

Chapter 3

数列とプログラミング基礎

本章では、数列を扱いながら for ループという、プログラミングの重要な要素の一つである「繰り返し」をマスターする。

3.1 数列

数列とは、数がある規則に則って並んだものである。数列の長さ (個数のことを言う) は、 n など任意の自然数で表現される。つまり、無限につづくことがある。

まず、「素直な」数列を考えてみよう。

$$A = 1, 2, 3, 4, \dots$$

この数字の並びの規則は、

$$a_k = k$$

と表現していい。4の次に5が来ない(9とか)可能性もあるが、数列は、そのルールを表現しうるだけの数の並びを表現しなくてはならない。

次に、「ひねくれた」数列を考えてみよう

$$B = 1, 2, 3, 4, 29, \dots$$

これを表現するには、多くの可能性があるが、例えば

$$b_k = (k-1)(k-2)(k-3)(k-4) + k$$

という意地の悪い規則である。

さて、まず簡単な規則で作られる数列の性質を考えよう。

$$c_k = 2k - 1$$

は、奇数を表現している。(では偶数は?)

3.1.1 Octave による数列の表現

Octave では、数列は有限なベクトルの一種として表現するか、ルールを関数の形で定義する。

```
a=[2,4,6,8,10,12,14,16];
```

は 8 個の数からなり、関数

```
function y=An(n)
    y=round(n)*2;
endfunction
```

は (n 番目の) 偶数を返す。 n が実数でも構わないが、 `round` を用いて強制的に整数にしている。なお、関数とプログラムの違いに関しては、3.3 章 (pp.10) を参照のこと

さて、関数 A_n を利用して、実際に数列を作ってみよう。まず、 A_n のプログラムを “an.m” という名前で保存しよう。(ディレクトリに注意)

```
octave-3.0.0:45> source("an.m"); #an.m を読み込む ここでは関数を定義しただけなので、何も実行されない。
```

```
octave-3.0.0:46> A(1,1)=An(1)
```

```
A = 2
```

```
octave-3.0.0:47> A(1,2)=An(2)
```

```
A =
```

```
2 4
```

```
octave-3.0.0:48> A(1,3)=An(3)
```

```
A =
```

```
2 4 6
```

```
octave-3.0.0:49> A(1,4)=An(4)
```

```
A =
```

```
2 4 6 8
```

```
octave-3.0.0:50>
```

数列が伸びてゆく様子がわかるが、何かがおかしい。 計算機とは決まりきった作業を実行するためにつくられた機械である。機械に実行させてその楽さを実感してみ

よう。同じ作業を繰り返すときに便利なのが、for ループと呼ばれる繰り返し命令である。for()...endfor の間を、for の () 内に書いた条件が成立している間繰り返すものである。とりあえず、for(i=1:n) と書くと、i という変数を 1 から n まで増加させながら n 回繰り返すものと覚えておけばよい。

```
octave-3.0.0:50> source("an.m"); #以前一度実行していれば、ここでは不要。
```

```
octave-3.0.0:51> for(i=1:10) A(1,i)=An(i); endfor
```

```
octave-3.0.0:52> A
```

```
A =
```

```
    2    4    6    8   10   12   14   16   18   20
```

10 だけでなく別の数を代入すれば、任意の長さの数列を作ることができる。

3.1.2 数列の和

数列の和を求めることが必要になることも多い。例えば、平均を求めるときなどである。10 個の要素からなる列ベクトル X の和を求めることにしよう。以下のプログラムを "sum1.m" という名前を付けて保存する。

```
#sum1.m
X=[1;2;3;4;5;6;7;8;9;10];
sum=0;
for(i=1:10)
    sum=sum+X(i,1);
endfor
sum
```

プログラムを保存したら、Octave で実行する。

```
Octave:> source("sum1.m")
```

source() というのは、() 内に与えられたプログラムを実行する関数である。

なお、プログラムを変えたら、その度にエディタで保存し、source し直すこと

しかし、これでは、X という決められたベクトルの和しか計算できない。どんなサイズのものでも計算したい。同じプログラムを何度も実行したり、プログラムの中から呼び出すためには関数を定義する。一度関数を定義して読み込む (source コマンドを実行する) と、Octave に組み込まれている関数と同様に、再度読み込みせずに何度も実行することが出来る。関数とプログラムの違いに関しては章末にまとめた。

どんなサイズのものでも和を計算するためには、引数として与えられたベクトルや行列のサイズを知る必要がある。Octave のもっている `size()` という関数は、ベクトルや行列の型を調べる関数である。これを利用して以下のプログラムを下に示した。sum2.m という名前で保存しよう。

```
#sum2.m
function sum=sum2(X)
    [rows, cols]=size(X);
    sum=0;
    for(i=1:rows)
        sum = sum+X(i,1);
    endfor
endfunction
```

先頭の#はコメントを表す。この行は実行されないなので、メモを残すのに利用出来る

これではどうだろうか？

```
Octave:> X=[1;2;3;4;5;6;7;8;9];
Octave:> source("sum2.m")
Octave:> sum2(X)
ans = 55
```

source しただけでは実行されないことに注意。その次の行で関数を呼び出して実行している。

良さそうである。しかし、 $X=[1,2; 3,4; 5,6; \dots]$ など、列が複数ある場合には対応できない。各行において、列ごとに和をとることを考えよう。その場合には for ループを入れ子にした 2 重ループというものを使う。sum3.m としよう。

```
#sum3.m
function sum=sum3(X)
    [rows,cols]=size(X);
    sum=zeros(1,cols);
    for(i=1:rows)
        for(j=1:cols)
            sum(1,j)=sum(1,j)+X(i,j);
        endfor
    endfor
endfunction
```

注意しなくてはならないのは、`sum` は `rows×cols` 回実行されることである。また、`i` のループと `j` のループは、外側と内側の関係にあり、はみ出すようなループを作ることは出来ないということである。

ついでに、平均を計算するプログラム `avg1.m` を作ろう。

```
#avg1.m
function a=avg1(X)
    [rows,cols]=size(X);
    a=zeros(1,cols);
    for(i=1:rows)
        for(j=1:cols)
            a(1,j)=a(1,j)+X(i,j);
        endfor
    endfor
    for(j=1:cols)
        a(1,j)=a(1,j)/rows;
    endfor
endfunction
```

数列を複数同時に扱うのは、数学よりも、実験などで複数の変数を同時に観測する場合などが一般的である。ここでは、`series1.m` というプログラムで 100 個からなる数列を複数作成することにする。

```
#series1.m
n=100;
cols=3;
a=3;
X=zeros(n,cols);
for(i=1:n)
    for(j=1:cols)
        X(i,j)=a*i+(j-1);
    endfor
endfor
```

`avg1` と `sum3` により平均と和を求めよ。Octave の関数である `mean()`, `sum()` の結果と比較せよ。また、和の公式 (3.2 節参照) と比較せよ。

練習問題

`avg1.m` では、`j=1:cols` のループが、2 回登場している。何故だろう？

3.1.3 行列の計算

多重ループの応用として、行列の積を計算するプログラムを作ってみよう。

プログラム mat1.m

```
#mat1.m
function C=mat1(A,B)
[row1,col1]=size(A);
[row2,col2]=size(B);
if(col1 != row2)
    printf("type mismatch.\n");
    return;
endif
C=zeros(row1,col2);
for(i=1:row1)
    for(j=1:col2)
        for(k=1:col1)
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
        endfor
    endfor
endfor
endfunction
```

Octave の関数では数学的な関数よりもプログラムに近く、if 文を入れることが可能である。これを用いて 5-8 行目で型のチェックをしている。型が合わない場合は return 文により値を返さずに終了している。if 文とは、if() という表現のカッコの内側の式を評価し、式が論理的に真であれば if() ... endif の間の文を実行するものである。これを条件分岐と呼ぶ。

ただし、Octave 上で計算するのは、計算時間がかかるためにお勧めしない。可能な限り用意されている関数を使うべきである。これら Octave に組み込まれた関数は、高速で実行するための工夫がなされている。C 言語等で読者が組むプログラムを実行するより Octave 上で処理する方が高速である場合が多い。

練習問題

適当な行列 A,B をつくって上のプログラム mat1(A,B) と Octave での行列の積 AB の結果を比較せよ。

3.2 数列再び

再び数列に話をもどす。数列とは、ある規則に則って求められた数の並びのことである。ここでいう規則とは、

$$a_k = 3k - 2 \quad (3.1)$$

であるとか、

$$b_k = 4^k - 3 \quad (3.2)$$

のように、 k の関数として表されるものを指す。3.1 のような形式で表された数列を等差数列、3.2 のような形式で表された数列を等比数列と呼ぶ。

数列の和は、以下の公式を使って求めることができる。

$$\sum_{k=1}^n k = \frac{1}{2}n(n+1) \quad (3.3)$$

$$\sum_{k=1}^n k^2 = \frac{1}{6}n(n+1)(2n+1) \quad (3.4)$$

$$\sum_{k=1}^n r^k = \frac{r(r^n - 1)}{r - 1} \quad (3.5)$$

上2式は分散を求めるところでも触れたので詳細は略す。一番下の式は、 $0 < r < 1$ のとき、 $n \rightarrow \infty$ である値に収束する。 $(r > 1$ なら発散する)。(ここでいう、ある値とは?)

3.2.1 数列の応用

数列の応用として、金利を考えてみる。

例えば、年利 5% で 100 万円借り入れたとき、2 年 (48 ヶ月) 以内に返済するには毎月いくら返せばよいだろう。金利は、元金の残高だけではなく、発生した利子に対しても発生する、複利計算とする。また、月利は年利の $1/12$ とする (本来は 12 乗根である筈だが、業界の慣例である)。また、残額に関しては元金と金利を区別しないことにする (全額が返済されない場合の残額の精算には必要であるが、ここでは考慮しない)。

金融計算は、毎月の返済が元本を減らすためにやや複雑である。前月末の融資残高を x_n とすると、今月の残高 x_{n+1} は、

$$x_{n+1} = x_n * (1 + a) - p \quad (3.6)$$

ただし a は金利、 p は返済額である。これも一種の数列である。このように、数列の定義の右辺に n の関数ではなく、以前の値 x_n がある場合、これを漸化式と呼ぶ。

また、返済が可能であるためには、 $x_{n+1} < x_n$ でないといけない。例えば、 $p = 0$ とすると等比数列になるので、残高が指数的に膨らんでしまう。つまり、

$$p > ax_n \quad (3.7)$$

である必要がある。
 なお、上の漸化式は

$$x_{n+1} - \frac{p}{a} = (1+a)\left(x_n - \frac{p}{a}\right) \quad (3.8)$$

と変形すると等比数列と見なすことが出来る。ここではそれを考えずに、漸化式をそのまま計算することを考える。プログラム money1.m は、ローン額 L を万円単位で、毎月いくら定額で支払えば何ヶ月で支払いが可能かを調べるプログラムである。支払額は x として、万円単位で与える。

```
#money1.m
#parameters
L=100;
M=48;
RY=0.05;
RM=RY/12;

#x=5;

#inital condition
loan=L;
sum=0;

#main loop
for(t=1:M)
    loan=loan*(1+RM);
    loan=loan-x;
    # printf("%d %f\n",t,loan);
    sum=sum+x;
    loan2(1,t)=loan;
    if (loan < 0)
        back=loan*-1;
        printf("payback complete in %d months. Balance=%f\n",t,back);
        sum=sum+loan;
        break;
```

```
endif
endfor
if(loan>0)
    printf("payback not complete in %d months\n",M);
else
    sum
    total_interest = sum-L
endif
#plot(loan2)
```

実行例

```
octave:84> x=3;
octave:85> source("money1.m")
payback complete in 36 months. Balance=0.112783
sum = 107.89
total_interest = 7.8872
```

なお、あまり長期のローンを組むのはおすすめしないので、48ヶ月で終了する設定になっている。sum は全支払額、total_interest はそのうちの金利分である。残高が減ると、36ヶ月でも、約8万円分の金利しか払っていないことになる。

3.2.2 レポート問題 6

1. 上のプログラムを変更して、250万円を48ヶ月以内で返済するプランを立てよ。(例:月々X万円 YYヶ月)。プログラムも提出すること。
2. 1と同じプランを、式3.8を用いて漸化式の形で表現せよ。
3. 上のプログラムを変更して、4000万円を30年以内で返済するプランを立てよ。

3.3 付録：関数とプログラムの違い

関数とプログラムの違いについて簡単にまとめておこう。関数とプログラムは、一見似ているが異なるものである。プログラムは実行する計算の「本体」であり、関数はそのための「部品」である。プログラムには似たようなことを繰り返すことがあり、「似たようなこと」を切り出して関数とすると、プログラムの複雑さを減らすことが出来る。また、似たようなことを繰り返す場合でも、部品が既にあるのであれば容易にプログラムを作ることが出来る。人間がおなじことを繰り返さない、というのが計算機を使いこなす鍵である。

Octave の関数において、もっともわかりにくいのが、「戻り値」の定義である。sum2.m をもう一度見てみよう。

```
#sum2.m
function sum=sum2(X)
    [rows, cols]=size(X);
    sum=0;
    for(i=1:rows)
        sum = sum+X(i,1);
    endfor
endfunction
```

冒頭の function の次にくる “sum=” の sum という変数が戻り値である。関数の結果が、この変数に代入されるという意味である (“sum2” は関数の名前であるから混同しないように)。しかし、この “sum” という変数は プログラムからは見えない。

```
Octave:> sum=0;
Octave:> source("sum2.m");
Octave:> sum2(X);
ans = 55
Octave:> sum
sum = 0
```

一見紛らわしいがこれは正しい結果である。1 行目で sum に 0 を代入していることに注意。sum2() は、プログラムの中の “sum” という変数をいっさい操作しない。

プログラムで使うには、

```
S1 = sum2(XX);
```

など、プログラムの中で使用している変数に代入する必要がある。