) と区別するために大文字で書いたが，小文字でかまわない。前出の例で，i10 は整数を幅 10 文字で出力することを意味し，f10.5 は実数を幅 10 文字，小数点以下 5 桁で出力することを意味している。

重要なことは，各編集記述子と出力の数値を 1 対 1 対応にすることである。また，数値の型に応じた編集記述子を使わなければならない。

例えば,

```
real x,y
integer n
x = 1.5
y = 0.03
n = 100
print "(f10.5,f10.5,i10)",x,y,n
```

というプログラムの出力は,

```
    1.50000   0.03000       100
+----+----+----+----+----+----+----+----+----+
```

となる。2 行目の目盛りは位置を確認するために書いたものであるが，10 文字の中に，右寄りで出力されているのがわかる。なお，指定の数が適切でなく，出力文字数が指定の幅 w を越えると,「*****」のように「*」が w 個出力される。

E 編集を使って実数を浮動小数点形式で出力すると，小数点の前は 0 になる。例えば,

```
real x,y
x = 1.5
y = 3.14e10
print "(e15.5,e15.5)",x,y
```

の出力結果は

```
    0.15000e+01    0.31400e+11
```

となる。これは感覚的にわかりにくいし，表示字数が 1 個無駄になるので，ES 編集や EN 編集を使う方が良いだろう。例えば,

```
real x
x = 3.14e10
print "(es15.5,en15.5)",x,x
```

の出力結果は

```
    3.14000e+10   31.40000e+09
```

となる。

同じ編集を並べるときには，編集記述子の前に整数 r を指定して反復指定をすることができる。

```
real x,y
integer n
x = 1.5
y = 0.03
n = 100
print "(2f10.5,i10)",x,y,n
```

すなわち，2f10.5 は f10.5,f10.5 と等価である。

また，実数，整数，実数，整数のような繰り返しの時には，かっこでくくって反復指定をすることができる。

```
real x,y
integer m,n
x = 1.5
y = 0.03
m = 5
n = 100
print "(2(f10.5,i10))",x,m,y,n
```

出力文中の数値に対応していない編集記述子もある。

出力文中の数値と無関係な編集記述子

編集指定	編集の意味
文字列	文字列を出力する
rX	r 文字スペースを挿入する
kP	以降の出力数を k 桁シフトする (k は負も可) F 編集の時は，数字が 10^k 倍になる E 編集の時は，数字は等しいが仮数部の整数部の桁が k 桁になる
/ または r/	改行する，r を付けると r 回改行する
:	出力文中の数値の出力が終わった時点で format 中の以後の出力を打ち切る

X 編集は数字と文字の間にスペースを開けたい時に使うが，スペースの文字列を与えても同じである。

P 編集は，例えば，実数をパーセントで表示したい時，F 編集の前で 2P を指定すれば，100 を掛けるという計算をせずに出力することができる。ただし，P 編集を指定すると，それ以降の全ての編集記述子に影響を与えるので注意しなければならない。

「:」編集はわかりにくいので例を使って説明しよう。もし，format 中の編集記述子の数よりも print 文や write 文の数値の方が少ない場合には，print 文や write 文で指定した数値を全て出力した時点で終了する。

例えば，

```
real a,b,c
a = 1.5
b = 0.03
c = 5
print "(10f12.5)",a,b,c
```

の場合，print 文で出力したいのは，a,b,c の 3 個だが，format では 10f12.5 と，10 個の固定小数点出力ができる記述になっている。この場合，print 文の 3 個が優先して

```
     1.50000      0.03000      5.00000
```

のように 3 個の数値が出力される。

この時，もしそれぞれの数字の頭に$を付けたくて，

```
real a,b,c
a = 30.25
b = 300.2
c = 50000.6
print "(10('   $',f10.2))",a,b,c
```

と書くと，出力は

```
   $      30.25   $     300.20   $  50000.60   $
```

となるのである。つまり，最後に余分な$が出力される。これは，数値と編集記述子が一致しなくなった段階で print 出力が終了するためである。これを防ぐのに「:」編集を使う。つまり，

```
real a,b,c
a = 30.25
b = 300.2
c = 50000.6
print "(10('   $',f10.2:))",a,b,c
```

のように，f10.2 の後に「:」を入れておけば，ここで終了するのである。この例では

```
   $      30.25   $     300.20   $  50000.60
```

と出力される。

なお，format 中の編集記述子の数よりも print 文や write 文の数値の方が多い場合には，編集記述子の指定数だけ出力した後，改行をして，残りの数値を再びその format で出力するという形式になる。

3.1.5 書式なし出力文

write 文は *form* を省略して装置番号のみの指定で出力することが可能である。これを「書式なし write 文」という。

```
write(nw) 数値 1，数値 2   ...
```

書式なし write 文の出力はデータ形式が「バイナリ形式」になる。これに対し，画面に文字で表示されるのは「テキスト形式」である。元来，計算機内部の数値は 2 進数データであり,「10」とか,「1.000e20」とか言った「文字」ではない。画面に表示する時は「2 進数→文字による数字表現」というデータ変換を行っているのである。

しかし，変換には時間がかかる上に，有効数字通りに文字に直すとデータ量が増える。そこで，大量のデータを保存する時には，文字に変換せず元の 2 進数データをそのまま保存する方がよい。このいわゆる「生データ」をバイナリ形式というのである。

ただし，Fortran の書式なし write 文を使って出力したバイナリ形式ファイルは，後述の書式なし read 文で読まなければならないので，あくまでも Fortran プログラムからしか利用できない。別のアプリケーションで使うためにファイルを作成するときは，書式付きの write 文を使ってテキスト出力をした方が良いだろう。

3.1.6 出力ファイルのオープンとクローズ

装置番号 *nw* を指定して write 文で出力すると，出力ファイル名は「fort.*nw*」という名前になるが，open 文を使えばファイル名を変更することができる。

```
open(nw,file= name [,form= format][,status= status][,err= num])
```

[ ] の部分は省略可能である。それぞれの記述の意味を以下に示す。

出力ファイル作成時の open 文の指定

nw	装置番号	オープンした後，write 文の装置番号として使う。
name	ファイル名	文字列で指定する（指定文字列を「'」で囲むか，文字変数を使う）。ファイル名は大文字・小文字を正しく指定する必要がある。
format	ファイル形式	文字列で指定する。書式付きのテキスト形式出力の時は省略する。書式なしのバイナリ形式出力を使うときは,「'unformatted'」を指定しなければならない。
status	ファイルの存在に関する情報	文字列で指定する。省略すると，ファイルが存在すればそれを使い，存在しなければ新しく作る。存在するファイルに書き込むと，古いデータは消えるので，注意しなければならない。 この他，既存のファイルを使うときは'old' を，存在しないファイルを使うときは'new' を指定する。この時，条件に合わないとエラーが起こる。
num	エラーしたときにジャンプする文番号	省略すると，エラーが起きたときにはプログラムがストップする。

例えば，装置番号「10」のテキスト形式ファイルを「test.out」という名にするときは

```
open(10,file='test.out')
```

と書く。また，装置番号「20」のバイナリ形式ファイルを「test.dat」という名にして，エラー処理をするときは

```
open(20,file='test.dat',form='unformatted',err=999)
```

と書く。エラーが起これば，文番号 999 の行へジャンプする。

オープンしたファイルは，クローズして初めて完成する。クローズする時は，close 文を使う。

```
close(nw)
```

nw はクローズするファイルの装置番号である。

但し，close 文を使わなくても，プログラムが正常に終了すれば，全てのファイルが自動的にクローズされるので通常は不要である。

3.2 データ入力

実行中のプログラムに対して，外部からプログラム中の変数に数値データを与えることを「入力する」といい，read 文を使って行う。read 文はプログラムの要求に応じて必要なデータを指定された変数に与えるのに用いる。シミュレーションでは初期値を与えればそれで後の計算を全て行うことができることが多いので，read 文でわざわざデータを与えなくてもプログラム中の変数に代入文で数値を指定し，必要に応じて変更してコンパイルし直してもさほどの手間ではない。しかし，大型コンピュータを利用する時など，コンパイルに時間がかかる時には，ファイルに入力データを書き込んで，プログラムから読み込むようにすれば便利である。

3.2.1 入力文の一般型

入力には read 文を用いる。

```
read(nr, form) 変数1, 変数2  ...
```

nr は装置番号 (整数) で，*nr* に適当な整数を与えると「fort.*nr*」という名のファイルから入力する[18]。出力ファイルと違うのは fort.*nr* という名のファイルが用意されていなければエラーになることである。write 文と同様に装置番号は $nr \geq 10$ が良い。

form の部分には「*」を与えておけばよい。read 文にも format による書式指定はあるのだが，あまり重要ではないので省略する。

例えば，

```
read(30,*) x,y,z
```

と書けば，fort.30 という名前のファイルを用意して，その中に x,y,z の数値を書き込んでおく必要がある。

例えば，

```
5.2    1.5    3
```

と書いておけば，read 文実行後，x=5.2，y=1.5，z=3 となって実行が継続する。

ファイルではなく，キーボード (正確には標準入力) から入力したいときには，以下のように書く。

```
read *, 変数1, 変数2  ...
```

この文を実行すると，キーボードからの数値入力が完了するまで，プログラムの実行が一時停止しているので，read 文のタイミングを考慮して適切に入力する必要がある。

read 文でも *form* を省略して装置番号のみ指定することができる。これを「書式なし read 文」という。

```
read(nw) 変数1, 変数2  ...
```

書式なし read 文で読み込む場合は，入力データ形式が「バイナリ形式」だと仮定されるので，書式なし write 文で作ったファイルを使う必要がある。また，出力変数の並びに合わせて実数・整数の区別をして入力する必要がある。

例えば，

```
real x,y
integer n
x = 10
y = 100
n = 10
write(20) x,n,y
```

というプログラムで作成された fort.20 というファイルから読み込むには，

```
real x,y
integer n
read(20) x,n,y
```

のように入力しなければならない。x も n も同じ 10 だからと思って，

[18]出力と同様，「fort」の部分はコンパイラに依存する。

```
real x,y
integer n
read(20) n,x,y
```

と読み込もうとするとエラーになってプログラムが終了する。

3.2.2 入力時のエラー処理

ファイルからデータを読み込むときに，要求したデータが存在しないときはエラーとなり，プログラムが強制終了する。これを防ぐには read 文中に err=*num*（文番号）を入れる。

```
read(nr, form,err= num) 変数1，変数2  ...
```

err=*num* を指定すると，入力エラーを検知した時に，*num* の文番号の行へジャンプする。もし「ファイルの終了」を検知するだけなら，err=*num* の代わりに end=*num* でも良い。

例えば，

```
do k = 1, 100
  read(10,*,err=999) x,y,z
  .......
enddo
.......

999  x = 100
```

と書けば，エラーが起こると文番号 999 の行にジャンプしてその行から引き続き処理を行う。

書式なしの read 文でも使える。

```
do k = 1, 100
  read(10,err=999) x,y,z
  .......
enddo
.......

999  x = 100
```

データの出力回数がわからない時には err か end 指定を入れておいた方がよい。

3.2.3 namelist を用いた入力

便利な入力形式として，namelist を用いる方法がある。例えば

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

という入力文では，入力ファイル fort.10 を

```
10.0   1.e10   100
```

のように作成するが，作成するためには入力変数の対応を常に憶えておかなければならない。必要なデータを全部書き込まなければならないし，順番を間違えてもいけない。

これに対し，namelist 入力文では，入力データを「変数＝データ」という形で作成するためわかりやすい。

namelist 入力文を使うときは，まず，入力する可能性のある変数名を namelist 文で登録する。

```
namelist /ネームリスト名/ 変数1，変数2  ...
```

これは非実行文なので，型宣言のように，実行文より前に書かなければならない。

例えば

```
real x,y
integer n
namelist /option/ x,y,n
```

と書く。変数の型宣言は別途必要である。

これに対して，入力文は

```
read(nr, ネームリスト名 [,err= num])
```

だけである。変数の指定は必要ない。

入力ファイルは次の形で用意する。

```
 &ネームリスト名
  変数１＝データ１，変数２＝データ２，...
 &end
```

例えば上例のようにネームリスト名が option の時には

```
 &option
  x=10.0, y=1.e10, n=100
 &end
```

のように記述する。次のようにコンマ無しで１行づつ書いても良い。

```
 &option
  x=10.0
  y=1.e10
  n=100
 &end
```

namelist 文を使うもう一つの利点は，必ずしも全部のデータを記述する必要がないことである。省略した場合は，read 文を実行する前までに代入された値がそのまま残る。このため，あらかじめ変数にデフォルト値を代入しておき，変更したいものだけ入力ファイルに記述すれば良い。

例えば

```
real x,y,a(10)
integer n,i
namelist /option/ x,y,a,n
x = 100.0
y = 100.e10
n = 0
do i = 1, 10
   a(i) = i
enddo
read(10,option)
```

のようにプログラムを書いて，fort.10 という名のファイルに

```
 &option
   x=10.0
   a(3)=5.0
 &end
```

と書き込んでおけば，read 文実行後，x だけが変化して，y や n は変化しない。配列 a の場合には，指定された要素 a(3) のみが変更される。

なお，namelist に登録された変数の内容は，次の namelist 出力文で出力することもできる。

```
write(nw, ネームリスト名)
```

ただし，namelist 出力を使うと，登録された変数全部のデータが標準出力形式で出力されるので，変数が多いとごちゃごちゃになる。取りあえず出力するのには使えるが，あまりお勧めはしない。

3.2.4 入力ファイルのオープン

上記のように，read 文は入力する時に「fort.*nr*」という名前のファイルから入力するのが標準である。これに対し，任意のファイルから入力するときは open 文を使う。

```
open(nr,file= name [,form= format][,status= status][,err= num])
```

[ ] の部分は省略可能である。それぞれの記述の意味は以下の通り。基本的には write 文と同じである。入力ファイルの場合にはファイルの存在をチェックすることが多いが，ファイルが存在しない時のエラー処理をする場合には status='old' にしておく必要がある。

入力ファイルの open 文の指定

nr	装置番号	オープンした後，read 文の装置番号として使う。
name	ファイル名	文字列で指定する（指定文字列を「'」で囲むか，文字変数を使う）。ファイル名は大文字・小文字を正しく指定する必要がある。
format	ファイル形式	文字列で指定する。書式付きのテキスト形式入力の時は省略する。書式なしのバイナリ形式入力を使うときは,「'unformatted'」を指定しなければならない。
status	ファイルの存在に関する情報	文字列で指定する。省略すると，ファイルが存在すればそれを使い，存在しなければ新しく作る。入力ファイルの場合，err によるエラー処理をする時には'old' を指定しなければならない。
num	エラーしたときにジャンプする文番号	省略すると，エラーが起きたときにはプログラムがストップする。入力ファイルが存在しない時の処理に使える。

例えば,「test.in」という名のテキスト形式ファイルを装置番号「10」で何の指定もなくオープンするときは

```
open(10,file='test.in')
```

となる。また，バイナリ形式ファイル「test.dat」を装置番号「20」で指定して，エラー処理をするときは

```
open(20,file='test.dat',form='unformatted',status='old',err=999)
```

となる。エラーが起これば，文番号 999 の行へジャンプする。例えば,「test.dat」という名のファイルが存在しないときの処理ができる。

第 4 章　知っておくと便利な文法

4.1　継続行

自由形式でプログラムを書けば 1 行にいくらでもプログラムを書くことができるが，実際にはエディタの横幅を越えると読みづらい。そこで，長くなるときには適当な所で切って，次の行にまたがって書くことができる。これを「継続行」と言う。行を継続するには，続ける前の行の最後に「&」を書く。

例えば，

```
print *,alpha,beta,gamma,delta,epsilon,zeta,eta,iota,kappa,lambda,mu
```

は，

```
print *,alpha,beta,gamma,delta,epsilon &
       ,zeta,eta,iota,kappa,lambda,mu
```

と書ける。継続行は何行にわたってもかまわない。

```
print *,alpha,beta,gamma &
       ,delta,epsilon &
       ,zeta,eta,iota &
       ,kappa,lambda,mu
```

と書いても良い。

4.2　parameter 指定をした変数

宣言文で配列を宣言するとき，その要素数は a(100) のように数値で指定しなければならない。このため，配列の長さを変更するときには宣言文中の数値を一度に全て変更する必要があり，労力が要ると同時にエラーを起こす原因となる。そこでこの不便さを補うために，parameter 指定をした変数を作ることができる。

```
integer, parameter :: nmax=10
real a(nmax),b(nmax)
integer i
do i = 1, nmax
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

integer の後に続く「, parameter」が parameter 指定である。またこの時は変数リストの前に::を書かなければならない。

このプログラムは次のプログラムと同等である。

```
integer i
real a(10),b(10)
do i = 1, 10
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

parameter 指定された変数はコンパイルの段階で指定した定数に置き換えられるので宣言文の要素として使えるのである。上記の例で言えば，parameter 指定を使わなければ変更箇所 (10) は 3 箇所あるが，parameter 指定を使った場合には 1 箇所 (nmax=10 の部分のみ) でよい。

parameter 指定をした整変数を別の parameter 指定で使うこともできる。例えば，

```
integer, parameter :: nmax=10, nmax2=nmax**2
```

と書ける。これは,

```
integer, parameter :: nmax=10, nmax2=100
```

と同じである。使う変数 (この例では nmax2) は使われる変数 (nmax) より後で宣言しなければならない。

注意しなければならないのは，parameter 指定をした「変数」が，実際は定数であることである。このため，実行文で parameter 指定をした変数に他の値を代入しようとすると文法エラーになる。このため，parameter 指定をする変数は 1 文字程度の簡単な名前にしないように気をつけるべきである。

例えば，横着して上記の例文を

```
integer, parameter :: nmax=10, i=100
real a(nmax),b(nmax)
do i = 1, nmax          ! これはエラー
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

などと書かないようにすること。

4.3 文字列の色々な使い道

Fortran は基本的に数値計算用の言語であるが，Fortran77 から文字列が使いやすくなり，より積極的な利用が可能になっている。以下にその中のいくつかの利用法について述べる。

4.3.1 文字列変数を使う時の注意

文字列とは「'」または「"」ではさんだものをいう。

```
例： 'abc'  'Taguchi T.'  "123.5678"  など
```

文字列にはスペースも入れられる。また数値定数 123 と文字列'123' は全く異なるものであるから注意すること。文字列の変数を作るには character 宣言をする。

```
character c1*n1,c2*n2,...
```

ここで c1，c2 が変数名，$n1$，$n2$ はそれぞれの変数の文字数である。例えば

```
character c1*10
```

と宣言すると c1 には 10 個までの文字が代入できる。これに'abc' という文字を代入したければ

```
c1 = 'abc'
```

と通常の代入文でできる。代入される文字'abc' は c1 の先頭から順に代入され，残りの領域にはスペースが入る。

文字は部分的に取り出したり，連結したりすることが可能である。部分的に取り出すには「:」で区切って位置の範囲を指定する。例えば，c1='abcdefg' ならば，c1(3:5) は'cde' を示す。前後の数を等しくすれば 1 文字を取り出すことができる。

```
c1 = 'abcdefg'
do i = 1, 7
   print *,c1(i:i)
enddo
```

とすれば，1 行 1 文字ずつ出力される。

文字列を連結するには演算子「//」を用いる。

```
c1 = 'abc'//'xyz'
```

この結果，c1 には'abcxyz' という文字列が代入される。もっともこの例ならば最初から c1='abcxyz' で良いのだが，変数に入った文字列をつなぐことができるところがポイントである。

ただし，変数をつなぐ時には注意が必要である。例えば，

```
character c1*10,c2*20
c1 = 'abc'
c2 = c1//'xyz'
```

と書いた結果，c2 には'abcxyz' という文字列が代入されると思ったら間違いである。結果は'abc　　xyz'。これは，Fortran の文字変数にその宣言した文字数よりも短い文字列を代入すると，余った部分をスペースで埋めるからである。即ち，c1 という文字変数の内容は，'abc' ではなく，'abc　　' である。そこで，この無駄なスペースを削除するには，上記の部分文字列指定を使う。

```
character c1*10,c2*20
c1 = 'abc'
c2 = c1(1:3)//'xyz'
```

この結果，c2 には'abcxyz' が代入される。

しかし，この方法は代入した文字数が不明の時には使えない。そこで，関数 trim が用意されている。

```
character c1*10,c2*20
c1 = 'abc'
c2 = trim(c1)//'xyz'
```

この結果も c2 には'abcxyz' が代入される。

なお，文字数を返す関数として len が用意されているが，同様の問題が存在するので使う時には注意が必要である。例えば，

```
character c1*10
integer m,n
m  = len('abc')
c1 = 'abc'
n  = len(c1)
```

とすると，m の値は 3 だが，n の値は 10 である。

4.3.2　出力における文字列の利用

ここでは，出力における文字列の利用法を述べる。最もよく使うのは，print 文中であろう。

```
a = 10.0
b = 5.0
print *,a,b
```

と書けば，出力は

```
10.000000     5.000000
```

のように数字が画面に出るだけなので，どの数値がどの変数の出力なのかわからない。そこで，print 文を

```
print *,'a=',a,'   b=',b
```

と書けば，出力は

```
a= 10.000000   b= 5.000000
```

のようになる。なお，b の文字の前にスペースを入れてあるところがポイントである。書式を指定しない場合にはデータが隙間なく出力されるので，スペースが無いと前の数字（0）と「b」がくっついてしまう。

3.1.3 で紹介したような，format の内容を print 文の中に埋め込む方法も文字列の利用である。

例えば，

```
  print 600,x
600 format(' x=',es12.5)
```

は次のように書き換えることができる。

```
print "(' x=',es12.5)",x
```

即ち，format 以下をかっこ付きで，文字列にして *form* の位置に書き込むのである。この文字列は変数でも良い。変数を使うと実行時に format を変更できる。例えば

```
real x
character form*20
if (x >= 1.e5) then
   form = "('x=',e12.5)"
 else
   form = "('x=',f10.5)"
endif
print form,x
```

とすれば x が 1.e5 以上のときには e12.5 編集で，小さいときには f10.5 編集で出力される。一個の print 文でデータに応じた format を使うことができるわけである。

4.3.3 数値・文字列変換

さてここまでの例は write 文や read 文の *form* の位置に文字列を使用していたが，装置番号（*nr* や *nw*）の位置に使用することも可能である。これは「文字」→「数値」またはその逆の変換をするときに使用する。上記のように 123 という数値と'123' という文字列は異なるが，場合によっては 123 という数値からそれに相当する文字を作ったり，逆に'123' という文字を数値として使用したい場合がある。よく考えれば，read 文はファイルなどに入った「文字データ」を「数値」に変換して読み込む文であり，write 文は「数値」を「文字列」に置き換えて出力しているのであるから，その変換機能だけを使うのである。

「数値」→「文字」変換をするには write 文を使う。例えば

```
real x
character ch*20
x = 123.5
write(ch,'(f10.5)') x
```

とすれば，ch に' 123.50000' という「文字」が代入される。但し，文字に変換するときは format 文に従って埋め込まれるので，十分な長さの文字列を用意しておく必要がある。この変換はグラフィックサブルーチンを使って

図形中に数値データを描く場合などで使える。

逆に,「文字」→「数値」変換をするには read 文を使う。例えば

```
real x
character ch*20
ch = '12345.0'
read(ch,*) x
```

とすれば，x に 12345.0 という「数値」が代入される。

4.4 配列計算機能を使った配列の初期化

通常，配列に数値を代入する時は do 文を使うが，全ての配列要素に同じ値を代入する時は Fortran の配列計算機能を使って簡単に書くことができる。例えば,

```
real a(10,100)
integer m,n
do n = 1, 100
  do m = 1, 10
    a(m,n) = 0
  enddo
enddo
```

のように，2 次元配列 a(m,n) の全要素に 0 を代入するプログラムは,

```
real a(10,100)
a = 0
```

と書ける。すなわち「a」という配列名が，a(m,n) という全配列要素を代表するのである。

また，配列の一部に代入することもできる。例えば,

```
real a(10,100)
integer m,n
do n = 20, 30
  do m = 4, 8
    a(m,n) = 1
  enddo
enddo
```

のように配列 a の一部にのみ 1 を代入する時は,

```
real a(10,100)
a(4:8,20:30) = 1
```

と書けるのである。このように配列範囲の下限と上限を「:」でつないで指定した配列を部分配列という。

Fortran における配列計算はこの機能を一般化したもので，配列の次元と要素数が一致していれば,

```
real a(10,100),b(10,100),c(10,100)
a = 3
b = 2*a
c = a/(b - 30)
```

のような計算ができるものである。これらは全配列要素について，要素毎に計算したのと同じ結果になる。

配列計算を使えばプログラムが短くてスマートになるし，配列要素の並びを考慮してコンパイルされるので計算も速い。その反面，プログラムの動作をわかりにくくして返ってエラーを探すのに手間取ることがある。計算機の動作をあまり良く理解していないプログラム初心者は，数値を代入する初期化に利用する程度にしておいた方が良いだろう。

第 5 章　読みやすいプログラムを書くには

ここではプログラムを読みやすくするための「こつ」をいくつか示します。これらの「こつ」はプログラムが長くなればなるほど意味があるもので，最初のうちは重要性があまりわからないかもしれませんが，面倒がらずに心掛けましょう。

5.1　コメント文を多用せよ

Fortran では「!」の後に続く文字列は全て無視される。逆に言えば，何を書いてもよい。これをコメント文と言う。コメント文を機会あるごとにプログラム中にいれて，内容を説明するように心掛けること。さもなくば，プログラムが長くなるにつれて，自分でも何を書いたか忘れてしまう。

例えば，

```
!  area of circles
   s = pi*r*r
```

のように，1 列目に!を使えば，その行はコメント行となる。また，

```
   s = pi*r*r          !  円の面積
   v = 3*pi*r*r*r/4    !  球の体積
```

のように，プログラム文の最後に使ってもよい。またこの例のように日本語を使って書いてもかまわない。ただし，日本語環境によっては正しく認識できず文字化けすることがある。ローマ字でも良いから半角英数字だけに限定した方が無難である。

5.2　ブロックは区別するために，字下げせよ

今までの例でも示したが，プログラムは do や if などのブロック中で，字下げ（インデント）をするのがよい。これは，ブロックの範囲が一目でわかるからである。

例えば，if ブロックの中，

```
if (n < 0) then
   x = 10.0
 else if (n < 10) then
   x = 1.0
 else
   x = 0.0
endif
```

や do ブロックの中

```
do i = 1, n
  b(i) = a(i)
enddo
```

などはスペースを入れてずらせておくとブロックの範囲がわかりやすい。2〜4 個のスペース程度で良いだろう。

最近のエディタは,「オートインデント」といって，改行すると，その前の行と先頭をそろえるようにカーソルの位置が移動する機能があるので，インデントするのはそれほどの手間ではない。

5.3　文字間や行間は適当に空ける

文中の文字間は適当に空けたほうが読みやすい。筆者は「=」と「+」の両側に必ずスペースを入れることにしている。特に「=」は適当にそろえたほうが読みやすい。さらに，同じパターンを持ち，少しずつ内容が違う文を

並べるときにはパターンが並ぶようにスペースを入れるのがよい。以下は，筆者のシミュレーションプログラムの 1 節である。

パターンをそろえた場合

```
wm( 1,m1) = (1-dx)*(1-dy)*(1-dz)
wm( 2,m1) =    dx *(1-dy)*(1-dz)
wm( 3,m1) = (1-dx)*   dy *(1-dz)
wm( 4,m1) =    dx *   dy *(1-dz)
wm(11,m1) = (1-dx)*(1-dy)*   dz
wm(12,m1) =    dx *(1-dy)*   dz
wm(13,m1) = (1-dx)*   dy *   dz
wm(14,m1) =    dx *   dy *   dz
```

この部分をスペース抜きにしてみると以下のようになる。

スペースを抜いた場合

```
wm(1,m1)=(1-dx)*(1-dy)*(1-dz)
wm(2,m1)=dx*(1-dy)*(1-dz)
wm(3,m1)=(1-dx)*dy*(1-dz)
wm(4,m1)=dx*dy*(1-dz)
wm(11,m1)=(1-dx)*(1-dy)*dz
wm(12,m1)=dx*(1-dy)*dz
wm(13,m1)=(1-dx)*dy*dz
wm(14,m1)=dx*dy*dz
```

この 2 例は全く同じ結果を出すので文法的な優劣はない。それなら,「パターンをそろえた場合」は単なる美的趣味なのかというと，そうではない。パターンを統一することでエラーを発見しやすくすることが第一目的である。

プログラムのチェックというのは書くこと以上に骨の折れる仕事である。プログラムというのはちょっと書き間違えただけで，全く異なる答えを出す。上記の例で dx のどれか 1 つを dy と書き間違えただけで全く予想しない結果を生むのである。このようなミスを如何に確実に見つけるか。これこそがプログラマーの重要な技能であり，はっきり言って経験でしか養われない。初心者の学生さんは「細かいことなんかどうでもいいや」と思われるかもしれないが，それは大きな間違いである。

また，文と文の間に適当にコメント文や，何も書いていない行を挿入すれば，プログラムの流れに区切りができてわかりやすくなる。つまり文章を段落に分けるように，プログラムを区分けするのである。

5.4 文番号は規則を付けよ

文番号は，使用する文によって規則を付けるようにする。例えば goto 文は 10，20，30，...，read 文の format 文は 500 台，print 文の format 文は 600 台，エラー処理は 900 台，... など。また最初から，1,2,3 のように一つづつ増やさないこと。後で挿入しなければならなくなったときに番号を単調増加にするのが面倒になる。単調増加にしないからと言って文法的な間違いではないのだが，結局プログラムをわかりにくくしてしまう。

5.5 意味不明の定数はできるだけ減らす

数学的に決まっている定数（2 倍するとか，1 を加えるとか，3 乗するとか）の場合以外は，できるだけ定数の使用量を減らすほうがよい。

例えば，プランク定数を使ってエネルギー計算をするときに,

```
energy = 6.626176e-34*f
```

と書くよりも

```
hplanck = 6.626176e-34
energy  = hplanck*f
```

と書いたほうがわかりやすい。また，電子の質量 m_e に対して，$m_e c^2$ を計算してそのデータを利用するときに，$m_e c^2 = 511.0$ keV だからといって，

```
g = e/511.e3
```

と書くよりも，$m_e = 9.1093897 \times 10^{-31}$ kg，$c = 2.99792458 \times 10^8$ m/s，1 eV$= 1.60217733 \times 10^{-19}$ J，という良く知られた定数を使って，

```
me  = 9.1093897e-31
cl  = 2.99792458e8
ev  = 1.60217733e-19
mc2 = me*cl**2/(1000.0*ev)
g   = e/mc2
```

としたほうが，わかりやすいし確認しやすいと思う。

この例のような，物理定数を使うときには 4.2 で述べた parameter 指定を使うとさらに良い。

```
real, parameter :: me=9.1093897e-31, cl=2.99792458e8, ev=1.60217733e-19
real, parameter :: mc2=me*cl**2/(1000.0*ev)

g   = e/mc2
```

これを module にしておくとルーチン毎に宣言する手間が省ける。

```
module physics
   real, parameter :: me=9.1093897e-31, cl=2.99792458e8, ev=1.60217733e-19
   real, parameter :: mc2=me*cl**2/(1000.0*ev)
end module physics
```

また，do ブロックを 100 回繰り返す，とか，10 回ごとに出力する，とかいう制御数も後で変更しやすいように，変数にしておくべきである。例えば，

```
do i = 1, 100
   a(i) = b(i)**2
enddo
```

という do ブロックにおいて，100 という数は配列要素数に依存するのだから，後で変更しやすいように

```
nmax = 100
do i = 1, nmax
   a(i) = b(i)**2
enddo
```

というように変数にしておく。この例では，100 という数が 1 カ所しかないのであまり有り難みがないが，長いプログラムになると重要性がわかるはずである。

5.6 配列要素範囲の変数化

さらに配列を使うときはその要素範囲も変更しやすいように考えておく必要がある。

```
real a(10)
integer i
do i = 1, 10
  a(i) = i*i
enddo
```

もしこのプログラムを変更して，a(10) を a(20) にしようとすれば，do 文中の i = 1, 10 も i = 1, 20 に

変更しなければならない。こういう変更はプログラムが大きくなると，変更場所を探し出すだけで大変である。

そこで，

```
real a(10)
integer i,nmax
nmax = 10
do i = 1, nmax
  a(i) = i*i
enddo
```

というように，変数を使ってできるだけ変更箇所が少なくてすむようにするべきである。さらに，複数のプログラム単位で使うときは，nmax を module 文中に入れて，どこか 1 カ所，例えばメインプログラムで変更すればよいようにする。ただし，このようなグローバル変数には短い名前を付けないようにしなければならない。

また，配列宣言も含めたいときには，parameter 指定を使えばよい。

```
integer, parameter :: nmax=10
real a(nmax)
integer i
do i = 1, nmax
  a(i) = i*i
enddo
```

こういう書式は最初は煩わしいかもしれないが，後で大きな御利益がある。あまり横着しないのが上達への早道である。これは，横着者を自称する筆者が言うのだから間違いない。

演 習 問 題 3

(3–1)　ヘロンの公式 (その 2)

三角形の三辺の長さを a，b，c とすると，その三角形の面積 S は次式（ヘロンの公式）で表される。

$$S = \sqrt{s(s-a)(s-b)(s-c)}$$

ただし，$s = \dfrac{a+b+c}{2}$ である。

a，b，c の数値を引数とし，この公式を使って三角形の面積を計算して戻り値 `s` にするサブルーチンを作れ。

次にそれを使って問題 (1-2) の解答をサブルーチンを使う形に修正せよ。

(3–2)　2 次方程式の根 (その 3)

3 個の実変数 a，b，c を引数とし，それから，

$$ax^2 + bx + c = 0$$

の 2 根 `x1`，`x2`，および判別式 `D`$= \sqrt{b^2 - 4ac}$ を計算して戻り値にするサブルーチンを作成せよ。ただし，2 実根になる場合はそれぞれを `x1` と `x2` に代入し，2 虚根になる場合は，実部を `x1` に，虚部を `x2` に代入するようにせよ。

次にそれを使って問題 (1-1) の解答をサブルーチンを使う形に修正せよ。

(3–3)　3 次元ベクトルと電磁気学 (その 2)

3 次元ベクトル $\mathbf{A} = (A_1, A_2, A_3)$ を 3 次元配列 `a(3)` で表すとする。電場ベクトル $\mathbf{E}$，磁場ベクトル $\mathbf{B}$ および荷電粒子の速度ベクトル $\mathbf{v}$ を配列で表して，これらの配列を引数として与えると内積 $\mathbf{v}\bullet\mathbf{E}$，および外積ベクトル $\mathbf{v}\times\mathbf{B}$ を計算して戻り値とするサブルーチンを作れ。なお戻り値の外積ベクトルも配列とする。

次に問題 (1-7) の解答をサブ－ルーチンを使う形に修正せよ。

(3–4)　統計計算

1 次元配列，`a(n)` を引数とし，

$$\text{平　均}\bar{A} = \frac{1}{N}\sum_{i=1}^{N} A_i$$

$$\text{標準偏差}\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(A_i - \bar{A})^2}$$

を計算して戻り値とするサブルーチンを作成し，問題 (1-8) の解答をサブ－ルーチンを使う形に修正せよ。

(3–5)　Newton 法 2

問題 (2-3) の Newton 法を用いて方程式の根を計算するプログラムをサブルーチンを使う形に修正せよ。まず関数 $f(x)$ に対し，変数値 x を与えると $f(x)$ と $f'(x)$ を返すようなサブルーチンを作成する。それを使って Newton 法で根を求めるプログラムにすればよい。

(3–6)　行列計算

2 次元配列，a(n,n)，b(n,n) から行列の和，c(n,n)，および行列の積，d(n,n) を計算するサブルーチンを作成せよ。この時，整合配列を用いて与える配列の次元が自由に変えられるようにすること。

なお，行列の和と積の定義を以下に示す。

$$c_{i,j} = a_{i,j} + b_{i,j}$$
$$d_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

(3–7)　行列式

u(3)，v(3)，w(3) という 3 つの 1 次元配列データから行列式

$$Det = \begin{vmatrix} u(1) & v(1) & w(1) \\ u(2) & v(2) & w(2) \\ u(3) & v(3) & w(3) \end{vmatrix}$$

を計算するサブルーチンを作成せよ。

(3–8)　連立方程式

3 元 1 次の連立方程式,

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

を解くプログラムのサブルーチンを作成せよ。

クラメールの公式によれば，答えは以下のようになる。

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{D}, \qquad x_2 = \frac{\begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix}}{D}, \qquad x_3 = \frac{\begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{vmatrix}}{D}$$

ただし，

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

である。

プログラムは，a_{mn} を a(m,n) という配列で表し，b_m を b(m) という配列で表し，x_m を x(m) という配列で表して，サブルーチンの引数は，a と b と x とする。行列式の計算には，問題 (3-7) で作成したサブルーチンを利用すればよい。

(3–9)　常微分方程式 (境界値問題) の解

電荷密度 $\rho(x)$ がわかっているときに電位が満足する 1 次元ポワソン方程式

$$\frac{d^2V(x)}{dx^2} = -\rho(x)$$

を解くには，以下のようにする。

等間隔 h 毎に並んだ座標点 x_0，x_1，x_2，$\cdots$，x_N に対して，$V_m = V(x_m)$，$\rho_m = \rho(x_m)$ として，上式の左辺を差分で近似すれば，

$$\frac{V_{m+1} - 2V_m + V_{m-1}}{h^2} = -\rho_m$$

となる。これを座標点に関して書き下せば，

$$\begin{aligned}
V_0 - 2V_1 + V_2 &= -\rho_1 h^2 \\
V_1 - 2V_2 + V_3 &= -\rho_2 h^2 \\
V_2 - 2V_3 + V_4 &= -\rho_3 h^2 \\
&\vdots \\
V_{N-2} - 2V_{N-1} + V_N &= -\rho_{N-1} h^2
\end{aligned}$$

となるが，両端の電位 $V_0 = V(x_0)$，$V_N = V(x_N)$ が与えられていれば，最初と最後の方程式は，

$$\begin{aligned}
-2V_1 + V_2 &= -\rho_1 h^2 - V_0 \\
V_{N-2} - 2V_{N-1} &= -\rho_{N-1} h^2 - V_N
\end{aligned}$$

になるので，これは V_1，V_2，$\cdots$，V_{N-1} に対する連立一次方程式になる。

一般に，

$$a_m V_{m-1} + b_m V_m + c_m V_{m+1} = d_m \qquad (m = 1, 2, \ldots, N-1)$$

を解くときには，まず，$G_0 = 0$，$H_0 = 0$ から出発して，

$$G_m = -\frac{c_m}{b_m + a_m G_{m-1}}, \qquad H_m = \frac{d_m - a_m H_{m-1}}{b_m + a_m G_{m-1}} \qquad (m = 1, 2, \ldots, N-1)$$

を計算し，次に，V_m を以下の式で計算する (計算の方向に注意！)。

$$V_m = G_m V_{m+1} + H_m \qquad (m = N-1, N-2, \ldots, 1)$$

以上の方法を用いて，次の電荷密度における電位を計算するプログラムを作成せよ。

$$\rho(x) = \sin\left(\frac{2\pi x}{Nh}\right)$$

ここで，境界条件は $V_0 = V_N = 0$ でよい。

(3–10)　連立常微分方程式：Runge-Kutta 法

今，$N+1$ 個の質量 m のおもりが一直線に並んでいて，隣り合ったおもりはバネ定数 k のバネで繋がれているとする。このおもりの位置を左側から，z_0，z_1，$\cdots$，z_N とすると，i 番目のおもりの運動方程式はその速度を v_i として，

$$\begin{aligned}\frac{dz_i}{dt} &= v_i \\ m\frac{dv_i}{dt} &= -k(z_i - z_{i-1}) - k(z_i - z_{i+1})\end{aligned}$$

である。このおもりの運動を 2 次の Runge-Kutta 法で解析せよ。

ここで，2 次の Runge-Kutta 法とは微分方程式 $\dfrac{dx}{dt} = f(t,x)$ に対し，時刻 t_n の値 x_n から出発して時刻 $t_{n+1} = t_n + \Delta t$ の値 x_{n+1} を

$$\begin{aligned}k_1 &= f(t_n, x_n)\Delta t \\ k_2 &= f(t_n + \Delta t, x_n + k_1) \\ x_{n+1} &= x_n + \frac{1}{2}(k_1 + k_2)\end{aligned}$$

で計算する方法である。Δt は適当に決める。プログラムは

1. 時間を進めるメインプログラム
2. オイラー法または Runge-Kutta 法の 1 ステップを計算するサブルーチン
3. 運動方程式の右辺を計算するサブルーチン

の 3 個から構成するものとする。さらに，z_i や v_i は引数で与え，m や k はグローバル変数として `module` と `use` 文を使って共有するようにせよ。なお，初期条件は以下の通り。

$$\begin{aligned}z_i &= hi + g\sin(p\,hi) \qquad\qquad (i = 1 \cdots N-1) \\ v_i &= 0\end{aligned}$$

但し，p は適当な整数で，$h = \dfrac{2\pi}{N}$ である。また，端にあるおもり（z_0 と z_N）は動かないとする。（つまり，計算しないで初期条件のままとする）

(3–11)　楕円型偏微分方程式

電荷密度 $\rho(x,y)$ がわかっているときに電位の満足する 2 次元ポワソン方程式

$$\frac{\partial^2 V(x,y)}{\partial x^2}+\frac{\partial^2 V(x,y)}{\partial y^2}=-\rho(x,y)$$

を解くには次のようにする。

等間隔 h 毎に並んだ x 座標点 x_0，x_1，$\cdots$，x_M と y 座標点 y_0，y_1，$\cdots$，y_N に対して，$V_{m,n}=V(x_m,y_n)$，$\rho_{m,n}=\rho(x_m,y_n)$ として，上の偏微分方程式を差分化すれば次式になる。

$$\frac{V_{m-1,n}-2V_{m,n}+V_{m+1,n}}{h^2}+\frac{V_{m,n-1}-2V_{m,n}+V_{m,n+1}}{h^2}=-\rho_{m,n}$$

これは $V_{m,n}$ に対する連立一次方程式である。これを解くには以下のように変形して反復法で解く。これを，ガウス・ザイデル法という。

$$V_{m,n}^{(k+1)}=\frac{1}{4}\left(V_{m-1,n}^{(k+1)}+V_{m,n-1}^{(k+1)}+V_{m+1,n}^{(k)}+V_{m,n+1}^{(k)}+\rho_{m,n}h^2\right)$$

ここで，反復回数 k の値に注意すること。

ガウス・ザイデル法を使って，次の電荷密度における電位を計算するプログラムを作成せよ。

$$\rho(x,y)=\sin\left(\frac{2\pi x}{Nh}\right)\sin\left(\frac{4\pi y}{Nh}\right)$$

ここで，境界条件は $V_{0,n}=V_{m,0}=V_{M,n}=V_{m,N}=0$ でよい。

また，収束の判定は

$$\sum_{n=1}^{N-1}\sum_{m=1}^{M-1}|V_{m,n}^{(k+1)}-V_{m,n}^{(k)}|<\epsilon$$

で行うこと。

(3–12)　放物型偏微分方程式 (陽解差分法)

次の温度 $T(x,t)$ に関する熱伝導方程式を，陽解差分法 (Explicit 法) を用いて解析せよ。

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$$

但し，陽解差分法とは，$T_i^n = T(h\,i, \Delta t\,n)$ に対して，

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \kappa \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{h^2}$$

と近似する方法である。この方程式を $T_i^{n+1} = \ldots$ の形に変形して解く。

ただし，境界条件として，$T_0^n = 0$，$T_L^n = 0$ とする。ここで初期条件は，i の最大値を L として

$$\begin{cases} T_i^0 = i & (i < L/2) \\ T_i^0 = L - i & (i > L/2) \end{cases}$$

とせよ。プログラムは

1. 時間を進めるメインプログラム
2. 初期条件を計算するサブルーチン
3. 陽解法で 1 ステップ進めるサブルーチン

の 3 個から構成するものとする。温度 T_i^n，h，Δt，κ などの変数はグローバル変数にして `module` と `use` 文で共有する。また，メモリの節約のため，T_i^n を配列 `T1(i)` で表し，T_i^{n+1} を配列 `T2(i)` で表して，`T1` と `T2` のデータをうまく交換するように考える。

完成したプログラムで，$\frac{\kappa \Delta t}{h} > 0.5$ とすると，不安定になることなども確かめよ。

(3–13)　放物型偏微分方程式 (陰解差分法)

温度 $T(x,t)$ に関する熱伝導方程式を，陰解差分法 (Implicit 法) を用いて解析せよ。

但し，陰解差分法とは，

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \frac{1}{2}\left[\kappa \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{h^2} + \kappa \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{h^2} \right]$$

と近似する方法である。この方程式を変形すれば

$$a_i T_{i-1}^{n+1} + b_i T_i^{n+1} + c_i T_{i+1}^{n+1} = d_i \qquad (i = 1, 2, \ldots, L-1)$$

の形になるので，1 次元ポワソン方程式の項で述べた連立一次方程式の解法によって T_i^{n+1} が計算できる。

ただし，境界条件や初期条件は陽解法と同じでよい。プログラムは

1. 時間を進めるメインプログラム
2. 初期条件を計算するサブルーチン
3. 陰解法で 1 ステップ進めるサブルーチン

の 3 個から構成するものとする。温度 T_i^n，h，Δt，κ などの変数はグローバル変数にして `module` と `use` 文で共有する。また，メモリの節約のため，T_i^n を配列 `T1(i)` で表し，T_i^{n+1} を配列 `T2(i)` で表して，`T1` と `T2` のデータをうまく交換するように考える。

完成したプログラムでは，$\frac{\kappa \Delta t}{h} > 0.5$ にしても不安定にならないことなどを確かめよ。

(3–14)　分子動力学

2 個の距離 r 離れた分子間の位置エネルギーを表す近似式の一つに次式のようなレナード・ジョーンズポテンシャルがある。

$$U(r) = 4\varepsilon \left\{ \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right\}$$

この式は，分子が近づくと斥力（反発力）が働き，遠ざかると引力が働くことを示している。レナード・ジョーンズポテンシャルの勾配を計算することで分子に働く力が計算できる。すなわち，2 個の分子の位置を $\mathbf{r}_i$，$\mathbf{r}_j$ とすると，分子 j が分子 i に及ぼす力は，

$$\mathbf{F}_{ij} = -\nabla_i U(|\mathbf{r}_i - \mathbf{r}_j|) = \frac{4\varepsilon}{\sigma} \left\{ 12 \left(\frac{\sigma}{r_{ij}}\right)^{13} - 6 \left(\frac{\sigma}{r_{ij}}\right)^{7} \right\} \frac{\mathbf{r}_i - \mathbf{r}_j}{r_{ij}}$$

となる。ここで,$r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ である。

今，分子が N 個あるとすると，分子 i に加わる力は i 以外の分子から受ける力の合力なので,

$$\mathbf{F}_i = \sum_{\substack{j=1 \\ i \neq j}}^{N} \mathbf{F}_{ij} = \sum_{\substack{j=1 \\ i \neq j}}^{N} \frac{4\varepsilon}{\sigma} \left\{ 12 \left(\frac{\sigma}{r_{ij}}\right)^{13} - 6 \left(\frac{\sigma}{r_{ij}}\right)^{7} \right\} \frac{\mathbf{r}_i - \mathbf{r}_j}{r_{ij}}$$

となる。これを用いて N 個の分子の運動方程式

$$m_i \frac{d^2\mathbf{r}_i}{dt^2} = \mathbf{F}_i = \sum_{\substack{j=1 \\ i \neq j}}^{N} \frac{4\varepsilon}{\sigma} \left\{ 12 \left(\frac{\sigma}{r_{ij}}\right)^{13} - 6 \left(\frac{\sigma}{r_{ij}}\right)^{7} \right\} \frac{\mathbf{r}_i - \mathbf{r}_j}{r_{ij}} \qquad (i = 1...N)$$

を解くプログラムを作成せよ。ここで，分子数が増えると時間がかかるので，効率よく時間積分をするためにベルレ法を用いる。ベルレ法とは次式のように 2 段階で位置ベクトルを更新する方法である。まず第 1 段階で速度を更新する。

$$\mathbf{v}_i^n = \mathbf{v}_i^{n-1} + \frac{\Delta t}{m_i} \frac{\mathbf{F}_i^n + \mathbf{F}_i^{n-1}}{2}$$

この更新された速度を用いて第 2 段階で位置を更新する。

$$\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \Delta t \mathbf{v}_i^n + \frac{\Delta t^2}{2m_i} \mathbf{F}_i^n$$

ここで，$\mathbf{F}_i^n$ とは，n ステップ目の位置 $\mathbf{r}_i^n$ を用いて計算した力 $\mathbf{F}_i$ である。

初期条件としては，まず適当な間隔離れた 2 個の分子でテストし，正常に動作することが確認されたら，正三角形の頂点に配置した 3 個の分子や正四面体の頂点に配置した 4 個の分子などの動きを計算せよ。

(3–15)　モンテカルロ法

世の中の動きは，運動方程式のような微分方程式を用いて確実に予測できるとは限らない。例えば，さいころを投げて次にどの数字が出るかで進み方を決めるすごろくのように，次の瞬間に何が起こるかわからない事象を元にして未来の様子を計算する方法をモンテカルロ法という。

計算機でさいころの代わりをするものは「乱数」と呼ばれている。Fortran 90 では，サブルーチン `random_number` が用意されているので，これを用いれば乱数を使った計算ができる。例えば，1 個の乱数を使うときには

```
call random_number(x)
```

と書けば，変数 `x` に 0 から 1 までの値を持つ実数が代入される。このサブルーチンで返ってくる値は予測がつかないので，2 回目に同じ呼び出しをしても同じ `x` が返ってくるとは限らない。

もし，乱数を一度にたくさん用意したいときには配列を使う。例えば，配列 `a(10)` に 10 個の乱数を代入するときは次のように書けばよい。

```
call random_number(a(1:10))
```

モンテカルロ法を使って円周率 π を計算してみよう。2 個の乱数 x，y を発生させてこれから座標 (x, y) を作る。この乱数の座標点を N 個作り，その中で原点から半径 1 の円より内側にある点の数 N_1 を数えて $p = \dfrac{4N_1}{N}$ を計算する。座標点数 N が増えると，この p が π に近づくことを確かめよ。

(3–16)　粒子密度 – 分布関数

2 次元領域，$0 \leq x \leq Lh$，$0 \leq y \leq Mh$，の内部に N 個の粒子があるとき，粒子の密度分布は以下のような手順で計算することができる。

まず，2 次元領域に等間隔 h 毎に並んだ x 座標点 x_0，x_1，$\cdots$，x_L と y 座標点 y_0，y_1，$\cdots$，y_M で指定した格子を導入する。次に密度配列 `Rho(0:L,0:M)` に初期値として 0 を代入しておいて，`i` 番目の粒子座標 `(x(i),y(i))` から以下のようにして各格子内にある粒子数をカウントする。

```
m = x(i)/h
n = y(i)/h
Rho(m,n) = Rho(m,n) + 1
```

ただし，`m` と `n` は整数型変数である。すなわち，`x(i)/h` と `y(i)/h` を切り捨てることで格子座標を計算する。

乱数を使って適当な粒子の座標を N 個作成し，密度が全領域でほぼ一定になることを確かめよ。

(3–17)　ブラウン運動 – 拡散

ある点から粒子を速度 v で出発させたところ，時刻 Δt ごとに周りの粒子に衝突して方向を変えるとする。この時の粒子の進む様子を計算してみよう。

粒子の初期位置は，全て $(\dfrac{Lh}{2}, \dfrac{Mh}{2})$，すなわち，2 次元領域の中央とする。計算は

1. 1 ステップ毎に乱数 R を発生させ，$\theta = 2\pi R$ で，方向角 θ を計算する
2. 1 ステップ後の位置を $(x_{n+1}, y_{n+1}) = (x_n + v\Delta t \cos\theta, y_n + v\Delta t \sin\theta)$ で計算する
3. 粒子が $0 \leq x \leq Lh$，$0 \leq y \leq Mh$ の領域から外に出たら計算をストップする

のように進めればよい。

この過程によって N 個の粒子の J ステップ後の位置を計算し，その密度を計算せよ。また，ステップ J を増やすと，密度がどの様に変化するかを調べよ。

付録 A　gfortran を用いた Fortran プログラムの実行

パソコンに Linux や FreeBSD などのフリー OS をインストールすれば，有用なアプリケーションのほとんどを無料で使うことができます。フリーの Fortran コンパイラとしては gfortran と g95 がありますが，Linux では gfortran が付属しているディストリビューションが多いようです。gfortran は Windows 版や Mac OS 版もあるので，これをお手持ちのパソコンにインストールすれば，手軽に Fortran の練習ができます。ここでは gfortran を使用したコンパイルから実行までの手順とエラーの対処法について述べます。

A.1　プログラムのコンパイルとリンクおよびその実行

Fortran プログラムを作るときには，ファイル名の最後に「.f90」を付ける。つまり，

```
文字列.f90
```

という名前にする。このファイル名の最後に付加した「ドット (.)+文字」の部分を拡張子と言う。作成したプログラムファイルを計算機で実行させるには「コンパイル」と「リンク」という二つの過程が必要である。

「コンパイル」とは，Fortran や C 言語など，人間が理解できる言語で書かれたプログラムをコンピュータが理解できる機械語に翻訳することである。コンパイルするアプリケーションを「コンパイラ」という。フリー Fortran コンパイラの代表が gfortran である。

gfortran でプログラムをコンパイルをする場合，もっとも単純には，

```
$ gfortran　プログラムファイル名
```

と入力するだけで良い。

例えば，test1.f90 というファイル名のプログラムをコンパイルするには

```
$ gfortran test1.f90
```

と入力する。コンパイル時に文法エラーなどが見つかるとエラーメッセージを出力して終了する。

gfortran コマンドは，コンパイルが成功すると，引き続きリンクを行う。プログラムはコンパイルしただけでは実行できない。コンパイラはルーチン毎の翻訳を行うだけであり，複数のルーチンを結合して OS 上で起動可能な形式にする作業までは行わないからである。また，入出力文の read や write，組み込み関数の sin や log 等は，標準ライブラリルーチンとして用意されているので，これらも結合しなければならない。このようなルーチンの結合作業をリンクという。コンパイラである gfortran 自体はリンクをする能力がないが，内部から既存のリンカを呼び出してリンクを行うことができる。

リンクにも成功すると，最後に a.out という名前のファイルが作成される。これを実行形式ファイルと言う。リンクに失敗するとやはりエラーメッセージを出力して終了する。この時 a.out は作成されない。

実行形式ファイルは，一般のコマンドのようにファイル名を入力することで，プログラムを実行する。ただし，次のように頭に「./」を付ける必要がある[19]。

```
$ ./a.out
```

プログラムが正常に動作すれば，一連の計算を行い，print 文の出力があれば画面に結果を出力した後，終了する。プログラムの中に標準入力の read 文が存在する場合には，その時点でプログラムが一時停止して，入力待ちの状態になる。そこで，必要な数値をキーボードから入力して Enter キーを押せば，計算が再開する。

以上が gfortran を用いたもっとも単純なプログラム実行までの流れであるが，上記のような単純な命令では，どんなプログラムをコンパイルしても a.out という同じ名前の実行形式ファイルになってしまうので不便である。

[19]「./」とは「現在のディレクトリにあるファイル」という意味である。使っている環境によっては不要であるが，とりあえず付加する習慣をつけよう。

また，Fortran プログラムは計算速度が重要であるから，最適化したコンパイルを行ってできるだけ高速に実行する方が良い。さらに 1.2.2 で述べたように計算機シミュレーションは基本的に倍精度実数で計算をするべきなので，自動倍精度化オプションも付加しなければならない。

このため，gfortran でプログラムをコンパイル・リンクする時は最低限以下のようなオプションを付けることをお勧めする。

```
$ gfortran -O -fdefault-real-8 test1.f90 -o test1
```

-O が最適化のオプション，-fdefault-real-8 が自動倍精度化のオプションである。また，-o のオプションの次の文字列が，実行形式ファイル名指定になる。大文字の-O と小文字の-o を間違わないように注意しよう。

この場合，コンパイルとリンクが正常に終了すると,「test1」というファイルが作成される。そこで，実行は,

```
$ ./test1
```

と入力することになる。

オプション付きの gfortran コマンドで，別のプログラムファイルをコンパイルする場合，変更するのは 2 カ所の「test1」の部分だけなので，次のような内容のスクリプトファイルを作っておけば便利である。

```
gfortran -O -fdefault-real-8 $1.f90 -o $1
```

この他，FFT や行列演算などの数値計算ライブラリやグラフィックライブラリなどを使用したプログラムをコンパイル・リンクする場合は，それらのライブラリファイルをオプションとして付加すればよい。

例えば，FFTW を含んだプログラムの場合は,

```
$ gfortran -O -fdefault-real-8 test1.f90 -o test1 -L/usr/local/lib -lfftw3
```

のように入力する。ただし，-L/usr/local/lib の部分は FFTW ライブラリの入ったディレクトリを示すオプションなので，使っている環境に応じて書き換える必要がある。

3.1 で，ファイルに出力する方法を説明したが，print 文を使って画面に表示されるテキストは，実行時にリダイレクション機能を使って表示内容を指定したファイルに出力することが可能である。例えば,

```
$ ./test1 > output.dat
```

のように，実行するコマンドの後に「>」を書き，その後に出力先のファイル名を書く。「>」は標準出力のリダイレクション先を変更するための記号である。

ただし，この方法を使うと，print 文の出力が全てファイルに保存されるため，画面には何も出てこない。実行状況を確認するために print 文を使う時には意味がないし，出力データによってファイルを切り換えることもできない。取りあえず出力ファイルを作りたい時にはよいが，完成されたプログラムにするには，プログラム中でファイルに出力するよう明示すべきである。

A.2　エラー・バグへの対処法

プログラムを開発する際の最大の問題は「エラー (error)」である。コンパイル・リンク・実行という一連の過程の中でそれぞれ「エラー」が発生する可能性がある。エラーはプログラム開発にとって付き物であり，避けては通れない。エラーの出ないよう慎重にプログラムを書くことはもちろんであるが，出たら出たでそれらに如何に素早く対処できるかが腕の見せ所である。ここではエラーをまとめて，それぞれの対処法を示す。

(1) コンパイルエラー

コンパイルとは，作成したプログラムを文法に従って機械語に翻訳することである。このため,「コンパイルエラー」とは文法的な間違いのことである。プログラムに文法的間違いがあれば，コンパイラがエラーメッセージを出力して終了する[20]。コンパイラのエラーメッセージはコンパイラによるが，gfortran では次のようなエラーメッセージが出力される。

```
test1.f90:15.7:
   do i = 1, 100
      1
Error: Symbol 'i' at (1) has no IMPLICIT type
```

1 行目はエラーのあるプログラムファイル名 (この例では，test1.f90) とエラーの位置を示している。この例では，エラーの位置が 15.7 とあるが，15 はプログラムの行番号，7 は左から数えた文字の位置を示す数値である。2 行目はエラーのあるプログラム文のコピーで，3 行目の 1 はその文のエラーが起きている位置を示す記号である。4 行目にその箇所の具体的なエラーの種類が出力されている。このメッセージを頼りにエラーを見つけてプログラムを修正すれば良い。この例のメッセージは「3 行目の 1 が示している変数 i は，暗黙の型がない」という意味であるが，具体的には i を宣言していないことが原因である。

プログラムファイルを修正したら保存して再度コンパイルする。なお，文法エラーがあるとそのエラー行が存在しない状態で引き続きコンパイルを続けるため，その波及効果で下方のプログラムがエラーになることがある。このため，エラーメッセージが多い時には全部一度にチェックしないで，ある程度修正したら再度コンパイルした方が効率が良いだろう。

プログラミングに慣れないうちは文法ミスによるコンパイルエラーが多いと思うが，エラーメッセージに従って修正すればよいのでそれほどの手間ではない。

(2) リンクエラー

リンクとは，別々に翻訳されたルーチンを結合して実行形式ファイルにする作業のことである。よって，リンクエラーは用意されていないサブルーチンや関数を使ったときに起こる。

例えば,

```
program test1
   implicit none
   real x,y
   x = 5
   y = 100
   call subr(x,y)
   print *,x,y
end program test1
```

のようなプログラムを作って，これだけを単独でコンパイル・リンクすると

[20] コンパイラが出すメッセージには Warning(警告) と Error(エラー) がある。「警告」は「このままでも実行できるけど，ひょっとすると予期しない結果が起こる可能性がある」という意味なので，必ずしも修正する必要はない。このため，メッセージが警告だけのときは強制終了せずリンクに移る。しかし文法的に間違いがなくても，作成者の意図とは違ったプログラムになっている可能性があるので，念のためにプログラムを確認する方がよいだろう。

```
/tmp/ccy3lcdo.o: In function `MAIN__':
test1.f90:(.text+0x6d): undefined reference to `subr_'
collect2: ld returned 1 exit status
```

のようなメッセージが出力される。1 行目はどこのルーチン内でエラーが起こっているのかを示していて，この例だと「MAIN」，すなわちメインプログラム中で起こっていることが示されている。2 行目にエラーの原因が書かれているが，この例では「subr という名への参照が定義されていない」とある。これは，メインプログラム中に書かれている call subr(x,y) で呼び出している subr という名のサブルーチンが用意されていないのが原因である[21]。

標準の組み込み関数が用意されていないことはまずないので，リンクエラーはプログラム中で呼び出したサブルーチンや関数の名前を書き間違えたときに起こる。例のように，エラーメッセージの中に見つからなかったサブルーチンや関数名が出るので，それに従って修正すれば良い。

なお，リンクエラーの可能性はもう一つある。それは配列名を間違えたときである。例えば

```
real a1(10),b1(10),c1(10)
integer i
do i = 1, 10
   a1(i)=b2(i)*c1(i)
enddo
```

と書くと，4 行目の b2(i) は b1(i) の間違いである。しかしこれはコンパイルエラーになるとは限らない。なぜなら，b2(i) は配列としては宣言されていないが，関数として用意されている可能性があるからだ。Fortran では配列の添字も関数の引数もかっこ ( ) を使うため区別がつかないのが原因である。リンクエラーメッセージ中に見知らぬ関数名があった場合には配列名もチェックした方がよい。ただし，implicit none で，全ての変数の型宣言をしなければならないようにしておいた場合には，b2 という文字の型宣言をしていないという理由でコンパイルエラーとなる。implicit none にすることで得られる御利益は大きいのである。

リンクエラーのメッセージにはコンパイルエラーのようなエラー行の表示は出ないが，ほとんどがサブルーチン名の書き間違いや用意忘れが原因なので，エディタの検索機能を使えば見つけることはさほど困難ではない。

(3) 実行時エラー

一番厄介なのが実行時のエラーである (こういうプログラムは「バグ (虫)」があると言う)。なぜなら，プログラムが長くなると，どこでエラーが起こっているのかを特定するのが難しいからである。実行時エラーを探すには，プログラムを詳細にチェックするしかない (この作業を「バグを取る」という意味でデバッグすると言う)。それでも，答えが合っているか否かは別として，プログラムがなんとか終了すれば良いのだが，バグによっては実行したあとでうんともすんともいわなくなってしまう場合がある。

これは無限ループに入ってしまったか，暴走したかのどちらかだと考えられる。Linux の場合，プログラムが暴走した時に実行を中断させるにはコントロールキーを押しながら C キーを押せばよい。まず間違いなく止まる。

もう一つ，以下のようなメッセージを出して強制的に終了する場合がある。

```
Segmentation fault
```

これはプログラムではなく OS が出力しているメッセージで，主として実行中のプログラムがそれ以外の実行中のプログラムのメモリ領域を破壊しようとした時に起こる。例えば，次のように配列宣言の範囲外の要素に値を代入しようとすると起こることがある[22]。

[21] メッセージ中では MAIN__とか，subr_のように名前の後ろにアンダースコアが付いているが，これはコンパイラが自動的に付加するものなので無視して良い。リンクは gfortran がするのではなく，ld という別のコマンドがするのだが，ld は OS によって異なるので，エラーメッセージは必ずしもこの例の通りとは限らない。

[22] 実際にはこの程度の簡単なプログラムでは起こるとは限らないのだが，だからといってこんなプログラムを書いてはいけない。

```
real a1(10)
integer i
do i = 1, 100000
   a1(i) = i
enddo
```

この例では，10 個しか用意されていない配列に，100000 番目まで代入しようとしているのが問題なのである。

この他，サブルーチンの引数に型の合っていない数値を与えたり，戻り値を与える引数に定数を与えるなどでも起こる可能性がある。`Segmentation fault` は実行時に問題が出た段階で起こるので，まずどこで起こっているのかを特定する必要がある。適当に `print` 文を入れて，どこまでが出力されてどこからが出力されないかを確認することで特定をしていくのが良いだろう。

しかし，このような実行時エラーはコンピュータの動作が正常でないという形で表面化するので，まだましである。原因を特定するのに時間がかかるかもしれないが，修正しなければ動かないのだから実害はない。実を言うと，一番やっかいなエラーとは，ちゃんと実行してはいるのだが計算結果が何となくおかしい，というものである。

筆者は何年も Fortran のプログラムを書いているが，実行時のエラーほど見つけにくいものはない。それでも見つかれば良い方で，場合によってはバグの存在を知らずに結果を出し，学会直前に気がついたなんてこともあった。そういう意味で大規模なシミュレーションプログラムを作るときに最も面倒なのは，長いプログラムを書くことではなく，できたプログラムの計算結果が正しいかどうかのチェックをすることである。間違いのないプログラムを書くには，忍耐と努力を惜しまず，経験を積みあげていくしかないのである。

付録 B　自動倍精度化オプション

コンピュータシミュレーションでは大量の数値を使って複雑な計算を繰り返し，それらを集積する作業を行う。このため，計算精度が重要である。Fortran で取り扱える実数には，単精度と倍精度があるが[23]，倍精度実数計算は単精度実数計算と比べてそれほど実行時間はかからない。これは最近の CPU が数値演算プロセッサのような倍精度計算用ハードウェアを装備しているからである。よって基本的に全ての計算は倍精度で行うべきである。

Fortran の仕様では，デフォルトの実数型が単精度であるため，real 宣言をした実数は単精度である。このため，基本仕様のままで倍精度計算をするには変数宣言をする時に real*8 や real(8) などの 8 バイト指定をしなければならない。

例えば，

```
real*8 x,a1(10)
```

と宣言すれば，変数 x や配列 a1 は倍精度になる。しかし，宣言文を変更するだけでは不完全である。1.0 や 1.23e5 などの実定数もそのままでは単精度なので，倍精度実定数にする必要があるからだ。以下に，単精度実定数を倍精度実定数にする変更例を記す。

```
1.23e5      →   1.23d5        ! e を d で置き換える
4.56e-15    →   4.56d-15      ! e を d で置き換える
3.14        →   3.14d0        ! 末尾に d0 をつける
1.0         →   1.0d0         ! 末尾に d0 をつける
```

1.23e5 のように指数部を付加した実定数は，e の代わりに d を使って 1.23d5 と書けば倍精度実定数になるのだから，慣れてしまえばそれほど気にはならないだろう。しかし 1.0 や 3.14 のような指数のない定数は，どちらかというと自然な書き方なので間違いに気付かない可能性がある。しかし，Fortran の仕様ではこれらはあくまでも単精度実定数であって倍精度にするには d0 を付けなければならないのである。

そもそも倍精度で計算するのを基本にするのに，常に倍精度を意識した書式を利用しなければならないのは面倒だし，間違いに気がつかない可能性もあるのだからできれば避けたいところである。そこで最近の Fortran コンパイラのほとんどが自動倍精度化オプションを装備していることを幸いに，本書では単精度実数と倍精度実数を使い分ける文法の説明は省略し，コンパイラの機能に委ねることにした。自動倍精度化オプションを付けてコンパイルすれば，1.0 や 1.23e5 はそのままで倍精度実定数を意味することになる。

以下に筆者が使っている Fortran コンパイラの自動倍精度化オプションを列記する。この他のコンパイラについてはそれぞれのマニュアルをチェックしてもらえば，大抵はあると思う。

コンピュータ	コンパイラ	ベンダー	自動倍精度化オプション
パソコン	gfortran	GNU	-fdefault-real-8
パソコン	ifort	Intel	-r8
パソコン	f95	Absoft	-N113
スパコン	sxf90	NEC	-Wf,-A dbl4

自動倍精度化はコンパイルする時にオプションを加えれば良いだけなので，プログラムを変更することから考えれば楽なものである。

[23]倍精度より精度の高い 4 倍精度というのもあるのだが，全てのコンパイラで使えるとは限らないし，ソフトウェアで実現しているので計算時間がかかる。4 倍精度は 16 バイトなので，変数の宣言は，real*16 か real(16) で行う。また，4 倍精度実定数は上記の倍精度実定数への変更において，d と書くところを q と書けば良い。もし 4 倍精度が使える環境にあり，精度が命の計算をする時には使ってみよう。

付録 C　Big Endian と Little Endian

大量のデータを保存するときは書式なし write 文 (バイナリ出力) で保存した方が良い。これは計算機のメモリに保存された内部形式そのままでファイルに保存するため，書式付き write 文 (テキスト出力) よりも出力データ量が少なくてすむからである。書式付き write 文は数字を表す文字に変換して保存するため，出力ファイルをエディタなどで直接読むことができるという利点はあるのだが，数値を文字に置き換えて出力する分，メモリ使用量が増える。

現在，コンピュータで広く使われている ASCII コードは半角 1 文字を 1 バイトで表現している。よって 15 桁の数字を出力するには 15 バイト必要で，実際には符号や小数点や指数部も出力しなければならないのでさらに増える。これに対し，倍精度実数の内部形式は 8 バイトなので，書式なし write 文で出力すれば 8 文字分ですむのである。

しかし，バイナリ出力には注意が必要である。テキスト出力ファイルは，我々でも直接読むことができるのだから，どんなコンピュータでも同じ様に読むことができる。このため，大型計算機で計算した結果をテキスト出力すれば，パソコンのプログラムで処理することに何の問題もないのだが，バイナリ出力の場合には必ずしもそうはいかない。そもそも，バイナリ形式は計算機が直接解釈して計算を実行するためのものなのだから計算機のアーキテクチャによって変わるのは当然だともいえる。しかし，現在はバイナリ形式も標準化されていて，ほとんどの計算機で整数型も実数型も同じ形式が採用されているため，バイナリ出力でも大型計算機で作成したデータをパソコンのプログラムで処理できるようになっている。

ただやはりいくつかの制約は残っている。その一つが Endian 問題である。コンピュータが扱う数値の内部形式は整数型で 4 バイト，倍精度実数型で 8 バイトだが，このバイトの並びが上位の桁からのコンピュータと，下位の桁からのコンピュータがあるのだ。前者を Big Endian，後者を Little Endian という。イメージ的に表現すれば，データが 5 桁の 10 進数で表されているとして 1234 という数字を保存すると，Big Endian では，01234 と保存されるのに対し，Little Endian は 43210 と保存されるようなものである。これらの違いは CPU のアーキテクチャによるものなのでユーザーが変更することはできない。

最近のほとんどのパソコンは，CPU に Intel かその互換チップを使っているが，これらは Little Endian である。これに対し，以前の Macintosh で使われていた Motorola の CPU や NEC のスパコンは Big Endian である。このため，スパコンの Fortran で計算して書式なし write 文で出力すると，そのままではパソコンの Fortran で読むことができない。

しかし，要はデータ入出力時の形式の問題だけなので，最近の Fortran コンパイラの多くはコンパイラオプションや環境変数などで Big から Little へ，あるいはその逆へ変換することができる機能を持っている。

例えば，NEC SX の場合にはプログラム実行前に環境変数 F_UFMTENDIAN を指定することで変換できる。

```
setenv F_UFMTENDIAN 10,20    # C シェル系
```

環境変数で指定するのは書式なし write 文の装置番号である。この例の場合，write(10) と write(20) が Little Endian で出力される。この変換を使うと，元々Big な形式を Little にするのだから若干計算時間にロスが出るのだが，バイト並びを逆にするだけだからテキスト変換よりは楽である。

これに対し，読む方で変換するやり方もある。例えば，gfortran の場合には-fconvert=big-endian というオプションを加えれば，書式なし read 文が出力ファイルを Big Endian として処理してくれる。最近の Fortran コンパイラならば大抵変換機能が付いているようなので，マニュアルで確認すれば良いだろう。

筆者はどちらかといえばスパコン側で変換する方が良いと思う。スパコンで行う計算は元々時間がかかるものであり，それに比べれば出力時のデータ変換による時間増加はわずかである。これに対してパソコン側で変換するのは時間もかかるし，変換するかどうかを問題に応じて使い分けなければならないのも面倒だからである。

ただし，スパコンでの計算には時間制限があるので，データ変換のために CPU 時間を削るのはもったいないとも考えられる。スパコンはデータ変換に時間がかかるという話もあるので，どちらが良いかはスパコン管理者に確認した方が良いだろう。