数値計算と fortran の基礎

1996 年 4 月 18 日　　初版
2007 年 6 月 11 日　　改訂
2012 年 11 月 12 日　微修正

藪 哲郎

目 次

1　はじめに

1.1　数値計算

現在、パソコンは一般家庭にまで普及し、「ネット」「文書作成」「表計算」「プレゼンテーション作成」などの用途に使われています。しかし、コンピューターはもともとは計算をするために作られたものであり、当然のことながら、パソコンは科学技術計算のために使うことが出来ます。2007 年現在では、20 万円ほど出費すれば、(ベクトルマシンや並列コンピュータなど特殊なコンピュータを使える人を除いては)、誰もがほぼ世界最高速の計算機環境を入手することが出来ます。

科学技術計算の概念について説明します。あらゆる物理現象はそれを記述する（偏微分）方程式があります。そこで、その方程式を解くことにより、物理現象をシミュレートすることが出来ます。

方程式をコンピュータで解くにはプログラミングという作業が必要です。本テキストでは数値計算の基礎とプログラミングの基礎を学びます。

1.2　プログラミング言語

プログラムを組むためには、何らかのプログラミング言語を使わなければなりません。科学技術計算用のプログラミング言語としては、C, C++, Java, Visual Basic, Matlab, fortran などが考えられます。いずれも、手続き型言語なので、文法は類似しており、一つの言語を習得すれば他の言語も容易に習得することが出来ます（C++だけは全てを理解するのはかなり難解ですが ···)。

工学では定常現象を扱うことが良くあります。例えば電気回路においては、回路に加える電圧が単一周波数の正弦波であるとき、回路の全ての場所に発生する電圧や電流は、加えた電圧と同じ周波数で振動します。電磁波の伝搬においても同様のことが言えます。そのような定常現象を扱う場合、複素記号法と呼ばれる解析法が使われます。すなわち、複素数の計算をする必要があります。先ほど述べた言語のうち、複素数を扱えるのは C++, fortran, Matlab です。それ以外の言語では複素数の足し算と引き算は何とか書けても、掛け算や割り算、複素数を引数とした sqrt, exp などの関数を扱うのは非常に困難です。

また、科学技術計算においては連立一次方程式を解くのは日常茶飯事です。連立一次方程式を解く汎用のモジュール（サブルーチン）を記述するには、整合配列（サブルーチンに渡す引数において、配列のサイズを指定できること）の機能が必要ですが、整合配列と

同等の機能を持っている言語は C++, fortran, Matlab です。

以上の 3 つの言語のうち、C++は言語使用がかなり難解であり、Matlab は値段が 1 ライセンス 30 万円近くするので誰もが使えるものではありません。そこで、古いというイメージがありますが、fortran は依然として科学技術計算の用途としては重要な言語であります。本テキストでは fortran を扱います。

2 fortran 言語の基礎 I

ここでは、fortran というプログラミング言語の基本的なことを説明します。エディタの使い方、プログラムのコンパイルの方法、実行の方法については、ここでは割愛します。また、プログラムを組む場合に暗黙の型宣言を用いることは、避けるのが常識ですが、記述の簡略化のため、本テキストでは暗黙の型宣言を用います。

2.1 代入文と write 文

さっそく、次のプログラムを入れてみて下さい。但し、プログラム中の矢印 (<----) とその右側の文字は、プログラムの意味を説明したものであり、入力する必要はありません。

```
a = 1             <---- 変数 a に 1 を代入する
b = 2             <---- 変数 b に 2 を代入する
c = a + b         <---- 変数 a と b を足した値を c に入れる
write(6,*) a,b,c <---- 変数 a b c を画面に表示する
                  <---- 空白の行は読み飛ばされる
end               <---- プログラムの終了を表す
```

プログラムの書き方　fortran のプログラムはこのように、命令文を順番に書いてゆきます。空白の行は読み飛ばされます。fortran では、**命令文は 7 桁目から書きはじめる**という決まりになっています。1 桁目から 6 桁目までは空けておいて下さい。また、このプログラムは小文字を使って書いています。fortran は大文字と小文字を区別しませんので、大文字でプログラムを書いても結構ですが、小文字の方が見やすいですし、コンピューターでは小文字が基本ですので、**小文字を使うことを推奨します**。

各々の命令文（以下、単に文と略すこともある）の意味は右に書いてあります。プログラムを入力し終わったら、実行して下さい。実行すると

```
 1.000000 2.000000 3.000000
```

と表示され、プログラムは終了します。以下に、各々の命令について解説します。

代入文　1〜3 行目は代入文と呼ばれるもので、= の右側の値を左側に代入します。

四則演算

上のプログラムで + 記号が出てきましたが、fortran では四則演算は + - * / で表します。普通の数式の場合のように * 記号を省略することは出来ませんので注意して下さい。

```
d = a b
```

と書くと、変数 a と変数 b の積ではなく変数 ab の値を d に代入してしまいます。

また、べき乗を求める時と割算の余りを求める時は次のように書きます。

```
a = 5 ** 3.5
b = mod(11,2)
```

1 行目の例は $5^{3.5}$ を a に代入し、2 行目の例は 11 / 2 の余りを b に代入します。
四則演算の優先順位は次のようになっています。

$$** \quad \Rightarrow \quad *\,,\, / \quad \Rightarrow \quad +\,,\, -$$

括弧を使う時は次のようにします。例えば、

$$a = \frac{(b+c)d+e}{2.0}$$

$$p = \{\,(q+r)s+t\}\,u$$

を fortran で計算するには次のように小括弧を重ねて書きます。

```
a = ( ( b + c ) * d + e ) / 2.0
p = ( ( q + r ) * s + t ) * u
```

(と) の小括弧の数は同じにしないとコンパイル時にエラーになります。各自、適当にプログラムを書き換えて試してみて下さい。

write 文　write は変数の内容を表示せよという命令です。プログラム中では write(6,*) となっていますが、6 は画面に出力することを表しており、* は出力形式 (何桁で出力して小数点以下何桁目まで表示するか) は fortran に任せることを意味しています。とりあえずは、**変数の内容を表示するのは** write(6,*) と覚えて下さい。2 つ以上の変数を表示したいときは、コンマ (,) で区切ってやります。また、シングルクォーテーション (') で囲んだ文字列を表示させることもできます。

次のような例を実行してみて下さい。また、自分でいろんな write 文を書いて実験してみて下さい。

```
write(6,*) a,b
write(6,*) 'a = ',a
write(6,*) 'a = ',a,'b = ',b
```

それでは、3 行目の文を変えてプログラムを次のように変えてみましょう。

```
a = 1
b = 2
a = a + b
write(6,*) a,b
end
```

実行してみると a が 3 になっていますね。= は「等号の右側を計算し、その値を左側の変数に代入する」という意味を持っています。数学での使い方とは異なりますので注意して下さい。

end 文　プログラムの最後には end 文を置いて下さい。これがないとコンパイル時にエラーが出ます。

2.2 変数の名前と型

前節のプログラムでは、a b c といった変数名を使っていました。変数名は次を制約を満たせば好きなようにつけることが出来ます。

- 1 文字目は英文字 (a,b,c,d,. . . ,z)
- 2 文字目以降は英文字か数字 (0,1,2,. . . ,9) かアンダーライン (_)
- 最大で 32 文字以内

ですから、上の規則にしたがって、出来るだけ分かりやすい名前をつける方がプログラムが読みやすくなります。

fortran で扱う変数はいくつかの型に分類されます。まずは、基本的な次の 2 つの型を覚えて下さい。

名称	性質	規則	例
整数型変数	整数値のみをとることが出来る	1 文字目が i,j,k,l,m,n で始まる	i j num max_value
実数型変数	実数値をとることが出来る	1 文字目が i,j,k,l,m,n 以外で始まる	a heikin field_level

上の表のように、1 文字目が i,j,k,l,m,n で始まるときは整数型変数で、それ以外のときは実数型変数です。次のように覚えると覚えやすいかも知れません

整数のことを İNteger と言うから i〜n で始まる数は整数。

整数型変数は数をカウントする時など、変数が整数値のみをとった方が都合が良い時に使います。実数型変数は何にでも広く使われます。

上記の表のように変数は 1 文字目が何であるかによって型が決まってしまいます。これを **暗黙の型宣言** と言います。そして、次の規則があります。

- **実数を整数型変数に代入すると、小数点以下は切り捨てられる**

例をあげてみましょう。

```
i = 2.6
write(6,*) 'i = ',i
end
```

この例は、上の規則により小数点以下が切り捨てられ、i の内容は 2 となります。

また、定数も整数型と実数型の 2 つがあります。例をあげましょう

名称	例
整数型定数	3 40 -52
実数型定数	3.4 4. -2.0

以上のように、小数点を表す **コンマ (.) があるとき実数型定数で**、**コンマがないとき整数型定数** です。

2.3 演算と型

この節の話は少々ややこしい話なので、入門段階でこのような話はしたくありません。でも、コンピュータで数値を扱うときはどの言語にも言える大切な原則なので、我慢して聞いて下さい。

fortran における四則演算の計算順序は、前々節で説明したように、普通の数学における順序と同じです。すなわち、左側から計算をしますが、

$$** \ \Rightarrow \ *, / \ \Rightarrow \ +, -$$

という優先順位があり、括弧の中は再優先で計算されます。例えば、

```
a = ( b + c * d ** e ) * f
```

においては、次のような順番で計算が行われます。

1. `d ** e`
2. `c *` (1. の計算結果)
3. `b +` (2. の計算結果)
4. (3. の計算結果) `* f`

そして、上の それぞれのステップにおいて、次の規則が適用されます。

1. 整数型の変数や定数だけを含む二項演算の結果は整数
2. 実数型の変数や定数を 1 つ以上含む二項演算の結果は実数

例をあげましょう。

```
a = 5
i = 2

b = a / i               <----- 実数／整数
c = 5 / i               <----- 整数／整数
d = 5.0 / i             <----- 実数／整数
e = 1.0 * 5 / 2         <----- 実数×整数／整数
f =  5 / 2  * 1.0       <----- 整数／整数×実数
write(6,*) b,c,d,e,f
end
```

`b,d` は規則 2. に当てはまるので 2.5 になります。`c` は規則 1. に当てはまるので 2.0 になります。`e` は `1.0 * 5` が規則 2. に当てはまるので最初の二項の計算結果が 5.0 という実数値になり、`5.0 / 2` は規則 2. に当てはまるので結果は 2.5 になります。`f` は `5 / 2` が規則 1. に当てはまるので最初の二項の計算結果が整数値 2 になり、`2 * 1.0` は規則 2. に当てはまるので、結果は 2.0 になります。

ここでの例は極端な場合であり、実際のプログラミングにおいて、**このような紛らわしい部分を作ってはいけません**。**整数/整数 のパターンのみ要注意** とだけ頭の片隅に置いて下さい。

なお、整数型の変数を実数型の数として扱いたい時は変数を `real()` で囲みます。

```
i = 1
j = 2
write(6,*) i/j            <---- 整数／整数
write(6,*) real(i)/j     <---- 実数／整数
end
```

上のプログラムの 3 行目は **整数型/整数型** なので演算結果は整数型となり 0 となります。4 行目は **実数型/整数型** なので演算結果は実数型となり 0.5 となります。

2.4 read 文

次のプログラムを入力して実行して下さい。

```
read(5,*) a <--- 数値をキーボードから入力して a に代入する
write(6,*) a
end
```

プログラムを実行すると、カーソルが点滅して入力を催促します。数値を入力して ↵ キーを押して下さい。a に数値が入力されました。このように、read 文は「キーボードから数値を入力する」ための文です。`read(5,*)` の 5 はキーボードから入力することを意味します。

では、プログラムを次のように変更して下さい。

```
read(5,*) a,b
write(6,*) a,b
end
```

この場合のように a と b にそれぞれ数値を入力する場合は、キーボードから

2 4 ↵

のように、2 つの数値を空白で区切って入力して下さい。

2.5 if 文

if 文は、「ある条件が成立していたら処理 A を実行し、そうでないなら処理 B を実行する」という機能を実現します。次のプログラムを実行してみて下さい。

```
read(5,*) a
if ( a.lt.5 ) then  <----- もし a<5 なら then 以下を実行
    write(6,*) 'a = ',a
    write(6,*) 'a < 5 '
else                <----- そうでないとき else 以下を実行
    write(6,*) 'a = ',a
    write(6,*) 'a >= 5 '
end if
write(6,*) 'program end'
end
```

この例は、if 文の練習です。if 文は if の後に括弧を書き、その中に条件を記述します。条件が満たされていれば、then の後を実行し、満たされていない場合は、else の後を実行します。そして、if 文の終りを表すため、end if 文を最後に置きます。

if 文の後の条件部の記号としては次のようなものがあります。

fortran 記号	対応する数学記号	意味
.eq.	$=$	equal (等しい)
.ne.	$\neq$	not equal (等しくない)
.lt.	$<$	less than (左側が右側より小さい)
.le.	$\leq$	less equal (左側が右側より小さいか等しい)
.gt.	$>$	greater than (左側が右側より大きい)
.ge.	$\geq$	greater equal (左側が右側より大きいか等しい)

else 以降が不要な場合は省略して次のように書きます。

```
read(5,*) a
if ( a.lt.5)  then
   write(6,*) 'a = ',a
   write(6,*) 'a < 5'
end if
write(6,*) 'program end'
end
```

この例では、文を書き始める位置を調節して、プログラムをより見やすいように工夫しています。文の前に空白を入れてもプログラムの実行には影響がありません。このように、文を書き始める位置を調節してプログラムを見やすくすることを **字下げ（インデント）** と言います。皆さんも、字下げして書く習慣をつけて下さい。

また、then 以下の命令文が 1 行だけのときは、次のように条件部の後に命令文を書くことができます。

```
read(5,*) a
if ( a.le.5 )  write(6,*) 'you input ',a
write(6,*) 'program end'
end
```

最後に、if 文の一般形をまとめて書いておきます。

```
if ( 条件部 ) then
    条件が満たされている時の処理
else
    満たされていない時の処理
end if
```

2.6 do 〜 end do 文

数値計算の分野では、繰り返し計算をする事がよくあります。決められた回数の繰り返しを実現する方法がここで学習する do 文です。do 文は必ず end do 文とペアで使います。例をあげましょう。

```
do i = 1, 4        <--- 最初に i に 1 を代入する
   write(6,*) 'i =',i
end do             <---「 i が do 文の最終値より小さい時
write(6,*) 'owari'       i を 1 増やして do 文にもどる 」
end
```

さっそく実行して下さい。do 文はまず、i に 1 を代入します。そして、end do 文に来ると、i を 1 増やして do 文に戻ります。i に 4 が入った状態で end do 文にやってくると、end do 以降の文に進みます。つまり、do 〜 end do の間を i を 1, 2, 3, 4 と増加させながら 4 回実行します。

do 文はこのように決められた回数のループを行うための文です。ところで、回数は $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \ldots$ と整数で勘定します。ですから、do 文でカウンタとして使用する変数は整数型変数（1 文字目が i,j,k,l,m,n ではじまる変数）を使うのが慣用的な使用法となっています。但し、実数型変数を使っても do 文は正常に動作します。

例題 1

1 から 10 までの整数の合計を求めるプログラムを作りなさい。
解答例を以下に示します。

```
sum = 0
do i = 1,10
   sum = sum + i
   write(6,*) 'i  sum : ',i,sum
end do
write(6,*) 'sum = ',sum
end
```

実行すると、公式 $(n+1) \times n/2$ と同じ結果が得られますね。このプログラムで、sum は合計を入れる変数です。ここで、プログラムの最初で sum に 0 を代入していることに注意して下さい。fortran では変数の初期値は不定なので、sum = 0 を怠ると正しく合計を求めることができません。

なお、do 文を次のように変えると 1 から 9 までの奇数の合計を求めることが出来ます。

```
do i = 1, 9, 2  <---- 1 〜 9 まで 2 おきに増加させる
```

do 〜 end do による繰り返しを do ループと呼びます。do ループの中に do ループが入ってくることもあります。これをネスティング (入れ子構造) といいます。次の例を実行してみて下さい。

```
do i = 1, 3                <----- 外側のループの始まり
   write(6,*) 'i =',i
   do j = 1, 4             <----- 内側のループの始まり
```

```
      write(6,*) 'i j :',i,j
   end do                  <----- 内側 (j) のループの終り
end do                     <----- 外側 (i) のループの終り
end
```

プログラムの実行順序をよく理解して下さい。今一度、do ループについてまとめておきましょう。

```
do カウンタ用変数 = 初期値 , 最終値 [ , 増分値 ]

end do
```

2.7 goto 文

fortran は文を順番に実行していきますが、実行の順番を強制的に変える方法があります。それが goto 文です。プログラム中に goto 文があると、goto で示されている文番号の位置に無条件にジャンプします。まずは、下の例を実行してみて下さい。

```
    write(6,*) 'aaa'
    write(6,*) 'bbb'
    goto 100          <----- 文番号 100 の行へ飛ぶ
    write(6,*) 'ccc'
100 write(6,*) 'ddd'  <----- 文番号 100 の行
    end
```

goto 文によって、実行の流れが変わっていることがわかりますね。プログラムの 3 行目の goto 100 は、「文番号 100 の行にジャンプしなさい」ということを意味しています。プログラムの 5 行目は 2 桁目〜5 桁目に 100 という文番号が付いています。**文番号は 2 桁目〜5 桁目に書いて下さい**。1 桁目と 6 桁目に文番号を書いてはいけません。文番号は正の数値であれば好きな値をつけることが出来ます。また、小さい順に並べる必要はありません。

do 〜 end do 文を goto 文で表すと次のようになります。

```
    i = 1                 <----- do i = 1,4 に相当
100 write(6,*) 'i =',i
```

```
      i = i + 1              <-----  「この行と次の行が
      if ( i.le.4 ) goto 100          end do に相当　」

      end
```

この例では if 文を一行で書いています。then 以下の命令が一つだけの時、このように条件部の後に、命令を書くことが出来ます。このときは end if 文はつけません。

次に、3, 6, 12, 24, 48... という数列を 1000 以下の範囲で書くプログラムを作ってみましょう。

```
      i = 3
  100 write(6,*) 'i = ',i
      i = i * 2
      if ( i.le.1000 ) goto 100
      end
```

このように、**ある条件が成立するまでループを実行し続ける**という機能を実現するためには、if 文と goto 文を組み合わせるしか方法がありません。従って、goto 文は条件が成立するまでループを実行し続けるという目的のために主に利用されます。

2.8 関数の定義

このテキストでは数値計算を主に取り扱いますから関数計算を頻繁に使います。例えば、$f(x) = 3x^2 + \sqrt{x}$ という関数を扱う場合、プログラムの中に何箇所も `3 * x ** 2 + sqrt(x)` という文字列を書く必要が生じます。これは面倒ですし、関数を変更する場合には、プログラムを何箇所も変更する必要があります。このような場合のために、fortran には文関数という機能が用意されています。

例題 2

関数 $f(x) = 3x^2 + \sqrt{x}$ の値を x が 0〜1 の範囲において 0.1 刻みで求めよ。

解答例を以下に示します。

```
      do i = 0, 10
         x = i / 10.0              <--- 10.0 に注意
         write(6,*) 'x  f(x) : ',x,f(x)
```

```
    end do
    end

    function f(x)                <--- 関数の定義 開始
    f = 3 * x ** 2 + sqrt(x)     <--- 関数名が f だから f = と書く
    return                       <--- この return は省略可能
    end                          <--- 関数の定義 終了
```

関数の定義は「function 関数名 (引数名)」という行で始めます。関数の定義はプログラムの冒頭、あるいは end の後で行います。この例では関数名は f, 引数名は x でしたが、好きな名前を付けることが出来ます（例えば myfunc(arg)）。その場合は、2 行目は、myfunc = 3 * arg ** 2 + sqrt(arg) と書きます。

また、ここでは fortran であらかじめ定義されている関数である sqrt という関数を使っています。あらかじめ定義されている関数を組み込み関数と言います。fortran の組み込み関数には次のようなものがあります。

abs(x)	$\|x\|$ を計算する
int(x)	x の整数部を取り出す
real(i)	i を実数として扱う
sqrt(x)	$\sqrt{x}$ を計算する
log(x)	$\log_e x$ を計算する
exp(x)	e^x を計算する
sin(x)	$\sin x$ を計算する
cos(x)	$\cos x$ を計算する
tan(x)	$\tan x$ を計算する

関数名にも暗黙の型宣言は適用されます。例えば、ifunc という関数は 1 文字目が i からはじまるので整数型の関数であるとみなされます。ですから、

```
    a = 1
    b = ifunc(a)
------- 中略 --------
    function ifunc(x)
    ifunc = x / 2.0
    return
    end
```

というプログラムを実行した場合、b には 1.0 / 2.0 の値（0.5）の整数部が入ります。つまり 0 となります。

2.9 まとめ

以上で、皆さんは、fortran の中の最も基本的な命令をマスターしました。2 章をまとめてみましょう。

命令	例	働き
代入文	`a = a + 1`	= の右辺の値を左辺へ代入する
`write` 文	`write(6,*) 'a : ',a`	変数の内容を表示する

`if` 文	`if (i.eq.1) then`
	`  write(6,*) 'i = 1'`
	`end if`
		条件が満たされているかどうかによって実行する文を変える

`do` 〜 `end do` 文	`do i = 1,5`
	`  write(6,*) i`
	`end do`
		決められた回数の繰り返し計算を行う

`goto` 文	`goto 100`	指定された文番号の行へジャンプする
文関数	`f(x) = x^2`	関数 $f(x) = x^2$ を定義する
`end` 文	`end`	プログラムの最後に置く

2.10 練習問題

第 2 章で覚えた命令を使うだけでも様々なことが出来ます。まずは、理解を深めるために練習問題を解いて下さい。

練習問題 1

年利 8% の複利で 100 万円を預金した。10 年間にわたって毎年の預金高を表示させるプログラムを作りなさい。

練習問題 2

次の数列を 5 項目まで求め、それらの和を求めなさい

$$\begin{aligned} C_1 &= 3 \\ C_n &= 4\,C_{n-1} + 3 \end{aligned}$$

練習問題 3

2, 4, 8, 16, 32,. . . という数列のうち 1000 以下の数を表示し、その和を求めなさい。

練習問題 4

100 までの数のうち素数を求めるプログラムを作成しなさい。ある数 a が b で割り切れるかどうかを調べる方法として次の 2 つがあります。

1. a/b と int(a/b) が同じかどうか
2. mod(a,b) の値が 0 かどうか

ここでは、1. の方法を用いてプログラムを作成しなさい。

練習問題 5

ユークリッドの互除法を用いて最大公約数を求めるプログラムを作成しなさい。但し、余りを求める命令 mod を使いなさい。

3　方程式の解

3.1　はじめに

工学では、方程式の解を求めるという問題が頻繁に発生します。高校数学における解を求める問題は、必ず解が存在し、技巧的な方法（例えば因数分解）で解けました。しかし、実際の問題はそうでない場合がほとんどです。例えば、次の方程式の解を見つけるという問題は、解析的には解けません（この場合は偶然に $x = 1$ という解が分かっていますが……）。

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

この問題は次の関数

$$f(x) = -\exp\left\{-(x-1)^2\right\} + \log x + \sqrt{x} \tag{2}$$

において $f(x) = 0$ となる x を見つける問題と同じです。ちなみに、この関数の形は図 1 のようになります。

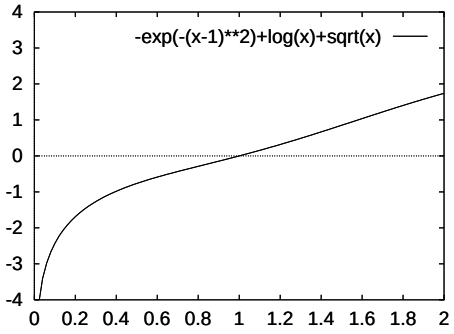

図 1　関数 $f(x)$ の図

3.2　原始的な方法

図 1 より、関数の定義域は $x > 0$ であり、滑らかな形をしていることがわかります。従って、例えば、$x = 0.01$ からスタートして 0.01 きざみに x を増加させてゆき、$f(x)$ の符号が変わったところでストップすれば、解が求まることが予想できます。

関数値 $f(a)$ と $f(b)$ の符号が同じかどうかは次のようにすれば簡単に判断することが出来ます。

$$f(a)\cdot f(b)=\begin{cases} \text{負} & : \ f(a)\text{ と }f(b)\text{ は異符号} \\ 0 & : \ f(a)\text{ と }f(b)\text{ のうち 1 つ以上が 0} \\ \text{正} & : \ f(a)\text{ と }f(b)\text{ は同符号} \end{cases} \tag{3}$$

練習問題 6

上の関数の解を求めるプログラムを作成し、解きなさい。

上の例は原始的な方法でした。次に、スマートな方法を 2 つ紹介しましょう。

3.3 二分法

図 1 の関数の場合、グラフより 0.1 ～ 2 の間に解が 1 つだけ存在することが分かります。このように、**ある区間に解が 1 つだけ存在することが分かっているとき**、次のようなアルゴリズムで解を求めることが出来ます。解が 1 つだけ存在する区間を (a, b) で表します。

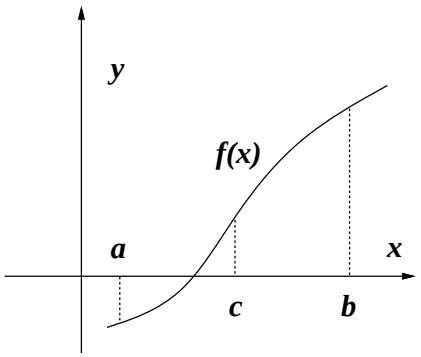

図 2　二分法の図解

二分法のアルゴリズム

1. 区間 (a, b) の端点における関数値を、それぞれ $f(a), f(b)$ とする。区間内に一つだけ解があるので、$f(a)$ と $f(b)$ の符号は逆である。

2. 中点 $c = (a + b)/2$ をとり、関数値 $f(c)$ を求める

3. $f(a)$ と $f(c)$ が同符号なら解は区間 (c, b) にあるので、(c, b) を 新しい (a, b) とする。異符号なら解は区間 (a, c) にあるので、(a, c) を新しい (a, b) とする

4. 区間 (a, b) の間隔が ε (収束判定の基準) より小さい場合は終了する。そうでない場合は 2. に戻る。

それでは、さっそくプログラムを作ってみましょう。

```
      function f(x)
      f = - exp( -(x-1)**2 ) + log(x) + sqrt(x)
      return
      end

      eps = 0.001
      a = 0
      b = 1.5

100   write(6,*) 'a b : ',a,b
      c = ( a + b ) / 2
      if ( f(a)*f(c).lt.0 ) then
          b = c
      else
          a = c
      end if
      if (  b-a .gt.eps ) goto 100
      write(6,*) 'solution : ',( a + b ) / 2

      end
```

ある区間 (a, b) に 1 つだけ解があることが分かっている時、二分法は確実に解が見つけられる大変強力な方法です。皆さんも、方程式の解を求める問題に直面したときは、二分法を思い出して下さい。

練習問題 7

方程式

$$x^3 + x - 1 = 0 \tag{4}$$

は 0 〜 1 の間に 1 つの解を持つ。解を求めよ。

3.4　ニュートン法

先に上げた二分法は 1 回の反復で探索区間を 1/2 に減らすということを繰り返すので、収束するまでに何回も関数を評価する必要があります。前節であげた例題のように、関数値が一瞬で求められる場合はいいのですが、現実問題としては、関数値を 1 回求めるために、何十分もかかる計算が必要とされることは珍しくありません。

関数が $f(x) = 0$ 付近で単調減少（増加）であり、初期探索点と解の区間においては関数が単調減少（増加）であることが分かっている場合、二分法よりもはるかに少ない反復回数で解を求められる方法があります。それがニュートン法です。図で説明しましょう。

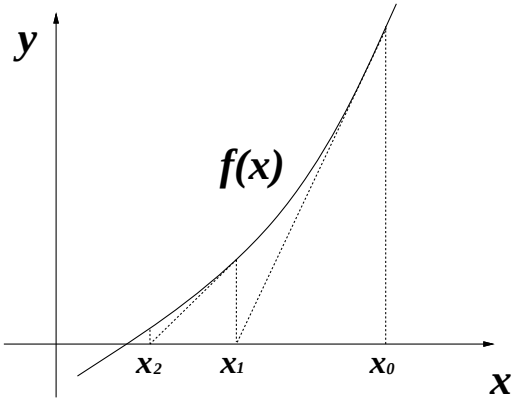

図 3　ニュートン法の図解

x_0 を初期探索点とします。x_0 での接線を引きます。接線と x 軸との交点を次の探索点 x_1 とします。次に x_1 での接線を引き、接線と x 軸との交点を次の探索点 x_2 とします。以下、同じことを繰り返すと、x_0 , x_1 , x_2 , $\cdots\cdots$ は解に収束してゆきます。アルゴリズムは次のようになります。

ニュートン法のアルゴリズム

1. 探索点における接線を求める
2. 接線と x 軸の交点を次の探索点とする
3. 前回の探索点と現在の探索点の距離が ε（収束判定の基準）より小さいとき終了。そうでないとき 1. へ戻る。

ニュートン法は、**探索点近傍を 1 次式で近似したときの解の位置へジャンプする** ということを繰り返すことにより、探索点を解へ収束させる方法です。ニュートン法を使うに

は探索点における傾き $f'(x)$ を知る必要があります。関数 $f(x)$ の傾き $f'(x)$ が解析的に求まる場合はその値を用います。$f'(x)$ が解析的に求まらない場合は、Δx として適当な小さな値を設定し、

$$\frac{f(x+\Delta x)-f(x)}{\Delta x} \tag{5}$$

の値を使います。

ニュートン法において、探索点 x_i での関数値が y_i で、傾きが a のとき、次の探索点 x_{i+1} の位置を求める公式を導きます。接線は次式で表されます。

$$(y-y_i)=a(x-x_i) \tag{6}$$

$y=0$ となる x を求めます (y に 0 を代入し、x について解く)。

$$\begin{aligned} a(x-x_i) &= -y_i \\ x-x_i &= -\frac{y_i}{a} \\ x &= x_i-\frac{y_i}{a} \quad \Rightarrow \quad x_{i+1} \end{aligned} \tag{7}$$

すなわち、新しい探索点 x_{i+1} は上式で表されます。それでは、実際に次の関数をニュートン法を使って解いてみましょう。

例題 3

$\sqrt{a}$ は関数 $f(x)=x^2-a$ の解として求められる。ニュートン法を使って $\sqrt{2}$ を求めなさい。但し、初期値としては 2 を用いなさい。

解答例を示します。

```
      eps = 0.0001
      x = 初期値
100   write(6,*) 'x : ',x
      y = f(x)
      a = g(x)
      x2 = - y / a + x
      if ( abs(x2-x).gt. eps )  then
         x = x2
         goto 100
      end if
      write(6,*) 'solution : ',x
      end
```

```
function f(x)
f = 関数値
return
end

function g(x)
g = 導関数値
return
end
```

上のプログラムでは、収束したかどうかを判断するために、前回の探索点と現在の探索点の距離を用いています。そして、距離を求めるために、絶対値をとる関数 abs を用いています。

練習問題 8

先ほどの例題と同じ方法で $\sqrt{3}$、$\sqrt{5}$ を求めなさい。但し、関数の傾きは解析的に解かず、$\dfrac{f(x+\Delta x)-f(x)}{\Delta x}$ を使いなさい。

図 3 からわかるように、ニュートン法は、非常に収束が速く、二分法より遥かに少ない反復回数で方程式を解くことが出来ます。しかし、その反面、初期探索点と解の区間において関数値が単調減少（増加）でなければならないという制約があります。

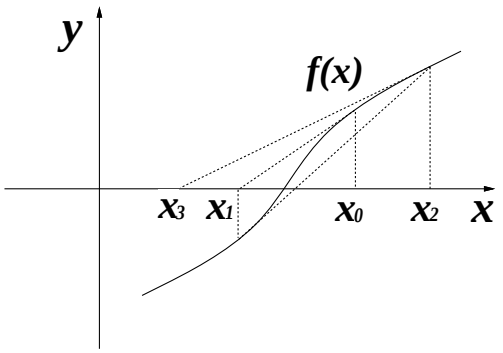

図 4　ニュートン法で解けない場合

例えば図 4 のように、関数が単調減少（増加）でなく途中に変曲点がある場合、探索点 x_0 , x_1 , x_2 , $\cdots\cdots$ が発散してしまい、解を求めることが出来ません。

しかし、実際の問題として、関数が単調減少（増加）であり、おおよその解があらかじめ分かっている場合は少なくありません。

例として光導波路の伝搬定数を求める問題があります。光導波路の正規化周波数を v とするとき、伝搬定数 b は次の方程式の解として与えられます。

$$f(b) \ = \ v\sqrt{1-b} \ - \ \tan^{-1}\sqrt{\frac{b}{1-b}} \tag{8}$$

そして、分散曲線を求めることは次の問題を解くことと等価です。

> 正規化周波数 v を 1 〜 10 まで 0.1 刻みに変化させた時の、各々の伝搬定数 b を求めよ

今、$v = 1$ のときの解が求まっていると仮定します。$v = 1.1$ のときの解は、$v = 1$ のときの解と非常に近い値をとることが予想されます。ですから、$v = 1$ のときの解を初期値としてニュートン法を使うことにより $v = 1.1$ のときの解を求めることができます。

このように、方程式を構成する係数を徐々に変えてゆき、そのときの方程式の解を求めるという問題の場合は、あらかじめ解のおおよその値が分かっていますから、ニュートン法を使うことにより効率よく解を求めることができます。

この他にも、関数の極小値を求める問題を解く場合にもニュートン法の考え方を応用することが出来ます。関数の極小値を探すことは、言い換えると導関数が 0 となる点を探すことと同じです。これは導関数という方程式の解を求めるのと同じです。従って、ニュートン法を応用して解くことが出来ます。このように、ニュートン法の考え方は大切ですから、よく理解して下さい。

4　微分方程式

4.1　はじめに

物理現象の多くは微分方程式や偏微分方程式で表現されます。例えば、自由落下は加速度が一定ですから、下向き方向を x とするとき、$\ddot{x} = g$ という式で表されます。バネにつながれたおもりの振動現象は、おもりの加速度とバネによる力が釣り合うわけですから、重りの変位を x とするとき $\ddot{x} + kx = 0$ です。余談ですが、波動は $\frac{\partial^2 \phi}{\partial x^2} - \frac{1}{c}\frac{\partial^2 \phi}{\partial t^2} = 0$ という双曲型の偏微分方程式で表され、熱伝導は $\frac{\partial^2 \phi}{\partial x^2} - \frac{1}{c}\frac{\partial \phi}{\partial t} = 0$ という放物型の偏微分方程式で表されます。

電気系の学科では、電気回路という教科がありますが、回路現象は非常に美しい体系を持っています。つまり、

$$\begin{array}{lrcl} \text{抵抗では} & v & = & Ri \\ \text{コンデンサでは} & C\dfrac{dv}{dt} & = & i \\ \text{コイルでは} & L\dfrac{di}{dt} & = & v \end{array} \tag{9}$$

という関係が電圧と電流の間に成り立ちます。従って、電気回路の問題は、微分方程式に帰着します。

また、自然界をモデル化するときにも微分方程式は使われます。例えば、生態系における各生物の個体数の変化は連立の微分方程式で表すことが出来ます。兎と狼の 2 種だけの生物がいると仮定してモデリングする場合を考えます。兎の数を x_1 で表し、狼の数を x_2 で表します。兎の数の変化率 $\frac{dx_1}{dt}$ は兎の数と天敵である狼の数によって定まり、狼の数の変化率 $\frac{dx_2}{dt}$ は狼の数と餌である兎の数によって定まりますから、

$$\frac{dx_1}{dt} = ax_1 - bx_1x_2 \tag{10a}$$

$$\frac{dx_2}{dt} = cx_1x_2 - dx_2 \tag{10b}$$

のような連立の微分方程式が成り立ちます。このように、私たちの身の回りにある現象は、微分方程式によって、簡潔に記述されるのです。

高校数学では微分方程式の解き方として求積法などの解析的な方法を学んできましたが、微分方程式の中で、解析的に解ける問題は稀な部類に属します。そこで、数値計算を使うことになります。

微分方程式は次のように分類することが出来ます。

1. 1 変数による 1 階の微分方程式

   例 ： $\frac{dx}{dt} + x = 0$

2. 1 変数による 2 階以上の微分方程式

   例 ： $\frac{d^2x}{dt^2} + \frac{dx}{dt} \cdot x + x = 0$

3. 多変数による 1 階の微分方程式

   例 ： $\begin{array}{l} \frac{dx_1}{dt} = x_1 x_2 \\ \frac{dx_2}{dt} = -x_1 \end{array}$

4. 多変数による 2 階以上の微分方程式

なお、ここからは、変数 x や x_1 , x_2 は時刻 t の関数であるとして議論を進めていきます。上で 4 つのタイプの微分方程式を見たわけですが、いずれのタイプの微分方程式も全て次の形に変形することが出来ます。

$$\begin{array}{lll} \frac{dx_1}{dt} & = & f_1(x_1, x_2, \cdots\cdots, x_n, t) \\ \frac{dx_2}{dt} & = & f_2(x_1, x_2, \cdots\cdots, x_n, t) \\ \frac{dx_3}{dt} & = & f_3(x_1, x_2, \cdots\cdots, x_n, t) \\ \cdots\cdots & \cdots\cdots & \cdots\cdots \\ \cdots\cdots & \cdots\cdots & \cdots\cdots \end{array} \tag{11}$$

つまり、微分方程式は、変数 x_1 , x_2 , $\cdots\cdots$ の変化率（傾き）を表していると考えることが出来ます。そして、傾きは通常、x_1 , x_2 , $\cdots\cdots$, x_n によって与えられます。物理現象においては、傾きを表す式の中に t が入ってくる（例えば xt のような項）ことは珍しい部類に属します。t が入ってくるということは、時間に依存する項があるということで、これは、時間的に変動する外力が加えられる（交流電源を接続するとか、ものを揺らすとか）ことを意味します。はじめに例として、バネにつながれたおもりの振動現象を表す式 $\frac{d^2x}{dt^2} + kx = 0$ をあげましたが、この式の中には t はありません。

4.2 オイラー法

まずは、1 変数による 1 階の微分方程式を解く方法について説明します。この微分方程式を数値的に解く方法はたくさん存在しますが、その中で最も分かりやすく、基本的な方法がオイラー法です。次の微分方程式を解くことを考えます。

$$\frac{dx}{dt} = x \tag{12}$$

初期値は $t = 0$ のとき $x = 1$

解析解は $x = \exp(t)$

この式 (12) における変数 x は位置を表していると考えてみましょう。式 (12) は位置 x における変化率（速度）は現在の位置 x によって定まることを意味しています。しかも、位置が大きくなるにつれて速度も増大しますから、どんどん速度が大きくなるような現象を表していることが想像できます。

さて、一階の微分方程式は、現在の x と t が分かれば、その時点での x の変化率、つまり傾きが分かるというものでした。つまり、微分方程式は傾きを表していると考えることが出来ます。

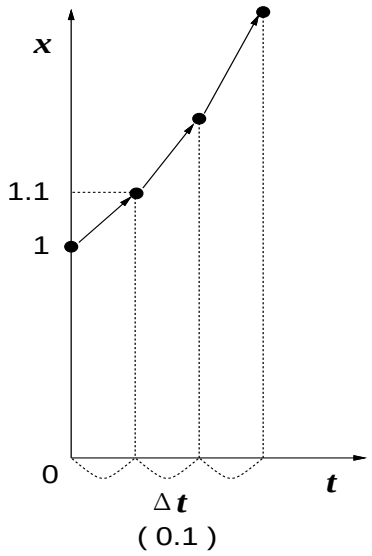

図 5　オイラー法

従って、図 5 のような作業を繰り返すと、解に近いものが得られると思われます。まず、$t = 0$ において、$x = 1$ という初期値が与えられています。式 (12) ($\frac{dx}{dt} = x$) より $t = 0$ での傾きは 1 です。ですから、傾き 1 で Δt だけ進みます。$\Delta t = 0.1$ とすると、$t = 0.1$ での x の値は 1.1 です。次に、ここでの傾きは式 (12) より 1.1 ですから、傾き 1.1 で更に Δt だけ進みます。だから、$t = 0.2$ における x の値は 1.21 です。これを繰り返すことにより解に近いものが得られることが予想されます。このような、単純作業の繰り返しはコンピューターが最も得意とする分野です。

上の作業で微分方程式を解いた時の誤差について考えます。関数は、テーラー展開することにより、いくらでも精度よく近似することが出来ます。t_i における関数値が x_i のとき、$t_i + \Delta t$ における関数値 $x_i + \Delta x$ はテーラー展開の公式により次のようになります。

$$x_i + \Delta x = x_i + x'(t_i)\Delta t + x''(t_i)\frac{(\Delta t)^2}{2!} + x'''(t_i)\frac{(\Delta t)^3}{3!} + \ldots\ldots \tag{13}$$

従って、

$$\Delta x = x'(t_i)\Delta t + x''(t_i)\frac{(\Delta t)^2}{2!} + x'''(t_i)\frac{(\Delta t)^3}{3!} + \ldots\ldots \tag{14}$$

微分方程式は $x' = f(x,t)$ という形式で与えられますので、上式の 1 次の項は計算可能です。$x' = f(x,t)$ を何回も微分することにより 2 次の項や 3 次の項も計算することは可能ですが、厄介です。ですから、1 次の項までで近似を打ち切ることにした場合が、オイラー法です。従って、2 次以下の項で表される量が誤差となります。

このように、関数をテーラー展開し、2 次以下の項を無視して近似する方法をオイラー法と言います。単純で分かりやすく、たくましさを持っている方法なので、微分方程式を数値計算するときの基礎といえます。今の方法を $\Delta t = 0.1$ としてオイラー法で解いてみましょう。但し、t の範囲は $0 \sim 4$ とします。

```
      dxdt(x,t) = x
      t = 0
      x = 1
      dt = 0.01

 100  write(6,*) 't   calc  rigid : ',t,x,exp(t)
      dx = dxdt(x,t) * dt
      t = t + dt
      x = x + dx
      if ( t .le. 4 ) goto 100
      end
```

今回、解いた微分方程式は厳密解 $x = \exp(t)$ が存在します。従って、厳密解と数値計算によって得られた解とを比べることにより、プログラムが正しいかどうかを検証することができます。また、数値計算によって得られた解の精度を見ることも出来ます。`write` 文で厳密解と数値計算によって求めた解を併記しています。`dt` を変えると、どのように精度が変わるかを自分で確かめて下さい。

オイラー法は、2 次以下の項を無視する方法でした。これを、1 次の公式といいます。それに対して、2 次や 3 次の項まで使う方法もあります。2 次の公式として改良オイラー法、3 次の公式としてルンゲクッタ法があります。ルンゲクッタ法は非常に精度が高いため、微分方程式を数値計算で解く場合の定番的な解法となっています。但し、理論的にかなり難しくなりますので、このテキストでは省略します。

4.3 連立微分方程式

前節では、変数は 1 つだけであり、1 階の微分方程式の場合について学習しました。しかし、実際の問題としては、次のような連立の微分方程式を扱うことの方がはるかに多いといえます。

$$\dot{x}_1 = f_1(x_1, x_2, x_3, t) \tag{15a}$$

$$\dot{x}_2 = f_2(x_1, x_2, x_3, t) \tag{15b}$$

$$\dot{x}_3 = f_3(x_1, x_2, x_3, t) \tag{15c}$$

また、1 変数による 2 階の微分方程式は 1 階の微分方程式 2 つが連立しているように変換して解きます。1 変数による 3 階の微分方程式は 1 階の微分方程式 3 つが連立しているように変換して解きます。

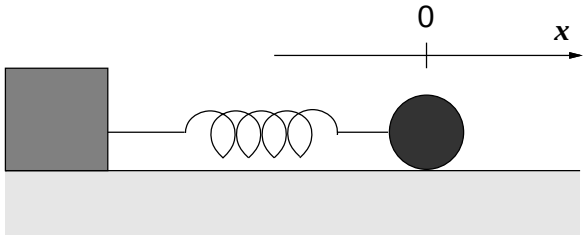

図 6　バネにつながれたおもりの振動

それでは、図 6 に示すように、バネにつながれたおもり振動を表す次の 2 階の微分方程式について考えてみましょう。

$$\ddot{x}_1 + kx_1 = 0 \tag{16}$$

x_1 の 1 次微分を表す x_2 という量を新たに導入します。

$$\dot{x}_1 = x_2 \tag{17a}$$

$$\ddot{x}_1 = \dot{x}_2 \tag{17b}$$

具体的には、x_1 はバネにつながれたおもりの位置を表し、x_2 はおもりの速度を表します。式 (16) の中の $\dot{x}_1$ を $\dot{x}_2$ に置き換えると次式が得られます。

$$\dot{x}_2 + kx_1 = 0 \tag{18}$$

この式と、式 (17a) をあわせて書くと次のようになります。

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{19}$$

それでは、この連立微分方程式を $k = 1$ のときについて解いてみましょう。但し、初期条件として

$$t = 0 \text{ のとき } x_1 = 1\ ,\ x_2 = 0 \tag{20}$$

が与えられていると仮定します。これは、時刻 $t = 0$ でおもりの位置が $x_1 = 1$ のところで静止した状態から手を離すことを意味します。もう一度、式を書き直します。

$$\frac{dx_1}{dt} = x_2 \tag{21a}$$

$$\frac{dx_2}{dt} = -x_1 \tag{21b}$$

初期値 $t = 0$ にて $x_1 = 1\ ,\ x_2 = 0$

解析解は $x_1 = \cos(t)\ ,\ x_2 = -\sin(t)$

これは、x_1 の値によって x_2 の傾きが決まり、x_2 の値によって x_1 の傾きが決まるという、たすきがけのような形をしています。このように変形することにより、前節で習ったオイラー法を適用することが出来ます。プログラムを作成すると次のようになります。

```
      x1 = 1
      x2 = 0
      dt = 0.1
      t  = 0

 100  write(6,*) 'x1  x2  : ',x1,cos(t),x2,-sin(t)
      dx1 =  dxdt1(x1,x2,t) * dt
      dx2 =  dxdt2(x1,x2,t) * dt
      x1 = x1 + dx1
      x2 = x2 + dx2
      t = t + dt
      if ( t .le. 10 ) goto 100
```

```
end

function dxdt1(x1,x2,t)
dxdt1 = x2
return
end

function dxdt2(x1,x2,t)
dxdt2 = -x1
return
end
```

この場合は、解析解が存在しましたが、解析的に解けないような場合でも、このような数値計算を行うことにより、連立の微分方程式を解くことが出来ます。

練習問題 9

次の微分方程式をオイラー法を用いて $t = 0 \sim 4$ の範囲で解け。

$$\ddot{x} + \dot{x} + x = 0 \tag{22}$$

初期条件：$t = 0$ において $x = 1$, $\dot{x} = 0$

5 fortran 言語の基礎 II

ここでは、今までに使っていた命令に加えて、2 つの新しい概念を勉強します。配列とサブルーチンです。

5.1 配列変数

まずは、配列から勉強しましょう。10 個のデータを読み込んで、変数に格納し、その合計とかけ算の和を求めるプログラムを作ることを考えます。普通に考えると以下のようになります。

```
read(5,*) a1
read(5,*) a2
read(5,*) a3
read(5,*) a4
read(5,*) a5
.....
.....
sum = a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10
mul = a1 * a2 * a3 * a4 * a5 * a6 * a7 * a8 * a9 * a10
write(6,*) 'sum = ',sum
write(6,*) 'mul = ',mul
end
```

これでは、データの数が 100 個になったりすると大変です。また、データの数が変わるたびにプログラムを大幅に書き換えないといけません。そこで、配列変数というものを使います。例を示しましょう。

```
dimension a(3)
a(1) = 2
a(2) = 4
a(3) = 6
write(6,*) a(1)
write(6,*) a(2)
write(6,*) a(3)
end
```

このように、dimension a(3) と宣言すると、a(1),a(2),a(3) の 3 つの配列変数を使うことが出来ます。図でイメージすると図 7 のようになります。

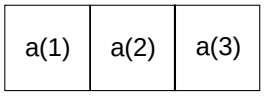

図 7　1 次元配列

配列の便利なところは、

```
dimension a(3)
do i = 1, 3
   a(i) = i * 2
end do
do i = 1, 3
   write(6,*) a(i)
end do
end
```

のように、配列の括弧の中（添字という）を変数にすることが出来るからです。ですから、n 個のデータを読み込み、その合計とかけ算の値を求めるプログラムは次のようになります。

```
dimension a(100)  <---------- あらかじめ多い目にとっておく
read(5,*) n
do i = 1, n
   read(5,*) a(i)
end do
sum = 0
mul = 1
do i = 1, n
   sum = sum + a(i)
   mul = mul * a(i)
end do
write(6,*) 'sum  mul : ',sum,mul
end
```

配列を使うことにより、とても便利になっていることがわかりますね。なお、**添字としては慣用的に整数型変数を使います**。実数型変数を使ってもプログラムは正常に動作しますが、できるだけ整数型変数を使うようにして下さい。

このような括弧の中の添字が 1 つの配列を 1 次元配列と言います。2 次元の配列も同様に作成することが出来ます。例えば、

```
dimension a(2,3)
```

と宣言すると、図 8 に示すように a(1,1),a(1,2),a(1,3),a(2,1),a(2,2),a(2,3) の 6 つの配列変数を使うことが出来ます。

a(1,1)	a(1,2)	a(1,3)
a(2,1)	a(2,2)	a(2,3)

図 8　2 次元配列

2 次元配列を使ったプログラムの例を示します。

```
dimension a(2,3)

do i = 1, 2
    do j = 1, 3
       a(i,j) = i*10 + j
    end do
end do

do i = 1, 2
    do j = 1, 3
       write(6,*) 'i j a(i,j) ',i,j,a(i,j)
    end do
end do
end
```

連立方程式を解く時などはこのような 2 次元の配列が必要になります。

練習問題 10

10 人の成績があります。それぞれ次の通りです。

名簿番号	得点	名簿番号	得点
1	45	6	53
2	56	7	85
3	54	8	49
4	20	9	45
5	82	10	99

この得点を配列に代入し、最高点、最低点、平均点、標準偏差を求めるプログラムを作成しようとして、途中まで作成したら、次のようになった。

```
dimension a(10)
n = 10
a(1) = 45
a(2) = 56
a(3) = 54
```

続きを完成させなさい。

5.2 サブルーチン

次の例題を考えます。

例題 4

配列 a(5) に 5 人分の国語の成績、b(5) に 5 人分の数学の成績が入っています。国語、数学の平均を求めなさい。

```
dimension a(5),b(5)
n = 5
a(1) = 1
a(2) = 2
a(3) = 3
a(4) = 4
a(5) = 5
b(1) = 2
b(2) = 4
b(3) = 6
b(4) = 8
b(5) = 10
```

これまでに習った知識を利用してプログラムを作成すると、次のようになります。

```
sum = 0
do i = 1,n
   sum = sum + a(i)
end do
ave = sum / n
write(6,*) 'average of japanese ',ave

sum = 0
do i = 1,n
   sum = sum + b(i)
end do
ave = sum / n
write(6,*) 'average of mathematics ',ave
end
```

ここで、プログラムの中に似た部分があります。これを、1 つにまとめることが出来たなら、プログラムが簡潔になることが予想されます。その方法として、サブルーチンという方法があります。まず、最も簡単な例を見てみましょう。

```
write(6,*) 'aaa'
call sample   <---- 「sample という名前のサブルーチン
write(6,*) 'ccc'      へジャンプする                  」
end

subroutine sample <---- sample という名前のサブルーチン
write(6,*) 'bbb'
return            <---- call で呼ばれた文の次の行へ戻る
end               <---- サブルーチンの終りを表す
```

実行してみて下さい。call sample は sample という名前のサブルーチンに飛びなさいという意味を持ちます。subroutine sample はこの行から sample という名前のサブルーチンを記述することを表しています。return 文はサブルーチンを呼んだ call 文の次の行から実行を再開しなさいという意味を持ちます。また、サブルーチンの終わりには end 文をつけて下さい。

では、次のプログラムはどうなるでしょうか。

```
a = 1
write(6,*) 'in main a = ',a
call sample2(a)     <------   「a を引数としてサブルーチ
write(6,*) 'in main a = ',a     ン sample2 をコールする 」
end

subroutine sample2(b) <--- 引数をうけとり b と呼ぶ
write(6,*) 'in sub b = ',b
return
end
```

call sample2(a) のように call 文の後に書いたサブルーチン名の後に括弧で変数が入っています。これを **引数** と呼び、サブルーチン sample2 に対して引数を渡すことを意味しています。また、サブルーチン側にも subroutine sample2(b) のようにサブルーチン名の後に括弧で変数が入っています。これも **引数** と呼び、引数を受けとり b と呼ぶことを意味しています。

実行してみると分かりますが、メインルーチンにおける変数 a が、サブルーチンからみると b となって認識されることがわかります。return 文の前に b = 2 という行を加えてみて下さい。メインプログラムに戻ってみると a の値が 2 に変わっていることが分かります。このように、メインルーチンの変数 a とサブルーチンの変数 b は結合され、サブルーチンで変数の値を変更すると、メインルーチンにも影響が及びます。

ここで大切なのは **メインルーチン側とサブルーチン側とで引数の型を合わせる** ということです。この場合はメインルーチンから実数型引数 a を渡し、サブルーチン側で実数型引数 b を受けとっています。もし、

```
subroutine sample2(i)
write(6,*) 'in sub i = ',i
return
end
```

とやると、i には間違った値が入ってしまいます。

次に、キーボードから入力した数の階乗を計算するプログラムをサブルーチンを用いて作ってみましょう。

```
read(5,*) n
call fact(n,n2) <--- n! を求め n2 に入れるサブルーチンを呼ぶ
```

```
write(6,*) 'n = ',n,'   n! = ',n2
end

subroutine fact(k,k2)  <---- k! を計算し k2 に入れる
i = 1
do j = 1,k
   i = i * j
end do
k2 = i
return
end
```

ここでは引数が 2 つあります。このように引数が複数個ある場合は順番に結合されます。すなわち、n と k が結合され、n2 と k2 が結合されます。サブルーチン fact は引数 n を受けとって $n!$ を計算し、その値を n2 に返すという機能を持っています。

いちばんはじめの例題のプログラムをサブルーチンを使って書き直します。つまり、配列と配列の要素数の 2 つを引数として与えると、その平均値を求めるサブルーチンを作ります。

```
call cal_average(a,n, ave)
write(6,*) 'average of japanese ',ave
call cal_average(b,n, ave)
write(6,*) 'average of japanese ',ave
end

subroutine cal_average(a,n, ave) <--「配列を受けとって
dimension a(n)                          その平均を求める」
sum = 0
do i = 1, n
   sum = sum + a(i)
end do
ave = sum / n
return
end
```

ここでは、引数が 3 つ使われています。引数が複数個ある場合は、順番に結合されます。ここでは、**引数として配列を指定しました**。call 文において引数として配列を指定するときは、このように配列名から括弧の部分を取り除いたものを指定します。サブルーチン側

でも同様の指定を行います。但し、**サブルーチン側でも** `dimension` **宣言** をし、引数が配列であることを宣言しなくてはなりません。サブルーチン側で行う `dimension` 宣言では、要素数のところに何を書いても構いません。その理由は、引数として渡されるのは配列 `a` が格納されている領域の先頭アドレスだからです。ここでは、分かりやすくするため `a(n)` と書きましたが、`a(*)` あるいは `a(1)` などと書いてもプログラムの動作は同じです。

このようにサブルーチンを使うことによりプログラムを簡潔に分かりやすく書くことができます。また、プログラムの再利用が容易になります。例えば、あなたが別のプログラムを作っているときに、配列データの平均を求める必要があったとします。その時は、今回作成したサブルーチンをそのままコピーして使えばよいのです。こうすることにより、素早くプログラムを作成することができ、間違いも少なくなります。

大規模なプログラム開発では、まずプログラムの設計という作業をします。そこでは、プログラムが果たすべき機能をうまく分割し、それぞれの機能をサブルーチンで実現します。皆さんも、将来、卒業研究などで大きなプログラムを組むときは、プログラムの設計にも注意を払って下さい。

練習問題 11

二項係数は次のような式で表されます。

$$ {}_mC_n = \frac{m!}{n! \cdot (m-n)!} \tag{23} $$

`m,n` を入力すると二項係数 `keisuu` を返すサブルーチン

```
subroutine nikou(m,n, keisuu)
```

を作成しなさい。

5.3 まとめ

配列と サブルーチンについてまとめておきます。

配列の使い方

1 次元配列

```
dimension a(10)
```

上の宣言により、a(1),a(2),a(3),.....,a(10) の計 10 個の配列変数を使うことができる。

2 次元配列

```
dimension b(2,3)
```

上の宣言により、b(1,1),b(1,2),b(1,3),b(2,1),b(2,2),b(2,3) の計 2 × 3 個の配列変数を使うことができる。

subroutine の使い方

サブルーチンの呼び方

```
call サブルーチン名 (引数 1, 引数 2,...)
```

サブルーチンの書き方

```
subroutine サブルーチン名 (引数 1, 引数 2,...)

return
end
```

6　連立方程式

6.1　はじめに

連立方程式を扱う方法として二つの方法があります。一つ目は記号的に解く方法で、もう一つは数値的に解く方法です。まず、記号的に解く例を示します。下のような連立方程式を考えます。

$$\begin{bmatrix} a & b \\ b & a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a^2+b^2 \\ a^2+b^2 \end{bmatrix} \tag{24}$$

この連立方程式を x, y について解くと、次のようになります。

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a-b \\ a+b \end{bmatrix} \tag{25}$$

次に数値的に連立方程式を解く例を示します。

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix} \tag{26}$$

この連立方程式を解くと、解は次のようになります。

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{27}$$

このように、連立方程式を扱う方法として 2 つの方法があります。問題によっては記号的に解くことが必要な場合もありますし、数値的に解くことが必要な場合もあります。ここでは、数値的に解く方法のみについて説明します。

6.2　ガウスの消去法

以下のような連立方程式について考えます。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{28}$$

1 行目に $\dfrac{a_{21}}{a_{11}}$ をかけた式を作ると次の式が得られます。

$$a_{21}x_1 + a_{12}\frac{a_{21}}{a_{11}}x_2 + a_{13}\frac{a_{21}}{a_{11}}x_3 + \frac{a_{14}a_{21}}{a_{11}}x_4 = b_1\frac{a_{21}}{a_{11}} \tag{29}$$

2 行目から上式を引くと、a_{21} を 0 にすることができます。この操作を施しても、連立方程式の解は変わりません。1 行目に $\frac{a_{31}}{a_{11}}$ をかけた式を作り、3 行目から引くと、a_{31} を 0 にすることができます。4 行目にも同様の処理と行うと、下のように変形することが出来ます。なお、a'_{22} は $a_{22} - a_{21}/a_{11}$ を表しています。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{30}$$

上式において、2 行目に $\frac{a'_{32}}{a'_{22}}$ をかけた式を作成し、3 行目から引くと、a'_{32} を 0 にすることができます。4 行目にも同様の処理を施すことにより次のように変形することが出来ます。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b''_4 \end{bmatrix} \tag{31}$$

これを繰り返すことにより、次のような形に変形することが出来ます。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \\ b'''_4 \end{bmatrix} \tag{32}$$

この形を上三角行列と呼びます。こうすると、解は

$$x_4 = \frac{b'''_4}{a'''_{44}} \tag{33}$$

$$x_3 = \frac{b''_3 - a''_{34}x_4}{a''_{33}} \tag{34}$$

$$x_2 = \frac{b'_2 - a'_{23}x_3 - a'_{24}x_4}{a'_{22}} \tag{35}$$

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4}{a_{11}} \tag{36}$$

のように、順次代入していくことにより、求めることが出来ます。これが、ガウスの消去法と呼ばれる方法で、連立方程式を解く最もポピュラーな方法として広く用いられています。

練習問題 12

連立方程式をガウスの消去法で解くプログラムを作成し、以下の連立方程式を解きなさい。

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 93 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{37}$$

6.3 ガウス・ザイデル法

この方法は、反復計算を繰り返すことにより連立方程式を求める方法です。下の連立方程式について考えます。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{38}$$

まず、$x_1 \sim x_4$ を適当な値に設定します (例えば $x_1 = x_2 = x_3 = x_4 = 0$)。次に、1 行目を利用して x_1 を修正します。1 行目を取り出すと、次のようになります。

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = b_1 \tag{39}$$

これを x_1 について解くと

$$x_1 = -\frac{a_{12}x_2 + a_{13}x_3 + a_{14}x_4 - b_1}{a_{11}} \tag{40}$$

となり、この式を利用して、$x_2 \sim x_4$ より x_1 を修正します。次に、2 行目を使って $x_1, x_3 \sim x_4$ より x_2 を修正します。

$$x_2 = -\frac{a_{21}x_1 + a_{23}x_3 + a_{24}x_4 - b_2}{a_{22}} \tag{41}$$

同様に、以下の式により x_3, x_4 を修正します。

$$x_3 = -\frac{a_{31}x_1 + a_{32}x_2 + a_{34}x_4 - b_3}{a_{33}} \tag{42}$$

$$x_4 = -\frac{a_{41}x_1 + a_{42}x_2 + a_{44}x_4 - b_4}{a_{44}} \tag{43}$$

これを一回の反復とします。この操作を何回も反復することにより、$x_1 \sim x_4$ を解に収束させることが出来ます。

しかし、注意すべきことはいつでもこの方法が使えるわけではないことです。収束条件があります。収束のための十分条件は a_{11} , a_{22} , a_{33} , a_{44} という対角要素が、同じ行の他の係数の和より大きいことです。式で表すと、次のようになります。

$$|a_{11}| > |a_{12}| + |a_{13}| + |a_{14}| \tag{44}$$

$$|a_{22}| > |a_{21}| + |a_{23}| + |a_{24}| \tag{45}$$

$$|a_{33}| > |a_{31}| + |a_{32}| + |a_{34}| \tag{46}$$

$$|a_{44}| > |a_{41}| + |a_{42}| + |a_{43}| \tag{47}$$

これを満たしていない場合は、満たすように行や列を入れ換えることを試みます。それでも、満たすことが出来ない場合はガウス・ザイデル法を使って方程式を解く保証はありません（但し、これを満たしていなくても収束することはあります)。これを見ると、この方法は使い道が少ないように思えますが、そうではありません。工学において、問題（例えば微分方程式）を連立一次方程式に帰着させて解く場合は多々ありますが、このとき出来る連立一次方程式の係数行列は、対角要素が大きくなる場合が非常に多いのです。

従って、このガウス・ザイデル法は、行列のサイズが大きく、対角項の係数が大きい行列を解く場合によく用いられます。また、対角要素の値が大きい方が速く収束します。

以上のことを、図を通じて感覚的に理解してみましょう。次の問題を解くことを考えます。

$$\begin{bmatrix} -2 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -4 \\ 8 \end{bmatrix} \tag{48}$$

この連立方程式のそれぞれの行は直線を表しています。ですから、連立方程式を解くことは、次の二つの直線の交点を求めることと同じです。

$$y = 2x - 4 \tag{49}$$

$$y = -\frac{x}{4} + 2 \tag{50}$$

初期値として $x = 0$, $y = -1$ を与えたとき、図の矢印で示した経路に沿って収束してゆきます。

練習問題 13

連立方程式をガウス・ザイデル法で解くプログラムを作成し、以下の連立方程式を解きなさい。

$$\begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \\ 5 \end{bmatrix} \tag{51}$$

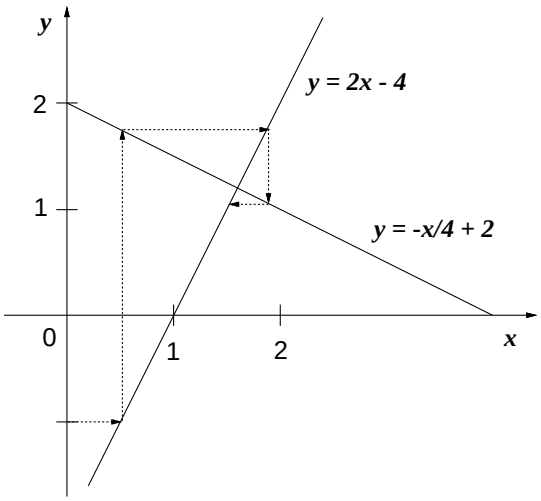

図 9　ガウス・ザイデル法

7　積分

7.1　はじめに

高校数学の積分問題は、紙と鉛筆を使って解析的に解く問題が出題されます。そこでは、置換積分や部分積分などのテクニックが必要でした。そのようなテクニックを使っても解析的に解けない場合は、数値的に解くことになります。

次の関数を区間 $(-2, 3)$ にわたって積分することを考えます。

$$f(x) = \exp\{1 - x - \exp(-x)\} \tag{52}$$

この関数は電離層における電子の密度分布を表す関数でチャップマン型関数と呼ばれ、図 10 のような形をしています。x 軸が高度に関係する量で y 軸が電子密度を表しています。この積分を解析的に解くことはできません。

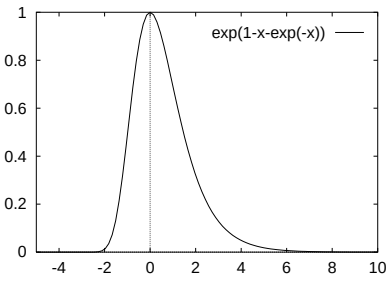

図 10　チャップマン型関数

7.2　台形公式

積分を数値的に解く最も簡単な方法として、積分を台形の面積の和で近似する方法があります。図 11 を見て下さい。積分の対象となる面積を n 個の台形の面積の和で表しています。斜線で示した 1 つの台形の面積 a は

$$a = \frac{(y_i + y_{i+1})\Delta x}{2} \tag{53}$$

で表されます。従って、台形全体の面積 A は

$$\begin{aligned} A &= \frac{(y_1 + y_2)\Delta x}{2} + \frac{(y_2 + y_3)\Delta x}{2} + \cdots\cdots\cdots + \frac{(y_{n-1} + y_n)\Delta x}{2} \\ &= \frac{\Delta x}{2}\left(y_1 + 2y_2 + 2y_3 + \cdots\cdots + 2y_{n-1} + y_n\right) \end{aligned} \tag{54}$$

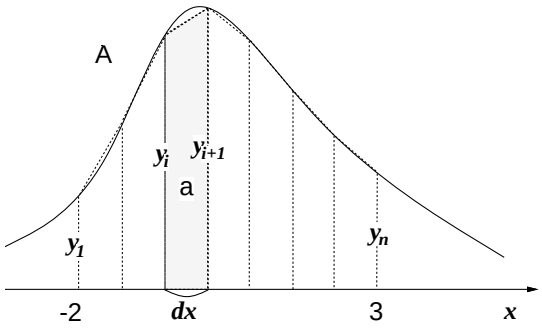

図 11　台形公式

で表されます。

練習問題 14

上の関数の積分を行おうとしてプログラムを作りはじめた。プログラムは、積分区間における関数値 y_i をあらかじめ配列に入れておき、サブルーチンで積分計算を行うという方針で、途中まで完成させた。続きを完成させなさい。

```
dimension y(101)          <------ 積分区間を 100 分割
n = 101                           だから節点数は 101 個
begin = -2
end   =  3
region = end - begin
dx = region / ( n-1 )
do i = 1, 101
   ytmp = begin + (i-1) * dx
   y(i) = chap( ytmp )
end do
call daikei(y,n,dx,   sekibun)
write(6,*) 'integral : ',sekibun
end

function chap(x)
chap = exp(1-x-exp(-x))
return
end

subroutine daikei(y,n,dx,  sekibun)
```

```
dimension y(*)
```

8 fortran 言語の基礎 III

ここではさらに fortran を使いこなすための雑多な知識を習得します。

8.1 コメント文

プログラム中に注釈を書き込むとプログラムが読みやすくなります。次の数列を計算するプログラムを作成します。

$$A_1 = 1 \tag{55a}$$

$$A_2 = 1 \tag{55b}$$

$$A_n = A_{n-1} + A_{n-2} \tag{55c}$$

プログラムは次のようになります。

```
c  初期値の設定と最初の 2 項の表示

      a_2 = 1                ! A n-2 と A n-1 を設定
      a_1 = 1
      write(6,*) 'a1 : ',a_2 ! 最初の 2 項を表示
      write(6,*) 'a2 : ',a_1

*  3 〜 10 項目の計算

      do i = 3,10
         a = a_1 + a_2             ! 漸化式
         write(6,*) 'a',i,' : ',a  ! 表示
         a_1 = a                   ! 更新
         a_2 = a_1
      end do

      end
```

プログラム中にコメントを書いているので、プログラムが読みやすくなっているのがわかります。fortran ではコメントを書く方法が 2 つあります。

- 1 桁目に c あるいは * が書いてある行はコメント行とみなす。
- ! 記号より右側はコメントが書いてあるとみなす

皆さんも、プログラムに適切なコメントを入れることにより読みやすいプログラムを書くよう心がけて下さい。

8.2 continue 文

continue 文は何もしない文です。文番号と併用して次のように用います。

```
      i = 1
 100  continue
         i = i + 1
         write(6,*) i
      if ( i.le.5 ) goto 100
      write(6,*) 'owari'
      end
```

continue 文を入れることによりプログラムが読みやすくなっていることが分かります。また、i = i + 1 の手前に i = i * 2 を入れたくなった場合、行を挿入するだけで済みます。もし、

```
  100 i = i + 1
```

と書いておいた場合、i = i * 2 を挿入するときに文番号を付け換える必要があり、間違いを犯しやすくなります。

8.3 変数の型宣言

fortran では、一文字目が i,j,k,l,m,n で始まる変数は整数型であり、それ以外の変数は実数型でした。これを **暗黙の型宣言**と言いました。ある程度の規模のプログラムを組む場合、暗黙の型宣言は用いてはいけません。その理由は、タイプミスによるバグが混入する可能性があるからと、実数型変数の精度が不足するからです。例えば、

```
      sum_value = 0
      do i = 1, 10
         sum_value = sum_vaule + i
      end
```

のようなプログラムがあったとします。タイプミスがあるため、プログラムは正常に実行され、終了しますが、変数 sum_value の中身はプログラマが意図したのとは異なる値が入っています。もし、変数の宣言が必須であったなら、コンパイル時に「sum_vaule という変数は未定義である」というエラーが出るので、タイプミスに気がつくことが出来ます。実際は以下のようにプログラムを組みます。

```
implicit none     <--- この文を入れると変数の宣言が必須になる

integer i,j,k     <--- 整数型変数（4 byte）の宣言
real    a,b       <--- 4 バイト実数の宣言（通常は使わない）
real*8  c,d       <--- 8 バイト実数の宣言

integer in(10)    <--- 整数型変数の配列宣言
real*8  r(10)     <--- 8 バイト実数の配列宣言

c = 1.1d0         <--- 8 バイト実数定数
```

fortran が扱える実数型は 4 バイト実数（値を格納するのに 4 バイトを消費する）と 8 バイト実数（同 8 バイト）があります。暗黙の型宣言による実数型変数は 4 バイト実数であり、4 バイト実数の精度は有効数字 6 桁程度です。一方、8 バイト実数の有効数字は 16 桁程度なので、通常は 8 バイト実数を用います。配列の宣言は dimension a(10) の代わりに real*8 a(10) のようにします。

また、8 バイトの精度を持たせたい定数は最後に d0 を付加します（d は double precision の略）。1.0 あるいは 1.5 のような数は二進数で割り切れる数なので、1.5 と 1.5d0 は同じ量を表しますが、1.1 は二進数では循環小数となる数値なので 1.1 と 1.1d0 は異なる値となります。

さらに、fortran において、数学関数は引数の型によってその精度が変わります。たとえば、acos(-1.0) と acos(-1.0d0) はどちらも π を返しますが、その精度は acos(-1.0d0) が有効数字 16 桁であるのに対して、acos(-1.0) は有効数字が 6 桁となります。

余談ですが、C 言語においては、double で宣言した実数は 8 バイト実数であり、sin, cos などの数学関数は、関数プロトタイプ機能により、引数として何を与えても、引数が 8 バイト実数に変換されてから処理され、返り値は 8 バイト実数の精度を持っています。

8.4 配列の write 文

配列変数を表示する時の方法として次のような方法があります。

```
dimension a(5)
do i = 1, 5
   a(i) = i
end do
write(6,*) a             <-----(1)
write(6,*) (a(i),i=2,4) <-----(2)
end
```

上のプログラムの (1) の write(6,*) a では配列 a の全ての要素を表示します。つまり、(1) は

```
write(6,*) (a(i),i=1,5)
```

と等価です。(2) は配列 a の 2,3,4 の要素を表示します。do ループと類似した書き方となっています。

8.5 文字型変数

これまでに皆さんは **整数型、実数型** の 2 つのデータ型を学習しました。ここで新しく、文字型を学習しましょう。

```
character a*40,b*10,c*80,d(5)*1
a = 'koreha mojiretu da. nagaiyoooo.'
b = a
c = a
write(6,*) a
write(6,*) b
write(6,*) c
if ( a.eq.c ) then
    write(6,*) 'a = c'
end if
end
```

プログラムの 1 行目が文字型変数の宣言部です。

```
character 文字型変数名 * 長さ , ......
```

の形式で宣言します。上の例では長さ 40 の文字型変数 a と長さ 10 の文字型変数 b と長さ 80 の文字型変数 c を宣言しています。

文字型変数に文字型定数を代入する時は、上の例のように

```
文字型変数 = ’文字列’
```

の形で代入します。’文字列’ の長さより文字型変数の長さが短いときは、超過する部分は切り捨てられます。ですから、上のプログラムの b = a の部分では切り捨てが生じます。b が長さ 10 の文字型変数なので a の 11 文字目以降は切り捨てられます。

また、if 文で文字列の比較を行っていますが、比較する文字列の長さが異なる時は、短い方の文字列の後に空白が必要な個数だけ補われ、長い方の文字列と同じ長さにしてから、比較を行います。

文字列同士の比較は .eq. か .ne. しか使えません。

例題 5

入力された数値 ($65535 = 2^8$ より小さいと仮定する) を 2 進数に変換するプログラムを作成しなさい。

解答例を示します。

```
character c(8)*1

read(5,*) num
do i = 8, 1, -1
   if ( num .ge. 2**(i-1) ) then
      c(i) = ’o’
      num = num - 2**(i-1)
   else
      c(i) = ’.’
   end if
end do

write(6,*) (c(i),i=1,8)
end
```

8.6 write 文における書式設定

これまでは、数値や文字を表示させる命令として、write(6,*) を使ってきました。この方法は便利なのですが、表示が見にくいというデメリットがあります。これに対して、「4 桁の整数で表示する」「10 桁で表示し、小数点以下を 3 桁に設定する」などのように表示形式を細かく指定する方法があります。このために * の代わりに、書式設定を書きます。例を見て下さい。

```
a = 1234.5678
write(6,*) '-------------------------'
write(6,'(f9.2)') a
write(6,'(e9.2)') a
end
```

これまでの例では * を書いていた部分に、文字型定数を使って書式を書きます。書式全体を ( ) で囲みます。実数型の変数に対しては以下の 2 つの形式が使えます。

f 型編集　f 型編集は 1234.57 のような表記法で数値を表示することを表します。f9.2 は全体を 9 桁、小数点以下がを 2 桁で表示します。

e 型編集　e 型編集は 0.12e+02 のように指数形式で数値を表示することを表します。0.12×10^2 を表しています。e9.2 は全体を 9 桁、仮数部の小数点以下を 2 桁で表示します。

小数点以下の最も小さい桁は四捨五入されます。

整数型の変数は次のようにします。

```
i = 123
write(6,*) '-------------------------'
write(6,'(i6)') i
end
```

i 型編集　i 型編集は整数を表示することを表し、i6 は 6 桁で表示することを表します。

文字型の変数は次のように表示します。

```
character  a*10
a = '1234567890'
write(6,*) '-------------------'
write(6,'(a5)') a
write(6,'(a)') a
end
```

文字型変数を format 文で整形する必要なほとんどない と言えます。ですから「文字型のときは a を指定する」と覚えておけばよいと思います。次のように組み合わせて使います。

```
character*5 moji
moji = ' a = '
a = 12.34567
write(6,'(a,f6.2)') 'a : ',a
write(6,'(a,f6.2)') moji,a
end
```

表示する数値と数値の間を空けるために、x 型編集というのもあります。

```
a = 1234.5678
b = 9876.5432
write(6,'(f8.2,2x,f8.2') a,b
write(6,'(f8.2,a,f8.2)') a,'  ',b
write(6,'(2(f8.2,2x))') a,b
end
```

上の 1 つめの write 文と 2 つめの write 文は等価です。3 つめの write 文は'(f8.2,2x,f8.2,2x)' と等価です。

8.7 if 文 II

例えば、以下の処理を if 文で書く場合を考えます。

$$\begin{cases} a<0 & : \quad b=-1 \\ a=0 & : \quad b=0 \\ a>0 & : \quad b=1 \end{cases}$$

if 文を入れ子にすることにより、書くことが出来ます。素直に書くと、例えば次のようになります。

```
if ( a .lt. 0 ) then
   b = -1
else
   if ( a .eq. 0 ) then
      b = 0
   else
      b = -1
   end if
end if
```

これを以下のように簡潔に書くことが出来ます。

```
if ( a .lt. 0 ) then
   b = -1
else if ( a .eq. 0 ) then
   b = 0
else
   b = -1
end if
```

以下のように else if を多数書くこともできます。

```
if ( i .eq. 0 ) then
   .....
else if ( i .eq. 1 ) then
   .....
else if ( i .eq. 2 ) then
   .....
else
   .....
end if
```

次の場合を考えます。

1. i=1 かつ j=1 のとき、k=1。
2. i=1 または j=1 のとき、k=1。

それぞれ、以下のように書けます。

```
if ( ( i.eq.1 ).and.( j.eq.1 ) ) then
   k = 1
end if
   .....
if ( ( i.eq.1 ).or.( j.eq.1 ) ) then
   k = 1
end if
```

この場合は、.eq. は .and. や .or. より優先されるので、1 つめの if 文は if (i.eq.1 .and. j.eq.1) と書いても同じですが、括弧を付けた方が分かりやすいと思います。

8.8 整合配列

サブルーチンに 2 次元配列を渡すときの書き方を考えます。この例では 4 行 5 列です。
以下のように書きます。

```
dimension a(3,2)
```

```
......
call sub(a,3,2)
......
end

subroutine sub(a,isize,jsize)
dimension a(isize,jsize)
.......
return
end
```

このように、サブルーチン側での配列宣言において行のサイズと列のサイズを指定します。fortran の場合、2 次元配列の格納順序は、上の 3 行 2 列の行列の場合、a(1,1) → a(2,1) → a(3,1) → a(1,2) → a(2,2) → a(3,2) という順番になります。本当に必要なのは 3 行であるという情報なので、サブルーチン側での配列宣言は a(isize,*) あるいは a(isize,1) などと書いてもプログラムは同じ動作をします。

これを整合配列と呼びます。C 言語にはこの機能がないので、数値計算の汎用的なプログラムを書くのは困難です。C++ の場合は、行列を表すクラスを自作するかどこかから取ってくることになりますが、コピーコンストラクタの必要性など、ある程度のプログラミングの知識が必要となります。