Short introduction to Octave

Lecture Notes

by Piotr Kowalczyk

Features

- high level programming language and interactive environment
- intended especially for numerical calculations
- natively supports many mathematical concepts
- many function libraries available

- *free* and compatible with MATLAB

`www.octave.org`

Basic use

- simple calculator:

```
>> 2+2
ans = 4

>> 3^(2+1)
ans = 27

>> 0^0
ans = 1

>> (-4)^(.5)
ans = 1.2246e-016 + 2.0000e+000i
```

- built-in support for complex arithmetic
- special kinds of “numbers”: **`Inf, NaN`**

Basic use

- built-in functions — all of the usual mathematical functions:

```
>> exp(1)-e
ans = 0

>> sqrt(-4)
ans = 0 + 2i

>> 1.5*log(2+sin(3+pi))
ans = 0.92996
```

- named variables:

```
>> deg = pi/180
deg = 0.017453
>> h = cos(60*deg);
>> h
h = 0.50000
```

Vectors

- row vector: `>> v=[3 5 1 -2]; #or v=[3, 5, 1, -2];`
- column vector: `>> v=[-1; 1.5; 3; 0];`
- colon notation: start[:increment]:end (*range*)

```
>> v=3:6
v =
    3   4   5   6
>> x=0:.2:1
x =
  0.00000  0.20000  0.40000  0.60000  0.80000  1.00000
```

- evenly spaced vector:

  linspace(x1,x2,N); *#N elements between x1 and x2*

- logarithmically spaced vector:

  logspace(x1,x2,N); *#N elements between 10^x1 and 10^x2*

Matrices

Generating:

```
>> A = [1 2 3; 4 5 6]
A =
     1  2  3
     4  5  6
>> B = [A; 3:-1:1]
B =
     1  2  3
     4  5  6
     3  2  1
>> [A A]
ans =
     1  2  3  1  2  3
     4  5  6  4  5  6
```

Matrices

Special matrices:

```
eye()   # create matrix with ones on the main diagonal
ones()  # create matrix of ones
zeros() # create matrix of zeros
rand()  # create random matrix
        # with entries uniformly distributed in (0,1)
diag()  # create diagonal matrix from the given vector
        # or extract diagonal of the given matrix
```

‘Equation solving’ operators:

```
>> X = A \ B;   # X is solution of AX=B
>> X = B / A;   # X is solution of XA=B
```

Vectors and matrices

Indexing (*indices of vectors and matrices start at one!*):

```
>> v = [-1 3:2:8 0];
>> v(4)
ans =  7
>> A = [1 2 3; 4 5 6];
>> A(2,2)
ans =  5
>> A(1,:)
ans =
      1  2  3
```

Assigning values:

```
>> A(1,:)=v(1:2:5)
A =
   -1  5  0
    4  5  6
```

Vectors and matrices

Dimensions:

```
>> size(v)
ans =
   1  5
>> size(A)
ans =
   2  3
```

Empty matrix:

```
>> P = [];
>> P = [P; [1 2]]
P =
   1  2
```

Built-in functions can take vector and matrix arguments:

```
>> t = 0:.5:2;
>> sin(t)
ans =
   0.00000  0.47943  0.84147  0.99749  0.90930
```

Operators

- arithmetic: `+ - * / ^ ++ --`
- matrix: `+ - * ^ ’ .’`
- element-wise: `.+ .- .* ./ .\ .^`
- logical: `< <= > >= == != ! & | && ||`

```
>> v = [5 8 1 -3 0 4 -8];
>> v < 2
ans =
     0   0   1   1   1   0   1
>> v(v<2)
ans =
     1  -3   0  -8
```

User-defined functions and script files

Defining a function:

```
function [out1, out2, ...] = name (input1, input2, ...)
  sequence of commands
endfunction
```

Calling the function:

```
[outvar1, outvar2, ...] = name(invar1, invar2, ...);
```

Each function is stored in a different file, which *must have the same name as the function.* Alternatively, several functions can be defined in a *script file* with the rest of the program.

Script file is a normal text file (it can not begin with the command `function`). This is the basic form of Octave program. Scripts are run by typing the name of the script (without the extension) in the *Octave* command window. Both function and script files must have an extension `.m`

Control statements

- `if` selection

```
if (condition)        # brackets are not necessary
  commands
elseif (condition)
  commands
else
  commands
endif
```

- `switch` selection

```
switch expression   # different than in C
  case label
    commands
  case label
    commands
  ...
  otherwise
    commands
end
```

Control statements

- `for` loop

```
for variable = expression  # expr.: vector or matrix
  commands
endfor
```

- `while` loop

```
while (condition)
  commands
endwhile
```

- `do-until` loop

```
do
  commands
until (condition)
```

Input and output

- **`save`** `data var1 [var2 ...]`
  saves the variables `var1` etc. into the file `data`
- **`load`** `data`
  restores the variables from the file `data`
- **`fprintf`**, **`printf`**
  resembles C syntax for formatted output
- `var = ` **`input`**`("Text");`
  prints the text `text` and waits for the user to enter a value
- **`format`** `long; ` **`format`** `short;`
  display 5 or 15 significant digits

Graphics

```
x = linspace(0,pi,200);
y = cos(x);
plot(x,y);       # plot a line through the points (x,y)
plot(x,y,"go"); # use green circles instead
hold on;
plot(x,sin(x),"r");  # add red sinusoid
hold off;
title("Sample plot");
legend("cosine","sine")
print("sample.eps","-deps")
figure                  # create new window
```

Other useful functions

- timing the execution of the statements:

```
tic;
# some calculations later ...
toc;   # prints the number of seconds since tic
```

  `cputime` can be used also.

- getting help:

```
>> help
>> help log
```

Sparse matrices

Sparse matrix — matrix with many zeros, only non-zero elements are stored in memory.

To convert a full matrix to a sparse one use:

```
B=sparse(A);
```

Use sparse matrices whenever the size is large!

Building band matrix (sparse diagonal matrix):

```
B=spdiags(V,C,m,n);
```

where column of V are diagonals of B represented by C (negative values — diagonals below the main diagonal, positive values — above the main diagonal) and m, n are dimensions of matrix.

Efficiency

Octave is designed to process matrices and vectors and this is its most powerful feature.

Vectorization is an essential programming skill in Octave.

A few tips on efficient programming in *Octave*:

- avoid using loops (especially `while` and `do-until`)
- if possible, vectorize all operations
- if necessary, rewrite the problem to use matrices and vectors
- use ranges (colon notation) for vectors, whenever possible

Efficiency example

Problem: calculate the sum of squares of numbers $1, 2, ..., n$

Solution 1:

```
n = input ("Type a value for n: ");
tic;
nn = 1;
s = 0;
while (nn <= n)
  s += nn^2;
  nn++;
endwhile
toc;
disp ("Result:"), disp(s);
```

Efficiency example

Problem: calculate the sum of squares of numbers $1, 2, ..., n$

Solution 2:

```
n = input ("Type a value for n: ");
tic;
s = 0;
for nn = 1:n
  s += nn^2;
endfor
toc;
disp ("Result:"), disp(s);
```

Efficiency example

Problem: calculate the sum of squares of numbers $1, 2, ..., n$

Solution 3:

```
n = input ("Type a value for n: ");
tic;
s = sum((1:n).^2);
toc;
printf("Result: %f\n",s);
```