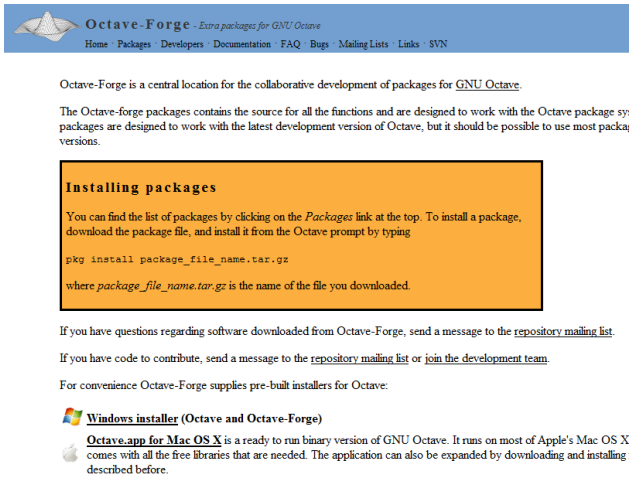
Octave の使い方とプログラムの基礎

1. インストールと環境設定

1.1. インストール

Octave は、市販の MATLAB によく似たフリーの数値計算ソフトウェアで、MATLAB のコマンドや関数が利用でき、3 次元など高度なグラフィック機能も統合されているので、プログラミングの経験がほとんどない人でも、簡単なコマンドを憶えてしまえば、かなり高度な数値計算ができ、簡単にグラフなどに表示できる。http://octave.sourceforge.net/から各 OS 用のインストール用ファイル(Windows Installer や Octave.app)がダウンロードできる。2011 年 10 月現在の最新バージョンは 3.4.0。ただし windows 版は 3.2.4 が最新。



インストールした Octave を起動すると、どの OS でも、

こんな黒い画面が出てくる。最後の行の「>」のことをプロンプトといい、この「>」の後にコマンドを打ち込んでいくと、インターラクティブにコマンドを実行することができる。このようなインターフェースのことをコマンドラインという。

本資料では、見やすさのため以下のような色分けする。

黒に白文字はコマンドラインを表す

```
> a=1+2;
```

グレーに黒文字はプログラムやスクリプトを表す

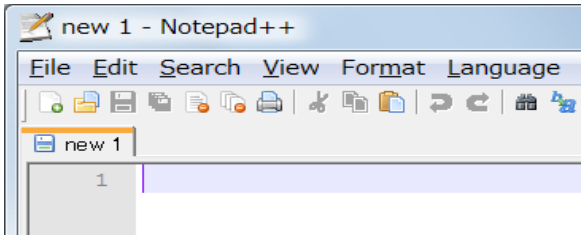
```
x=0:0.1:10*pi;
```

1.2. テキストエディタ

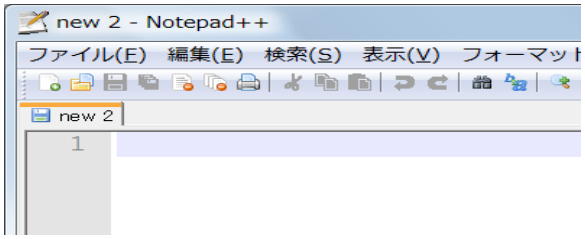
どんなプログラムでもスクリプトでも、通常はテキスト形式で書く。テキストを編集するソフトをテキストエディタと呼ぶ。もし自分の使い慣れたテキストエディタがあれば、それを使うといい。テキストエディタって何?という人は、Octave をインストールするとき、デフォルトでインストールされる Notepad++というテキストエディタをお勧めする。コマンドラインから

```
> edit
```

と打ち込むと Notepad++が起動する。見た目はこんな感じ。



Octave(Matlab)だけでなく、C や Fortran, TeX, Java など多数のプログラム言語に対応していていて便利なので、Octave 以外の言語を使うときもお勧めしたい。ただし、Octave をインストールするときデフォルトでインストールされた Notepad++は、英語版なので、日本語版をインストールしたい人は、「Notepad++ EUC-JP 対応版」で検索してしてほしい。下は日本語版。



1.3. 簡単なカスタマイズ

■ プロンプトに

```
Octave-3.2.4.exe:1>
```

などの文字が毎回出て来てウザイ! という人は、**PS1('>')**と入力すると次から表示されなくなる。

```
Octave-3.2.4.exe:1> PS1('>')
>
```

- フォントが見にくい場合(windows)は、Octave のショートカットのアイコンを右クリックし、プロパティからフォントタブを選択すると、フォントとサイズを変更できる。
- パソコン上級者や Octave に慣れてきた人で、毎回**PS1('>')**を打ち込むのが面倒、という人は、設定ファイルを作っておくと便利。テキストエディタで、

```
PS1('>')
```

と書き、Octave が保存されているフォルダ(デフォルトでは C:¥Octave¥3.2.4_gcc-4.3.0¥bin)に.octaverc という名前で保存する。.octaverc の先頭にピリオドがあることに注意。このファイルの中に書いたコマンドは、Octave を起動するとき、最初に実行されるので、起動するたびに実行したいコマンドなどは、あらかじめこの中に書いておくと便利。

2 基本操作（コマンドライン）

2.1 四則演算

```
> 2+1-5
ans = -2
> 2*3
ans = 6
> 3^3
ans = 27
```

特に説明するまでもない。**ans** は answer の略。

2.2 基本的な代入文

```
> x=2              x という変数に 2 を代入
x = 2
> y=3              y に 3 を代入
y = 3
> z=x/y            z を x/y で定義
z=0.6667
```

変数の中身をメモリーから消したい(クリアしたい)ときは、

```
> clear x    変数 x の中身をクリア
> clear      すべての変数をクリア
```

2.3 基本的な関数

Octave ではさまざま初等関数があらかじめ用意されている。これを**ビルトイン関数(組み込み関数)**という。

```
> sqrt(2)          %ルート
ans = 1.4142
> cos(0)           %三角関数
ans = 1
> exp(1)           %指数関数
ans = 2.7183
```

初等関数の他にも、

gamma(x) ガンマ関数
beta(z,w) ベータ関数
besselj(n,x) 第 1 種ベッセル関数

などの特殊関数も様々用意されている。

2.4 予約語

i, **j**, **e**, **pi** など、あらかじめ数値が定義されている文字例がある。このような文字列のことを**予約語**という。

```
> i                %i は複素数を表す
ans = 0 + 1i
> j                %j も複素数を表す
ans = 0 + 1i
> e                %自然対数の底（ネイピア数）
ans = 2.7183
> pi               %円周率
ans = 3.1416
```

プログラムを書く場合は、混乱を避けるため、これらの文字を変数として使用しない方がよい。例えば、電流を **i** と定義し、**i=0** などと代入してしまうと、複素数として予約されていた **i** が上書きされ、複素数としての **i** が使えなくなってしまう。

2.5 その他

- スクリプトの中で**コメント文**にしたいときは%を使う
- 打ち込んだコマンドは履歴に保存されていて、上下の矢印キー↑↓を押すと、それまで使ったコマンドが次々表示されるので、同じコマンドを入力するときに使うと便利。

■ 答え(ans =)を画面に表示させたくないときは最後にセミコロン「;」をつける。

```
> a= 1+2;          %最後にセミコロンをつける
>
変数 a には 3 が格納されているが、表示はされない
```

■ 15 桁（倍精度)で表示したいとき

```
> format long    %倍精度で表示
> pi
ans  =  3.14159265358979
```

■ 5 桁（単精度)で表示したいとき。

```
> format short   %単精度で表示
> pi
ans  = 3.1416
```

2.6 簡単なベクトル(配列、数列)の作り方

プログラム言語では、ベクトルや数列のことを配列と呼ぶこともある。プログラム上ではベクトル、数列、配列の区別なく、すべて同じように表現できる。下にいくつか例を示す。

```
> a=0:10    %a は 0 から 10 まで 1 ずつ増える行ベクトル
a = 0 1 2 3 4 5 6 7 8 9 10
```

要素を x ずつ増やす場合は、x を:で挟んで間に入れる。

```
> a=1:2:10  %a は 2 ずつ増える
a= 1 3 5 7 9
> a=0:0.3:1  % 0.3 ずつ増える
a=
  0.00000  0.30000  0.6000  0.9000
```

ベクトル(数列)の要素を表示したい時は、

```
> a(3)    %ベクトル a の 3 番目の要素を表示
ans = 0.6
```

(注) C 言語の配列の要素番号は 0 から始まるが、Octave は 1 から始まる。なので、0 番目の要素 a(0)を表示しようとすると、エラーを返す。

```
> a(0)
error: subscript indices must be either
positive integers or logicals.
```

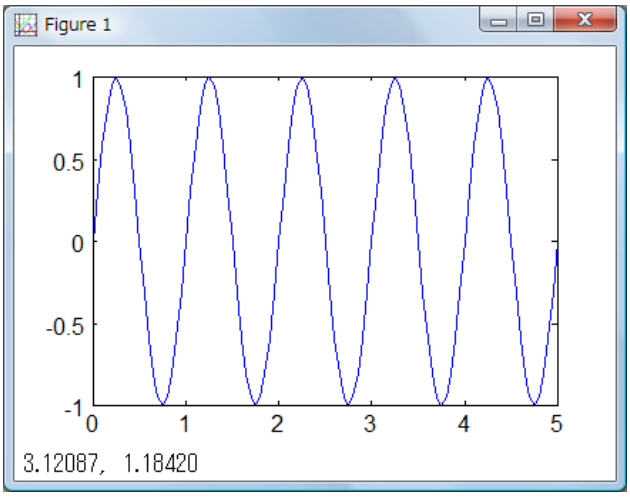
2.7 データの表示

Octave ではデータ表示用の 2D, 3D グラフィック用のコマンド(plot, pcolor, surf など)も多数用意されている。

(例題) $y=\sin 2\pi x$ を $0 \le x \le 5$ で図示せよ。

```
> x=0:0.05:5;
    %定義域をベクトルで作る。刻みは 0.05 にした。
> y=sin(2*pi*x);
    %関数にベクトルをそのまま代入。pi は予約語。
> plot(x,y);
```

のようにコマンドラインで打ち込むと、



こんなグラフが表示されるはず。sin 関数の入力、つまり() 内には、スカラーだけでなく、ベクトルや行列を入れても OK。通常プログラミング言語では、関数の入力(引数という)は、スカラーなのかベクトルなのか、あるいは行列なのか厳密に区別する必要があるが、Octave の場合は特に区別する必要はない。この辺も Octave の便利なところ。

軸にラベルやタイトルをつけたいときは、

```
> xlabel('x axis');      %x 軸ラベル
> ylabel('y axis');      %y 軸ラベル
> title('test');         %タイトル
> legend('sine curve'); %凡例
```

などのコマンドがある。Octave は、gnuplot というグラフソフトと統合されているので、gnuplot のコマンドがだいたいそのまま使える。gnuplot について詳しくはネットか何かで調べてほしい。

2.8 help コマンド

コマンドや関数の使い方が分からなくなったら、**help** コマンドが便利

（例）**plot** の使い方を調べたい。

```
> help plot
'plot' is a function from the file C:...
--Function File: plot (Y)
--Function File: plot (X,Y)
--Function File: plot (X,Y, PROPERTY, VALUE,.
```

つまり、**plot** は、**plot(Y)**，**plot(X,Y)** などの書式で使えるというようなことが英語で表示される。

3. ベクトルと行列

3.1 ベクトルと行列の作り方

■ 行ベクトル（row vector）

カギ括弧 **[]** の中で、スペースで区切って数字を並べれば行ベクトルを作ることができる。

（例題）行ベクトル a=(2 3 4)を定義せよ。

```
> a=[2  3  4]      %スペースで区切る
a =
   2  3  4
```

これは、コロン「：」記号を使って、

```
> a=2:4
a =
   2  3  4
```

としても全く同じ。

■ 列ベクトル(column vector)

列にするときは、セミコロン「；」で区切る。

（例題）

列ベクトル $a=\begin{pmatrix}2\\3\\4\end{pmatrix}$ を定義せよ。

```
> a=[2;3;4]        %セミコロンで区切る
a =
   2
   3
   4
```

セミコロン「；」の前や後にはスペースを入れてもよい。

■ 行列(matrix)

(例題)

3×3 行列 $\mathbf{A}=\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$ を定義せよ。

列ベクトルと行ベクトルを作る方法を組み合わせて、

```
> A=[1 2 3; 4 5 6 ; 7 8 9]
A =
   1  2  3
   4  5  6
   7  8  9
```

（例題）上記の 3×3 行列 A の要素(2,3)を表示せよ。

```
> A(2,3)
ans = 6
```

■ 転置行列

右にアポストロフィー「'」をつける。

（例題）行ベクトル a=(1 2 3 4)を列ベクトルに変換せよ。

```
> a=1:4
a=
   1  2  3  4
> a'
ans =
   1
   2
   3
   4
```

（注）複素数の場合は共役転置行列になる。

3.2. 行列の演算

■ **行列のかけ算**

スカラーのかけ算と同じように、アスタリスク「*」を使う。

（例題）

$\mathbf{M}=\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$ $\mathbf{A}=\begin{pmatrix}1\\2\\3\end{pmatrix}$のとき、**B=M×A** を求めよ。

まずは、**M** と **A** を定義する。

```
>  M=[1 2 3; 4 5 6 ; 7 8 9]
M=
   1  2  3
   4  5  6
   7  8  9
> A = [1;2;3]
A =
   1
   2
   3
```

```
> B=M*A   アスタリスク(*)を使う
B =
   14
   32
   50      (答)
```

(例題) $\mathbf{M}=\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$ $\mathbf{N}=\begin{pmatrix}-1&2&3\\0&5&0\\1&0&-2\end{pmatrix}$ のとき、**B=M×N** を求めよ。

```
> N=[-1 2 3; 0 5 0 ; 1 0 -2]
N =
   -1   2   3
    0   5   0
    1   0  -2
> B=M*N
B =
    2   12  -3
    2   33   0
    2   54   3
```

行列の足し算、引き算、べき乗は、それぞれ*****、**-**、**^**、などが使える。当然ながら、行列の次元が適切でないとエラーになる。

■ **逆行列**

行列の右側に**^-1** をつけると逆行列が求まる。

(例) **N** の逆行列 $\mathbf{N}^{-1}$ を求める(**N** は上で使った 3×3 行列)

```
> N^-1
ans =
   2.00000  -0.80000  3.00000
   0.00000   0.20000  0.00000
   1.00000  -0.40000  1.00000
```

(例)上で使った **B** は特異行列(singular)なので、**B^-1** は要素が無限大(**Inf**)になってしまう。

```
> B^-1
warning: inverse: matrix singular to
machine precision, rcond =0
ans =
   Inf  Inf  Inf
   Inf  Inf  Inf
   Inf  Inf  Inf
```

■ **行列式(determinant)**

関数 **det** が用意されている。

```
> det(N)
ans = -5
> det(B)
ans = 0     %B は特異行列なので行列式は 0
```

■ **行列の要素同士の計算(ドット付きの計算)**

行列計算ではなく、**要素**のかけ算、割り算、べき乗などをする場合は、それぞれの前にピリオド「.」をつける(「**.***」、「**./**」、「**.^**」など)。何を言ってるのか意味が分からんという人のために例で示そう。

(例題) $y=x^2$ を $0\le x\le 1$ の範囲で図示せよ。

まず、行列(ベクトル)**x** を定義しよう。

```
> x=0:0.1:1;   %x を定義
```

これで、**y=x^2** と打ち込めば、**y** が定義できるのではないかと思ってしまうが、実際やってみると、

```
> y=x^2;
error: for A^b, A must be square
```

と、エラーになる。「**x^2**」は、行列 **x** の要素の 2 乗ではなく、今の場合はサイズが 1×11 の行列 **x** の 2 乗(**x×x**)を表すことになる。したがって、上のエラーは、**x** が正方行列(squre)でなければダメだ、といっているわけだ。要素の 2 乗をとるためには、ドット付きの演算にしないといけない。

```
> y=x.^2;        %^の前にピリオドをつける
```

これで、それぞれの要素の 2 乗が得られる。図示は **plot(x,y)** とすれば良い。

このドット付き(要素同士の計算)と、ドットなし(行列計算)では意味が全く違ってくるので、注意が必要。この違いは見た目にも非常に紛らわしく、プログラミングするときも、見落としがち。もう一つの例として、3×3 行列の場合を下に示す。

```
> A=[1 2 3; 4 5 6; 7 8 9];
> A.*A
ans =
   1    4    9    %それぞれの要素が 2 乗に
  16   25   36    %なっている。
  49   64   81
> A*A
ans =
  30   36   42
  66   81   96
 102  126  150
```

A の要素の 2 乗である **A.*A** と、行列 **A** の 2 乗である **A*A** は答えが違う。上の例では正方行列になっているので、特にエラーは出ないから、気づきにくく、そのまま放っておくと、致命的な計算ミスに至る。

■ **行列の指数関数**

たとえば **X** が $m \times n$ 行列で、要素が X_{11}、X_{12}....X_{mn} のような場合、**sin(X)** は、**sin(X11) sin(X12)** のような値を要素にもつ $m \times n$ 行列を出力する。**exp(X)** なら、行列の要素の **exp** を出力するので、量子力学でよくでてくる行列の指数関数($e^{X} = \sum_{n=0}^{\infty} \frac{1}{n!} X^{n}$)を表すわけではない。行列の指数関数を出力したい場合は、**expm(X)** という別の関数が用意されている。

```
> X=[0 2; 0 0]
X =
  0  2
  0  0
> exp(X)
ans =
  1.0000  7.3891
  1.0000  1.0000
> expm(X)
ans =
  1  2
  0  1
```

というように、**exp(X)** と、**expm(X)** は答えが違う。

4. プログラミングの基礎(の基礎)

4.1 データの入出力

プログラムを実行した結果を保存してするには、**save**、保存したデータを読み出すには、**load** を使う。

(例) 行列 **M** とベクトル **a** を **test.mat** というファイル名のバイナリ形式で保存する場合。

```
> save test.mat M a
```

Octave のバイナリ形式は、MATLAB では直接、読み出すことが出来ない。市販の MATLAB との互換性を重視したいのなら、上の **save** の文の最後にオプションの「**-V6**」をつけて、

```
> save test.mat M a -V6
```

とすると、ver.7 以降でも読み出せる形式で保存できる。

(例) **test.mat** を読み出す

```
> load test.mat
```

上で保存した **test.mat** なら、行列 **M** とベクトル **a** が同時にロードされる。

(例) テキスト形式で行列 **M** を **M.txt** に保存したい場合はオプション(**-ascii**)をつける。

```
> save -ascii M.txt M
```

この場合は、一つのファイルに一つの行列しか保存できない。

4.2. for ループ

繰り返し似たような作業をするときには、for ループをよく使う。書式は微妙に違うが C、FORTRAN、その他の言語でも頻繁に使う命令だ。（FORTRAN の場合は do ループという）

（例題）要素が 0、0.1、0.2、.... 10 というように、0.1 ずつ増加するベクトル *a* を作れ。

```
> a=0:0.1:10
```

とすることも出来るが、for ループを使うなら、次のようになる。

```
for k=1:101
  a(k) = 0.1 * (k-1) ;
end
```

（例題）*n*=10 のとき、*n*!を求めるよ。

```
> n=10;          %n を定義
> a(1)=1;        %a の 1 番目の要素を 1 に
> for k=2:n      %2 から n まで(n-1)回繰り返す
>  a(k)=k*a(k-1);
        %k 番目の要素は、k-1 番目の要素と k の積。
> end            %ここまで繰り返す
> a(n)           % a の n 番目の要素が答え
ans = 3628800   （答え）
```

ただし、10!を求めるだけなら、

```
> prod(1:10)     %prod は、総積を求める関数
ans = 3628800   （答え）
```

の方がずっと簡単。

4.3. if 文

（例）a が負の数であれば、b を 0 に、そうでなければ 1 にする。

```
if a < 0
     b=0
else
     b=1
end
```

条件文の指定は、普通の数学記号と少し異なるので注意。

条件文として *a*=0 を表現する場合、イコール記号「=」を二つ並べる。

```
a == 0
```

大なりイコール(≧)などは、以下

```
a >= 0
```

二つの条件を OR で繋ぎたい場合は、縦棒「|」で繋ぐ。例えば *a*<0 または *a*>5 を表現する場合

```
a<0 | a>0
```

二つの条件を AND で繋ぎたい場合は、「&」を使う。0<*a*<5 を表現する場合は、

```
a>0 & a<5
```

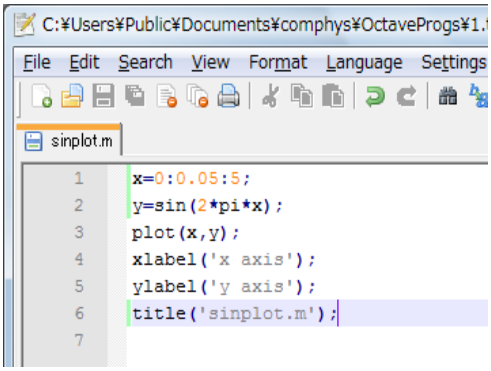
4.4. スクリプト

コマンドラインは便利だが長いプログラムを毎回いちいち打ち込むのは不便。そこでコマンドを順番に書き、拡張子.m のテキスト形式で保存すると、ファイル名をコマンド名としたプログラムとして実行出来るようになる。このような簡易プログラムのことをスクリプトという。

（例題）2.7 で取り上げた sin 波を表示するプログラム(スクリプト)を書け。

```
x=0:0.05:5;
y=sin(2*pi*x);
plot(x,y);
xlabel('x axis');
ylabel('y axis');
title('sinplot.m');
```

これを sinplot.m というファイル名で保存する。実際には



こんな感じ。保存した場所が、**C:¥Users¥ABC¥octave** の場合、そのフォルダにコマンド **cd** で移動してから実行する。

```
> cd C:¥Users¥ABC¥octave
                %保存したフォルダに cd で移動する
> sinplot       % 実行コマンド
```

あるいは、二つのコマンドを合わせて、

```
> C:¥Users¥ABC¥octave¥sinplot
```

とすると、

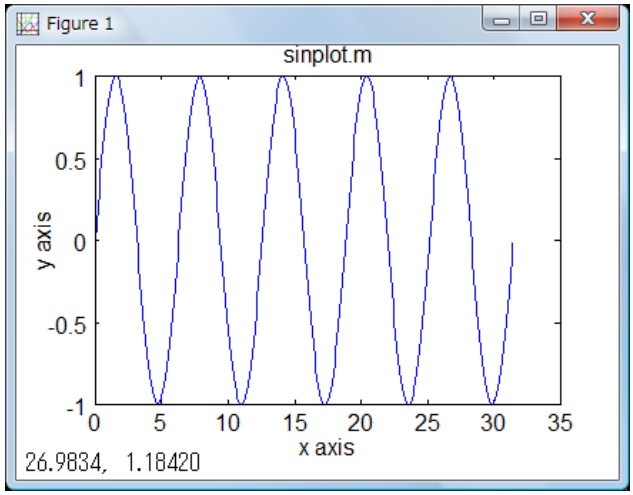


図 4.4.1

というような図が表示される。

プロンプトで使える便利なコマンド
UNIX などと同様なものが使える。
cd　ディレクトリの移動（change directry の略）
ls　ディレクトリ内のファイルのリスト
pwd　現在のディレクトリを表示

（例題）検出器で宇宙線(高エネルギーの粒子)を 1 秒ごとに測定したら図 4.4.2 のようなデータが得られた。ここで検出器を流れた電流 (Current from detector (nA))が 10nA を超えたときだけ、宇宙線が検出器に到達したと判断し、それ以外の場合はノイズだと判断しよう。測定中に何個の宇宙線が検出されたか?

図 4.2.2 を見れば、答えはたぶん 2 だと分かると思うが、データがもっと膨大になったら、いちいち人間が数えていられない。そこで宇宙線の個数を数えるプログラムを for ループと if 文を使って作ってみよう。

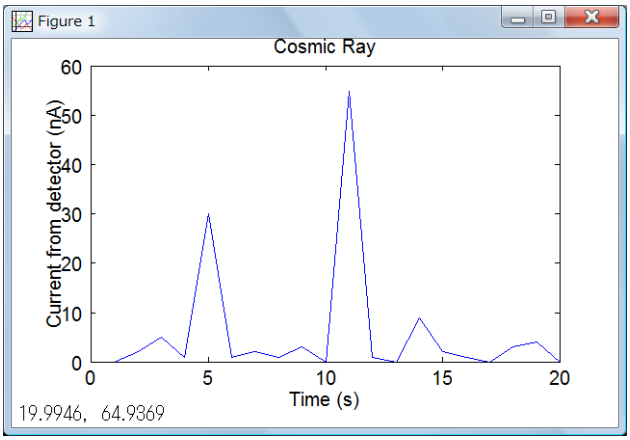


図 4.4.2

基本的には、データを読んで、もし 10 nA より上だったら、1 個とカウントし、そうでなければカウントしないという作業を、データの長さ分だけ繰り返せばいい。大体の流れを言葉で書くと、

0. メモリーに残っているデータをクリアする。
1. 測定結果のデータを入力
2. 判定基準 threshold を決める。
3. for ループで、測定点の回数だけ回す
もし、k 番目のデータ値が threshold より大きければ、
R の要素を 1 にする。
そうでなければ、
R の要素を 0 にする。
ここまでを繰り返す。
4. 出来上がったベクトル R の要素の合計を求める(これが答え)

これを実際にプログラムに直すと

```
% cosmic.m
clear          %メモリーに残っているデータをクリアする。
data=[0 2 5 1 30 1 2 1 3 0 55 1 0 9 2 1 0 3 4 0];
                %観測結果のデータ
threshold = 10;   %判定基準を 10nA にする
for k=1:20        %測定点は 20 個
  if data(k) > threshold
                %判定基準より大なら
    R(k) = 1;     %R の要素を 1 に
  else
    R(k) = 0;     %そうでなければ 0 に
  end
end
num=sum(R)        %R の合計を num とする。
```

最後の行で使っている **sum(R)** は、ベクトル **R** の要素の合計を求めるビルトイン関数。このプログラムを **cosmic.m** というファイルで保存し、実行すると、

```
> cosmic
num = 2          （答）2 個粒子を観測した。
```

(Octave、Matlab 独特の行列表現)

Octave では、**for** と **if** を使わずに、以下のように行列のまま扱うこともできる。

```
%cosmicMatrix.m
clear
data=[0 2 5 1 30 1 2 1 3 0 55 1 0 9 2 1 0 3 4 0];
threshold = 10;
R=(data > threshold);    %この文がみそ
num=sum(R)
```

「**R=(data > threshold)**」という文は、「ベクトル **data** の要素の中で、スカラー**threshold** より大きい場合は、同じ要素番号の要素を **1**、そうでない場合は **0** を要素にもつようなベクトルを **R** と定義する」というややこしいコマンドだ。例えば、**data=[0 2 0.9]**なら、**R=[0 1 0]**となるし、**data=[-1 12 1.1 2]**なら、**R=[0 1 1 1]**となり、**R** の要素は必ず **0** か **1** になる。

この表現は慣れないと取っつきにくいが、こちらの方が、プログラムの見た目が簡潔になるだけでなく、for ループに比べて計算スピードが極めて高速になるので、慣れてきたら、なるべく行列を行列のまま扱うような、「for ループを使わない表現」を使うことをお勧めしたい。

(例)

日本地図の画像データを使って、陸地は 1、海は 0 という 2 値のデータに加工したい。どうしたらいいか? (この 2 値データは 2 次元の偏微分方程式の数値計算の例として、日本列島近海での津波の伝搬を計算するときの境界条件として使う予定。)

画像データとしては、日本地図.png というファイルを wiki からダウンロードし、ファイル名を map.png に変更しておく。(日本地図.png で検索すると出てくる)

図 4.4.2

まず画像データを行列データとして読み込む。画像はデータが二次元なので For ループを x 軸と y 軸で 2 回用い、各ピクセルごとで、if 文を使って陸地か海かの判定を行う。ここでは各ピクセルの数値が 200 より小さかったら陸地、そうでなければ海として判定することにした。画像が少し大きいので、実際のプログラム(**map2bit.m**)では、画像のサイズを 1/3 にするために、 **for k=1:3:ymax** のようにしている。

```
%map2bit.m
clear
A=imread('map.PNG');   %画像データを読み込む
M=double(A(:,:,3));    %数値データに変換
[xmax ymax]=size(M);    %画像のサイズを求める
y=0;
for k=1:3:ymax     %y 軸に関する for ループ
    x=0;  y=y+1;
    for h=1:3:xmax  %x 軸に関する for ループ
        x=x+1;
        if M(h,k) < 200   %陸地と海の境界を 200 にする
            map(x,y)=1;   %陸地なら 1 に
        else
            map(x,y)=0;   %海なら 0 に
        end
    end
end
figure; %ここからは作図
pcolor(flipud(map));  %カラープロットで図示
colormap(gray); %カラーマップをグレースケールにする
shading flat    %ピクセルの境界線を非表示にする
```

For ループと if 文を使わない方法としては、

```
%map2bitM.m
clear
A=imread('map.PNG');
M=double(A(:,:,3)<200);   % この1文がみそ
[xmax ymax]=size(M);
map=M(1:3:xmax,1:3:ymax);
figure; %これ以降は map2bit.m と同じなので省略
```

となる。

5. 関数(function)

5.1. function を作る

関数といえば、sin とか exp のような初等関数をイメージすると思うが、情報処理の分野では、「ある入力に対して、何か出力してくれる手続きのまとまり」という意味でも使われている。この”関数”という概念は、どの言語でも極めて重要なので、しっかり理解して使いこなしてもらいたいものだ。言語そのものや、追加パッケージなどであらかじめ関数は用意されているものだが、どの言語でも、自分で新たな関数を定義することができる。これをユーザー関数と呼ぶ。

4.4 の宇宙線の例題を、関数(function)を使ったプログラムで書いてみよう。Octave の場合、関数を定義する場合は、スクリプトの一番上に、「このスクリプトは関数である」ことを示すために、**function** で始まる宣言文を書く必要がある。ただし、次の例のように宣言文の前にコメント文があっても問題はない*。

```
% function num=cosmicFunc(data,threshold)
function num=cosmicFunc(data,threshold)
                    %関数 cosfunc の定義
[m n]=size(data); %データのサイズ n を抽出する
for k=1:n
  if data(k) > threshold
    R(k) = 1;
  else
    R(k) = 0;
  end
end
num=sum(R)
```

関数 **size** は、行列のサイズを出力するビルトイン関数。例えば、**data** が 3×4 行列なら、**[m n]=size(data)** によって、**m=3**、**n=4** が代入される。このプログラムを cosmicFunc.m という名前で保存する。これで、**data** と **threshold** という二つの変数を入力すると、**num** という答えを出す **cosmicFunc** という関数ができ上がった。

5.2. プロンプトでの function の使い方

プロンプトでの使い方の例として、ここではあらかじめ、data_large.mat というファイルに、4.4 の例よりも、データ量の大きい **data_large** というデータがすでに保存されてあるとして、話しを進める。**data_large** は図 5.2 のようなもの。 なお、このデータは、

```
> data_large=sprand(1,1000,0.06)+
  rand(1,1000) * 0.4 + rand(1,1000)*0.1
> save  data_large.mat  data_large
```

というコマンドを使って作成した。**sprand, rand** は乱数を発生させる関数。関数の詳細は **help** 参照。

* **function** 宣言文の前にコメント文で説明を書いておくと、プロンプトから、**help** ”**関数名**”で、コメント文が表示されるので、関数の書式や使い方を書いておくと便利。

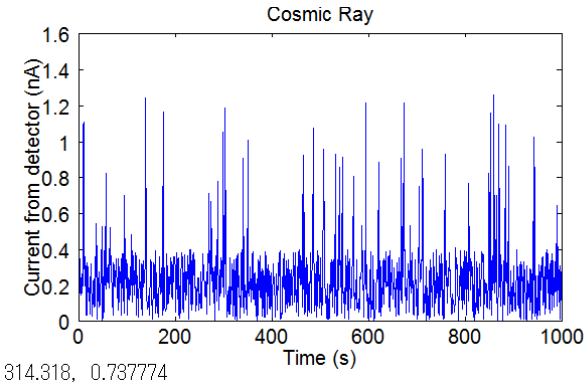


図 5.2

測定器に 0.5 nA 以上の電流が流れたとき、宇宙線が観測されたと判定し(しきい値 **threshold=0.5** に設定)、それ以下の電流が流れた場合は、ノイズだと判定した場合、

```
> load data_large.mat  %データのロード
> threshold=0.5;       %判定基準の設定
> cosmicFunc(data_large,threshold)
ans = 48        (答) 48 の粒子を観測した。
```

となる。判定基準を 0.6 nA としたときは、

```
> threshold=0.6;
> cosmicFunc(data_large,threshold)
ans = 41        (答) 41 個の粒子を観測した。
```

となる。判定基準が違うので、結果は当然異なる。

5.3. プログラム内での function の使い方

今度はプログラム内で function を使ってみよう。

(例題) 上の例の **cosmicFunc** を使い、判定基準(しきい値 **threshold**)の設定値を変えたとき、カウントされる粒子数がどのように変化するかを解析するプログラムを書け。

```
% cosmicJudge.m
clear
load data_large.mat
thresh=0:0.01:1;
[m n]=size(thresh);
for k=1:n
  num(k)=cosmicFunc(data_large,thresh(k));
        %ここで関数 cosmicFunc を使っている
end
plot(thresh,num)
xlabel('Threshold (nA)');   %図のラベルなど
ylabel('Counts');
title('Threshold dependence');
```

これを cosmicJudge.m という名前で保存し、拡張子なしで **cosmicJudge** と実行すると、図 5.3 のような図が表示される。

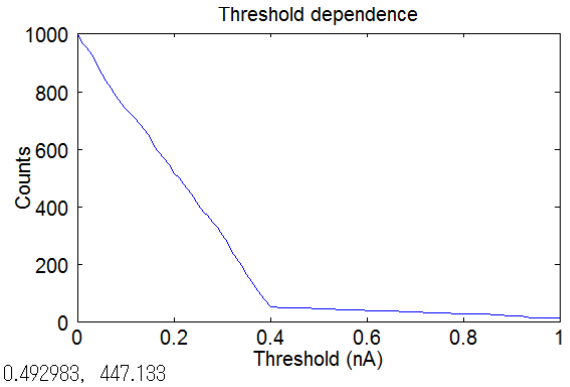


図 5.3

ちなみに上の例の場合、Threshold が 0.4 nA 付近を下回ると、カウント数が急激に増えているので、0.4 nA 以下の信号はノイズではないかと推測される、などと解析したりする。

5.4. 変数の有効範囲

関数を使うときは、変数の有効範囲を意識する必要がある。先の例でいうと、関数 **cosmicFunc** 内に、**[m n]=size(data)** というコマンドがあるが、ここで出て来ている **n** は、関数 **cosmicFunc** 内でしか有効ではない。つまり関数の入力である **data_large** と **threshold** 以外の変数は、メインプログラムから関数へは受け渡されず、逆に、関数の出力である **num** 以外の変数はメインプログラムに受け渡されない。このような受け渡されない変数(たとえば **n**)を**ローカル変数**という。

一方、メインプログラムと関数で同じ変数を同じ名前の文字列で使いたいときもある。その場合は、変数をプログラム全体で使用できる**グローバル変数**として宣言する必要がある。

たとえば、関数 **cosmicFunc** 内で **data** のサイズを、**[m n]=size(data)** で求めているが、これをメインプログラム(cosmicJudge2.m)に移動し、グローバル変数する場合、メインプロクラムは、

```
% cosmisJudge2.m
Clear
global n2    %n2 をグローバル変数として宣言する。
load data_large.mat
thresh=0:0.01:1;
[m n]=size(thresh);
[m2 n2]=size(data_large);
             %data_large のサイズを n2 として定義
for k=1:n
  R(k)=cosmicFunc2(data_large,thresh(k));
end
plot(thresh,R) %ラベルなどはこのあと適当に書く
```

となる。これを cosmicJudge2.m として保存。関数の方は、

```
% cosmicFunc2.m
function num=cosmicFunc2(data,threshold)
global n2    %n2 をグローバル変数として宣言する。
for k=1:n2        %n2 回繰り返す。
    if data(k) > threshold
       R(k) = 1;
    else
       R(k) = 0;
    end
end
num=sum(R);
```

とし、これを cosmicFunc2.m として保存。これで、**n2** はグローバル変数として定義され、プログラムのどこででも使えるようになった[†]。

5.5. 関数ファイルを置くフォルダについての注意

通常ユーザーが定義した関数とメインプログラムのファイルは、同じフォルダ内に置かなければならない。同じフォルダにない場合は「あらかじめ path を通さないといけない」のだが、説明は省略。とにかく慣れないうちは、メインプログラムと関数を同じフォルダにおいて置くようにすればよいだろう。

> 関数とサブルーチン
> プログラムの本文、つまりメインの部分のことをメインプログラムといい、手続きのひとまとまりをサブプログラム(サブルーチン)という。関数は、サブプログラムのうちでも、入力と出力以外をブラックボックスとして扱うものだが、C 言語のように、サブプログラムも関数(function)と呼ぶ言語もあってややこしい。用語を憶えるのは二の次として、とにかく関数を自由に使いこなせるようにしてほしい。

[†] このようにすると、for ループ内で毎回処理されていた **[m,n]=size(thresh)** がなくなるため、その分、計算時間が少し速くなる。一般に for ループ内には、不必要なコマンドはなるべく入れないようにすると、プログラムの高速化が図れる。