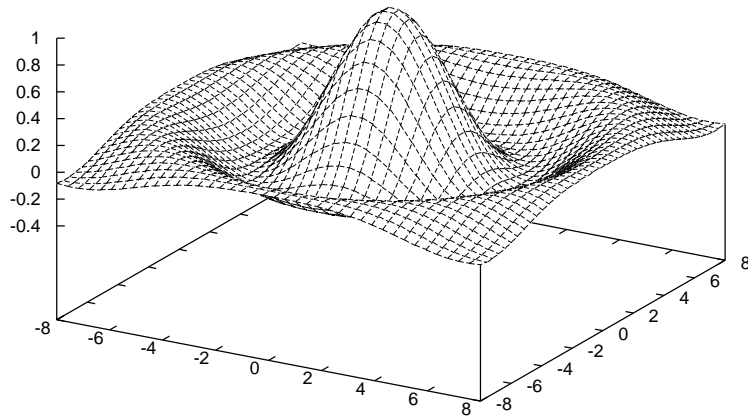


Introduction into GNU Octave

Hubert Selhofer, with revision from Marcel Oliver

2003/04/08

line 1 —



Contents

1	Basics	2
1.1	What is Octave?	2
1.2	Help!	3
1.3	Input Conventions	3
1.4	Variables and Standard Operations	3
2	Vector and Matrix Operations	4
2.1	Vectors	4
2.2	Matrices	5
2.3	Basic Operations	5

2.4	Element wise operations	6
2.5	Indexing and Slicing	6
2.6	Solving linear systems of equations	7
2.7	Inverses, Decompositions, Eigenvalues, etc.	7
2.8	Test of zero entries	8
3	Control structures	8
3.1	Functions	8
3.2	Global Variables	9
3.3	Loops	9
3.4	Branching:	9
3.5	Functions of functions	10
3.6	Efficiency considerations	10
3.7	Input and Output	11
4	Graphics	11
4.1	2D-Graphics	11
4.2	3D-Graphics:	12
4.3	Commands for 2D- and 3D-Graphics:	13
5	Exercises	14
5.1	Linear Algebra	14
5.2	Timing	14
5.3	Stability functions of BDF-Integrators	14
5.4	3D-Plot	15
5.5	Hilbert matrix	16
5.6	Best fit straight line	16
5.7	Trapezoidal rule	16

1 Basics

1.1 What is Octave?

Octave is an interactive programming language, specially optimized for vectorizable calculations and therefore makes standard routines of numerical mathematics (z.B. EISPACK or LAPACK) easily accessible.

The syntax of Octave is very similar to the one used in Matlab, i.e. an Octave program can most of the time also be ran in Matlab. The reverse does not always work, since Matlab, especially in terms of graphics, possesses a considerably larger variety of functions.

1.2 Help!

- `help` lists all the commands and internal variables.
- `help name` gives help topics on the variable or function "name".

Example:

```
octave:1> help eig
```

1.3 Input Conventions

- All commands can be interactive as well as imputed on script data.
- Scripts are textdata with the suffix `.m`. They will be imported through the calling of the data name without suffix, and will behave as if their content is imputed line by line.
- `;` separates more commands in rows, suspending the evaluation of the total.
- `,` separates the commands but does not suspend the evaluation.
- `...` at the end of the line means that the routine is continued in the next line.
- Commentaries are preceded by `%`.
- Octave is case sensitive and differentiates between small caps and big caps.

1.4 Variables and Standard Operations

- `varname = expression` allocates the evaluation of `varname` to `expression`.
- You then have the usual functions `+`, `-`, `*`, `/`, `^`, `sin`, `cos`, `exp`, `acos`, `abs`, etc.
- The elementary logic operators are:

<code><</code>	smaller	<code><=</code>	smaller or equal	<code>&</code>	and
<code>></code>	greater	<code>>=</code>	greater or equal	<code> </code>	or
<code>==</code>	equal	<code><></code>	nonequal	<code>~</code>	not

For the diagnosis of the user defined objects the following commands are used:

- `whos` shows all the defined variables and functions (see sections 3.1).
- `clear name` clears `name` from the memory; if `name` is not defined, *all* the variables and function will be cleared.
- `type name` gives out the variable or function `name` on the screen.

Examples:

```
octave:1> x12 = 1/8, long_name = 'A String'
x12 = 0.12500
long_name = A String
octave:2> sqrt(-1)-i
ans = 0
octave:3> x = sqrt(2); sin(x)/x
ans = 0.69846
```

And here is a script `doless`, that saved in `doless.m`:

```
one = 1;
two = 2;
three = one + two;
```

Calling the script:

```
octave:1> doless
octave:2> whos
*** local user variables:
prot  type                rows  cols  name
====  =====
wd    real scalar           1     1    three
wd    real scalar           1     1    one
wd    real scalar           1     1    two
```

2 Vector and Matrix Operations

Matrices and vectors are the most important building blocks for programming in Octave.

2.1 Vectors

- row vector: $v = (1, 2, 3)$:
 $v = [1 2 3]$
- column vector: $v = (1, 2, 3)^T$:
 $v = [1; 2; 3]$
- Automatic generation of vectors with constant increment:
`Start[:Increment]:End`

Examples:

```
octave:1> x = 3:6
x =
  3  4  5  6
octave:2> y = 0:.15:.7
y =
0.00000  0.15000  0.30000  0.45000  0.60000
octave:3> z = pi:-pi/4:0
z =
3.14159  2.35619  1.57080  0.78540  0.00000
```

2.2 Matrices

A Matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ is generated by:

```
octave:1> A = [ 1 2; 3 4]
A =
  1  2
  3  4
```

Matrices can also be put together out of other matrix components:

```
octave:2> b = [5; 6];
octave:3> M = [A b]
M =
```

```
 1  2  5
 3  4  6
```

For some $(m \times n)$ -matrices there are specific commands. If m equals n , only one argument should be supplied.

- `eye(m,n)` produces a matrix with 1-s on the main diagonal. In the specific case $m = n$ the identity matrix is generated.
- `zeros(m,n)` generates the zero matrix of dimension $(m \times n)$.
- `ones(m,n)` generates a matrix of dimension $(m \times n)$ with $a_{ij} = 1 \forall i, j$.
- `rand(m,n)` generates an arbitrary matrix of dimension $(m \times n)$ with uniformly distributed entries. For other distributions see `help rand`.

2.3 Basic Operations

- `+`, `-` and `*` call matrix addition, subtraction and multiplication.
- `A'` transposes and conjugates `A`.
- `A.'` transposes `A`.

Examples:

```
octave:1> A = [1 2; 3 4]; B = 2*ones(2,2);
octave:2> A+B, A-B, A*B
ans =
  3  4
  5  6
ans =
 -1  0
  1  2
ans =
  6  6
 14 14
```

2.4 Element wise operations

While for example `*` stands for the standard matrix multiplication, a period added in front of all operators stands for element wise assignment of operators: `.*`, `./`, `.^`, etc.

Examples:

```
octave:1> A = [1 2; 3 4]; A.^2 % element wise
```

```
ans =
```

```
    1    4  
    9   16
```

```
octave:2> A^2 % in comparison with matrix wise
```

```
ans =
```

```
    7   10  
   15   22
```

2.5 Indexing and Slicing

- $v(k)$ is the k -th element in the row or column vector v .
- $A(k,l)$ is the matrix element a_{kl} .
- $v(m:n)$ is the “Slice” of the vector v from the m -th to the n -th entry.
- $A(k,:)$ is the k -th row of matrix A .
- $A(:,l)$ is the l -st column of matrix A .

Dimensioning:

- `length(v)` gives back the length of the vector v .
- `[Rows,Columns] = size(A)` gives back the number of rows and columns of A .

Reshaping:

- `reshape(A,m,n)` transforms A into a $(m \times n)$ -matrix.
- `diag(A)` extracts the diagonal a_{jj} of the matrix A to a column vector.
- `diag(v)` generates a matrix with the vector v as a main diagonal.
- `diag(v,k)` generates a matrix with the vector v as the k -th diagonal.
- `A(k,:) = zv` assigns to the k -th row of A the row vector zv .
- `A(k,:) = []` clears the k -th row of A .
- `A(:,k) = sv` assigns to the k -th column of A the column vector sv .
- `A(:,k) = []` clears the k -th column of A .

Examples:

```
octave:1> A = [1 2 3; 4 5 6]; v = [7; 8];
```

```
octave:2> A(2,3) = v(2)
```

```
A =
```

```
 1  2  3
 4  5  8
```

```
octave:3> A(:,2) = v
```

```
A =
```

```
 1  7  3
 4  8  8
```

```
octave:4> A(1,1:2) = v'
```

```
A =
```

```
 7  8  3
 4  8  8
```

2.6 Solving linear systems of equations

- `A\b` solves the equation $Ax = b$.

2.7 Inverses, Decompositions, Eigenvalues, etc.

- `B = inv(A)` calculates the inverse of A .
- `[L,U,P] = lu(A)` calculates a LU-decomposition $LU = PA$.
- `[Q,R] = qr(A)` calculates the QR-decomposition $QR = A$.
- `R = chol(A)` calculates the Cholesky decomposition of A .
- `S = svd(A)` calculates the singular values of A .
- `H = hess(A)` brings A to the Hessenberg form.
- `E = eig(A)` calculates the eigenvalues of A .
- `[V,D] = eig(A)` calculates a diagonal matrix D , containing the eigenvalues and a matrix V with the corresponding eigenvectors. $AV = VD$ also applies.
- `norm(X,p)` calculates the p -norm of vector X . If X is a matrix, p can also take the values 1, 2 or `inf`. The default is $p = 2$.
- `cond(A)` calculates the condition number of A in 2-norm.

A lot of these commands support further options. There can be listed with `help funcname`.

2.8 Test of zero entries

- `[i,j,v] = find(A)` finds the indices of all the elements $\neq 0$ in **A**. $A_{i(l),j(l)} = v_l \neq 0$ also holds.
- `any(A)` is 1, if **A** contains $\neq 0$ elements. For matrices `any` operates on columns.

3 Control structures

3.1 Functions

Traditional are functions and also text data with the suffix `.m`. In contrast to scripts, functions can be called with arguments, and all variables used within the function are *local* (they do not influence the previously defined variables).

Example: A function `f`, saved in the data `f.m`.

```
function y = f (x)
    y = cos(x/2)+x;
end
```

Remark: In Octave a lot of user defined function can be integrated into the script data whereas Matlab allows strictly for one function for the `.m` data, and the name of the function must coincide with the name of the data. Since compatibility with Matlab is important, this restriction should also be applied to programs written in Octave.

Example with two function values: A function `dolittle`, which is saved in the data `dolittle.m`.

```
function [out1,out2] = dolittle (x)
    out1 = x^2;
    out2 = out1*x;
end
```

Calling the function:

```
octave:1> [x1,x2]=dolittle(2)
x1 = 4
x2 = 8
octave:2> whos
*** currently compiled functions:
prot type          rows  cols  name
====  =====
 wd  user function   -     -   dolittle
*** local user variables:
prot type          rows  cols  name
====  =====
 wd  real scalar     1     1   x1
 wd  real scalar     1     1   x2
```

Obviously, the remaining variables `out1`, `out2` were local in `dolittle`. A previously defined variable `out1` or `out2` will not be influenced through the calling of the function.

3.2 Global Variables

`global name` declares `name` as a global variable.

Example: A function `foo` in the data `foo.m`:

```
function out = foo(arg1,arg2)
global N % puts N to a global variable.
<Computation>
end
```

Should `N` be changed in the function, it also changes in the initial value of `N` in "workplace".

3.3 Loops

The syntax of *for*- and *while*-loops will be clear in the following example:

```
for n = 1:10
    [x(n),y(n)]=dolittle(n);
end

while t<T
    t = t+h;
end
```

For-loop backwards:

```
for n = 10:-1:1
    ...
end
```

3.4 Branching:

Conditional branching is programmed as following:

```
if x==0
    error('x is 0!');
else
    y = 1/x;
end

switch pnorm
case 1;
    sum(abs(v))
case inf;
    max(abs(v))
otherwise
    sqrt(v'*v)
end
```

3.5 Functions of functions

- `eval(string)` evaluates `string` as Octave-Code.
- `feval(funcname, arg1, ...)` is equivalent to calling of `funcname` with the arguments `arg1, ...`.

Example Assessment with the midpoint rule:

$$\int_a^b f(x) dx \approx \frac{b-a}{N} \sum_{j=0}^{N-1} f\left(a + \left(j + \frac{1}{2}\right) \frac{b-a}{N}\right)$$

The functions `gauss.m` and `mpr.m` would be defined as follows:

```
function y = gauss(x)
    y = exp(-x.^2/2);
end

function S = mpr(fun,a,b,N)
    h = (b-a)/N;
    S = h*sum(feval(fun,[a+h/2:h:b]));
end
```

Now the function `gauss` can be integrated through the following calling:

```
octave:1> mpr('gauss',0,5,500)
```

3.6 Efficiency considerations

Loops and function callings, especially through `feval`, are very have a high cost. Therefore, as possible, vectorize all the operations.

Example: We are programming the midpoint rule from the previous section with a *for*-loop (Data `mpr_long.m`):

```
function S = mpr_long(fun,a,b,N)
    h = (b-a)/N; S = 0;
    for k = 0:(N-1),
        S = S + feval(fun,a+h*(k+1/2));
    end
    S = h*S;
end
```

The evaluation times are compared like this:

```
octave:1> t = cputime;
> Int1=mpr('gauss',0,5,500); t1=cputime-t;
octave:2> t = cputime;
> Int2=mpr_long('gauss',0,5,500); t2=cputime-t;
octave:3> Int1-Int2, t2/t1
ans = 0
ans = 45.250
```

3.7 Input and Output

- `save data var1 [var2 ...]` saves the variables `var1` etc. in the data data ab.
- `load data` reads the data `data`.
- `fprintf(string[,var1,...])` is strongly based on the syntax of C, see `man fprintf` under Unix.
- `format [long|short]` enlarges or reduces the number of produced decimals. The calling without argument restores the default.

Example:

```
octave:1> for k = .1:.2:.5,
> fprintf('1/%g = %10.2e\n',k,1/k); end
1/0.1 = 1.00e+01
1/0.3 = 3.33e+00
1/0.5 = 2.00e+00
```

4 Graphics

4.1 2D-Graphics

- `plot(x,y[,fmt])` plots a line through the points (x_j, y_j) . With the string `fmt` you can opt for line art and color (see `help plot`).
- `semilogx(x,y[,fmt])` syntax like `plot`, the x axis is logarithmically plotted.
- `semilogy(x,y[,fmt])` syntax like `plot`, the y axis is logarithmically plotted.
- `loglog(x,y[,fmt])` syntax like `plot`, both axes are logarithmically plotted.

For plotting a function $y = f(x)$ the procedure is:

1. A vector with the x -coordinates is generates all the points to be:

```
x = x_min:schrittweite:x_max;
```

(See also section 2.1.)

2. A vector is associated with the corresponding y -values, by leaving the function f to act upon the x -vector element wise:

```
y = f(x);
```

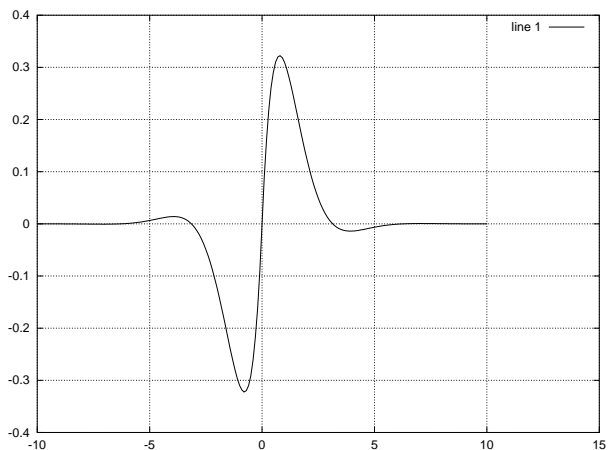
Important: Since f operates element wise, you have to make use of the operators `.*`, `./`, `.^` etc. instead of the usual `+`, `-` und `^`! (See section 2.4.)

3. Now you can call the plot command:

```
plot(x,y); grid; replot
```

Example:

```
octave:1> x = -10:.1:10;  
octave:2> y = sin(x).*exp(-abs(x));  
octave:3> plot(x,y); grid; replot
```

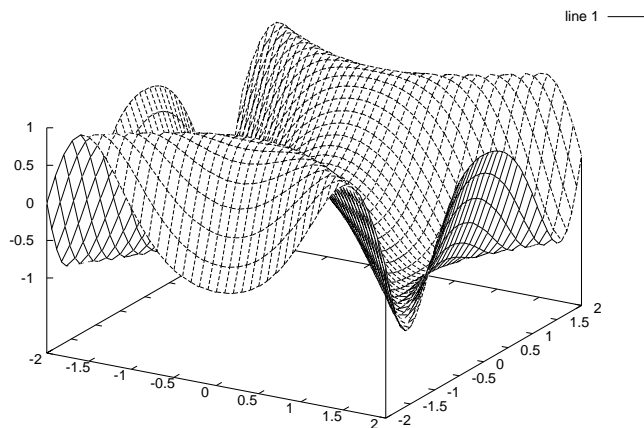


4.2 3D-Graphics:

- `[xx,yy] = meshgrid(x,y)` generates from the vectors x and y grid data.
- `mesh(x,y,z)` plots a surface in 3D.

Example:

```
octave:1> x = -2:0.1:2;  
octave:2> [xx,yy] = meshgrid(x,x);  
octave:3> z = sin(xx.^2-yy.^2);  
octave:4> mesh(x,x,z);
```



4.3 Commands for 2D- and 3D-Graphics:

- `title(string)` writes `string` as title for the graphics.
- `xlabel(string)` labels the x -axis `string`.
- `ylabel(string)` labels the y -axis `string`.
- `zlabel(string)` labels the z -axis `string`.
- `axis(v)` keeps the section; `v` is a vector of form `v = (xmin, xmax, ymin, ymax[, zmin zmax])`.
- `hold [on|off]` establishes whether the next graphic output should or not clear the previous graphics.
- `clf` clears the graphic window.
- `replot` produces a renewed output of the present graphics. This is necessary in Octave when altering an existing graphics. (This command exists only in Octave and it is also necessary only there.)

5 Exercises

5.1 Linear Algebra

Take a matrix A and a vector b with

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} 36 \\ 88 \end{pmatrix}.$$

Now solve the system of equations $Ax = b$. Then calculate die LU- and QR-decompositions an also the eigenvalues and eigenvectors of A . Compute the Cholesky decomposition of $A^T A$, and make sure that $\text{cond}(A^T A) = \text{cond}(A)^2$.

Solving suggestion:

```
A = reshape(1:4,2,2).'; b = [36; 88];
A\b
[L,U,P] = lu(A)
[Q,R] = qr(A)
[V,D] = eig(A)
A2 = A.'*A;
R = chol(A2)
cond(A)^2 - cond(A2)
```

5.2 Timing

Compute a matrix-vector product of a (100×100) arbitrary matrix with an arbitrary vector, such that you once use the built in function `*`, and the other time the *for*-loop. Compare the solving times for the two solutions.

Solving suggestion :

```
A = rand(100); b = rand(100,1);
t = cputime;
v = A*b; t1 = cputime-t;
w = zeros(100,1);
t = cputime;
for n = 1:100,
    for m = 1:100
        w(n) = w(n)+A(n,m)*b(m);
    end
end
t2 = cputime-t;

norm(v-w), t2/t1
ans = 0
ans = 577.00
```

5.3 Stability functions of BDF-Integrators

Calculate the all the roots of the polynomial

$$\frac{147}{60}\zeta^6 - 6\zeta^5 + \frac{15}{2}\zeta^4 - \frac{20}{3}\zeta^3 + \frac{15}{4}\zeta^2 - \frac{6}{5}\zeta + \frac{1}{6}$$

Hint: use the command `compan`.

Plot these roots as points in the complex plane and add also a unit circle.
(Hint: `hold`, `real`, `imag`).

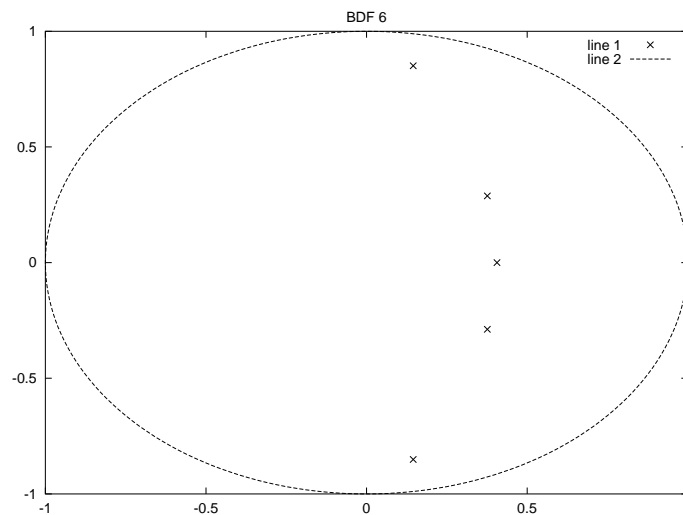
Solving suggestion:

```
bdf6 = [147/60 -6 15/2 -20/3 15/4 -6/5 1/6];
R = eig(compan(bdf6));
plot(R,'+'); hold on
plot(exp(pi*i*[0:.01:2]));
```

```

if any(find(abs(R)>1))
    fprintf('BDF6 ist instabil\n');
else
    fprintf('BDF6 ist stabil\n');
end

```



5.4 3D-Plot

Plot the graphic of the function:

$$f(x, y) = \exp(-x^2 - y^2).$$

Solving suggestion:

```

x = -3:0.1:3;
[xx,yy] = meshgrid(x,x);
z = exp(-xx.^2-yy.^2);
mesh(x,x,z);
title('exp(-x^2-y^2)');
replot;

```

5.5 Hilbert matrix

Calculate for the $(n \times n)$ -Hilbert matrix H ($n = 1, \dots, 15$) the solution for the equation system $Hx = b$, $b = \text{ones}(n, 1)$. Calculate the error and the condition of the matrix and plot them both in a window half logarithmically scaled. (Hint: `hilb`, `invhilb`.)

Solving suggestion:

```

err = zeros(15,1); co = zeros(15,1);
for k = 1:15
    H = hilb(k);
    b = ones(k,1);
    err(k) = norm(H\b-invhilb(k)*b);
    co(k) = cond(H);
end
semilogy(1:15,err,'r',1:15,co,'x');

```

5.6 Best fit straight line

Calculate for the arbitrary predetermined points (x_j, y_j) , generated by the x and y vectors, the best fit straight line. Plot the points and the line.

Solving suggestion:

```

function coeff = ausgl(x,y)
    n = length(x);
    A = [x ones(n,1)];
    coeff = A\y;
    plot(x,y,'x');
    hold on
    interv = [min(x) max(x)];
    plot(interv,coeff(1)*interv+coeff(2));
end

```

5.7 Trapezoidal rule

Write a program, such that an arbitrary function f with one variable on an interval $[a, b]$ is numerically integrated. Make use of the trapezoidal rule ($h = (b - a)/N$):

$$\int_a^b f(x) dx \approx h \left(\frac{f(a) + f(b)}{2} + \sum_{j=1}^{N-1} f(a + jh) \right).$$

Check in a double logarithmical plot by means of an arbitrary function f , that the trapezoidal rule has the order 2.

Solving suggestion:

```

function S = trapez(fun,a,b,N)
    h = (b-a)/N;
    % fy = feval(fun,[a:h:b]); better:
    fy = feval(fun,linspace(a,b,N+1));
    fy(1) = fy(1)/2;
    fy(N+1) = fy(N+1)/2;
    S = h*sum(fy);
end

function y = f(x)

```

```
    y = exp(x);
end

for k=1:15;
    err(k) = abs(exp(1)-1-trapez('f',0,1,2^k));
end
loglog(1./2.^[1:15],err);
hold on;
loglog(1./2.^[1:15],err,'x');
title('Trapezoidal rule, f(x) = exp(x)');
xlabel('Increment');
ylabel('Error');
replot;
```