Scientific visualization, Vetenskaplig visualisering
MVE080, MAI510

Thomas Ericsson
Computational mathematics
Chalmers/GU

office: L2075
phone: 772 10 91
e-mail: thomas@chalmers.se
www: http://www.math.chalmers.se/Math/Grundutb/CTH/mve080/0809/

Assignments, copies of handouts (lecture notes etc.) can be found at the www-address (look under Diary).

Some of the images from this introduction is not available in the handout, though (they take too much space).
[Image] denotes a missing image (images).

Here comes the syllabus (kursplan):

Aim
The solution of computational problems with the help of computers often generate large data sets. This course deals with how computer graphics can be used to visualize data in order to give a better understanding of the problem and its solution. In simple cases the solution can perhaps be represented as a curve. More complicated problems have solutions in the form of surfaces or volumes, maybe even time dependent. Many mathematical problems may not generate so large data sets but require an understanding of more three dimensions.

Goal
At the conclusion of the course, the participant should find it natural to think in visualization terms, be able to produce insightful graphics in a number of common cases, be quite familiar with Matlab graphics, and be acquainted with OpenGL and ParaView.

Prerequisites
Basic courses in mathematics, numerical analysis, programming and data structures. Basic Matlab programming. This is an introductory course so no prior knowledge of computer graphics is required.

Content
Introduction to visualization. Different techniques for visualizing surfaces, volumes and other common mathematical objects. Animation. Interaction. An orientation about the construction of user interfaces. OpenGL, ParaView and advanced Matlab graphics.

Matlab, easy and to get started. OpenGL to see how some basic computer graphics is done. ParaView, more capable than Matlab (but harder to use).

Computer graphics concepts, such as transformations and shading models, necessary to use and understand the graphics software. A sufficient amount of C to finish the the computer assignments.

Organization
Lectures and computer assignments. The assignments, which make up a substantial part of the course, consist of several problems where the student will use Matlab, OpenGL and ParaView to solve different visualization problems.
The problems are fetched from numerical analysis and applied mathematics.

The assignments vary in difficulty. Some are routine tasks (would take me minutes) while others require a bit of programming.

Literature
Lecture notes, articles and manuals.

Reference literature: F. S. Hill, Computer Graphics using OpenGL, 2nd ed., Prentice Hall, 2001 or Edward Angel, Interactive Computer Graphics, A Top-Down Approach with OpenGL, Pearson Education 2003, 3rd ed.

The topic of these titles are not strictly visualization, they are standard computer graphics books. On the next page I will list more literature.

Examination
Compulsory computer assignments and take-home exam.

We have two lectures and two labs per week. See the schedule on www. Show me your solutions to the assignments at lab-times. You do not have to hand in any reports.

More books from my shelf

Here comes a list of books which I collected with this course in mind. For other books, see the references on the home page. Some E-books are available via the Chalmers library home page. Finally there are man-pages and PDF-manuals.

- S. K. Card, J. D. Mackinlay, B. Shneiderman, Readings in Information Visualization: Using Vision to Think Morgan Kaufmann, 1999.
- C. D. Hansen, C. R. Johnson (eds.), Visualization Handbook av Johnson, Academic Press, 2004.
- R. Spence, Information Visualization, Addison-Wesle, 2001.
- D. Thompson, J. Braun, R. Ford, OpenDX: Paths to Visualization, 2nd ed. 2004, http://www.vizsolutions.com
- D. A. Norman, The Design of Everyday Things, Basic Books, 2002.
- C. Ware, Information Visualization Perception for Design, Elsevier, 2004.
- M. K. Agoston, Computer Graphics & Geometric Modeling, Implementation and Algorithms, Springer, 2004. There is one Computer Graphics and Geometric Modeling: Mathematics, which I do not have.
- S. R Buss, 3D Computer Graphics: A Mathematical Introduction with OpenGL, Cambridge UP, 2003.
- H. C. Hege, K. Polthier (eds.), Mathematical Visualization, Algorithms, Applications and Numerics, Springer, 1998.
- J. O'Rourke, Computational Geometry In C, Cambridge UP, 1998.
- D. F. Rogers, An Introduction to Nurbs: With Historical Perspective Morgan Kaufmann, 2000.
- A. Unwin, M. Theus, H. Hofmann, Graphics of Large Datasets, Springer, 2006. To the math-library.

Some typical visualization problems

The primary goal of Scientific Visualization, is to provide insight into scientific data. We often need a deeper understanding of a phenomenon, need to draw conclusions, make predictions. (Computer) graphics can (often) give us the help we need, after all:

"An image says more than a thousand words (or numbers)"

Scientific visualization usually has a natural physical or mathematical representation or background. We may want to visualize the flow of air around aircraft or the roots of an equation. When visualizing the data, we would probably make an outline of the aircraft and draw a coordinate system for our roots. [Image]

A related area is information visualization. It is less common with a physical background and it may not even be important. A classical example is Harry Beck's map of the London underground (1931). [Image]
See `http://en.wikipedia.org/wiki/Harry_Beck` for example.

Previous maps based on the actual layout, the geography, of the underground had not worked well. Beck's map, on the other hand, leaves the physical reality behind and shows the order of stations, where lines cross etc. It captures what is essential for the traveler.

Another example is given by business graphics (pie charts etc.), e.g. visualizing the number of admitted and graduated students for different programmes at Chalmers/GU.

This course will deal with scientific visualization.
You have already dealt with this in previous classes. Plotting the graph of a scalar function of a scalar variable, `plot(x, y)` provides almost a complete understanding of the function.

There are however harder visualization problems, where we only get a partial understanding, e.g. looking at $w = f(x, y, z)$, given a function f. [Image] Understanding $w = f(x, y, z, t)$ completely may be hopeless.

Another cause of problems may be the amount of data.
Computers are fast, and when a program has executed a few hours the output can be enormous, several Gbytes.
To visualize this amount of data may be difficult, but a thousand numbers may be hard enough.

It is not always easy to say what is a meaningful image.
Tastes differ as does the ability to interpret 3D-plots, for example. So this course will show different ways of visualizing data, but there is rarely a unique solution to a visualization problem (or to the assignments).

Use your imagination. If one plot is not that helpful there may be another, better, way to visualize the data. Trial and error may be a successful method.

An example, the cosine function

In Matlab it is possible to compute $\cos z$ where z is a complex number. Suppose we would like to understand how this function behaves. Since we know a lot of mathematics we can easily list several properties.

Let a, b be real numbers, then

$$\cos(a+ib) = \frac{e^{i(a+ib)} + e^{-i(a+ib)}}{2} = \cdots =$$

$$\frac{(e^b + e^{-b})\cos a}{2} - i\,\frac{(e^b - e^{-b})\sin a}{2}$$

If $z \in C$ then the following is true, for example:

$$\cos(z + 2\pi) = \cos z, \quad \cos z = \cos(-z), \quad \cos \bar{z} = \overline{\cos z}$$

So, it is sufficient to study $0 \le Re(z) \le 2\pi$ and $Im(z) \ge 0$.

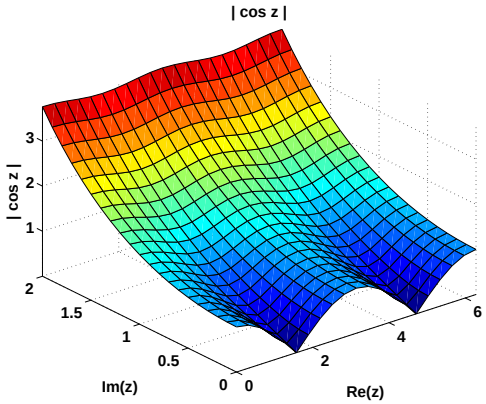
For large b

$$|\cos z| \approx \frac{e^b|\cos a - i\sin a|}{2} = \frac{e^b}{2}$$

In a real application it may not be possible to use mathematics this way. Perhaps the function is too complicated, or perhaps worse, we may not have an expression for the function. We may have to rely on a computer program that returns $f(z)$ for a given z.

The visualization of $\cos z$ is still a bit hard since we are dealing with four real dimensions. Here are a few alternatives (not all good).

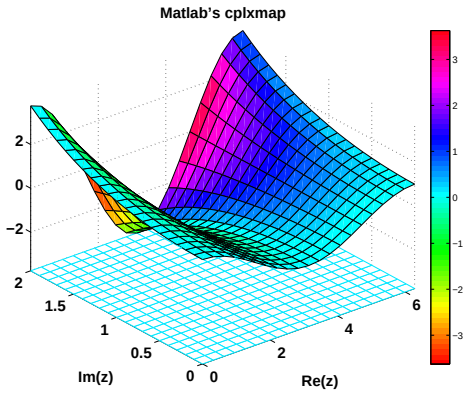
The obvious first try, plot $|\cos z|$ as a function of $(Re(z), Im(z))$.



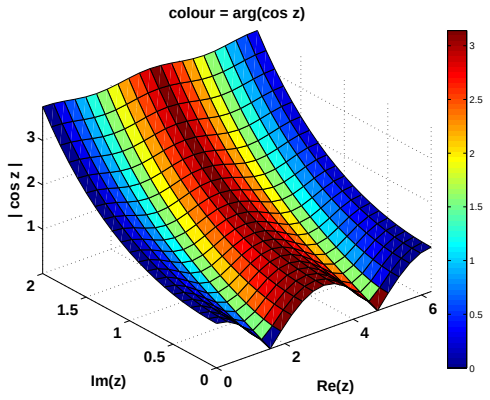
In Matlab this would be in colour, where the colour corresponds to $|\cos z|$.

It captures some of the behaviour: periodicity, what happens for large $Im(z)$. We have lost the sign information, and introduced corners (like $x \to |x|$).
On the other hand, this image may be exactly what we need. It is possible to use more fancy graphics, no grid but a smooth surface using light etc. [Image]

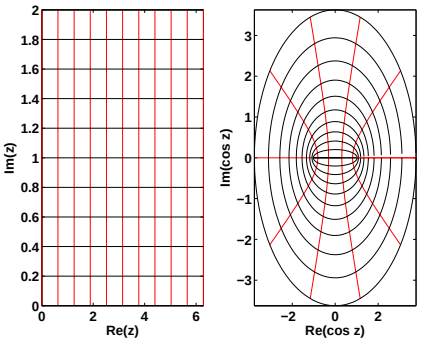
The next image was done with Matlab's cplxmap-command. It plots $Re(\cos(z))$ as a function of $(Re(z), Im(z))$. The colour is used for $Im(\cos(z))$. I have added a color bar. I have a problem with this plot. The shape of the surface dominates over the colour information.



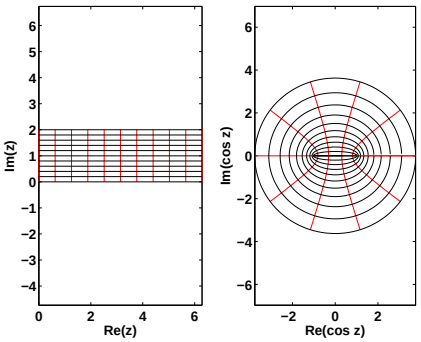
A similar idea is to plot $|\cos z|$ as before but to let the colour show the argument, so if $\cos z = r\ e^{i\varphi}$ we use colour for φ and height for $r = |\cos z|$.



In the next plot we do not loose any space-dimension. A grid in the $(Re(z), Im(z))$-plane is mapped onto $(Re(\cos z), Im(\cos z))$. We see the periodicity in a new way. Lines with constant imaginary-part seem to be mapped onto closed curves.



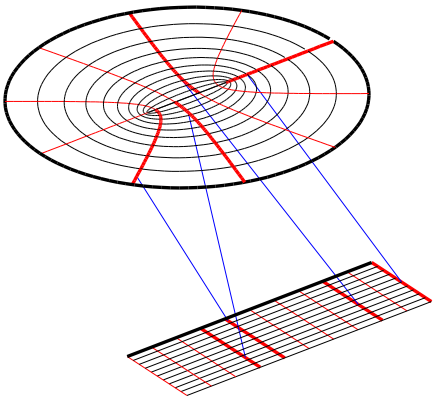
The plot is not quite truthful. Matlab tries to fill out the window, which may cause different scaling between the axes (a circle may look like an ellipse). After correction, `axis equal`, we see some new features.



It seems like we have right angles between the curves in the right diagram. So are the angles in the z-grid preserved? Yes. Those who have read complex analysis may recognize this as a special case of a more general theorem. cos is a conformal mapping and hence preserves angles (whenever the derivative is non-zero).

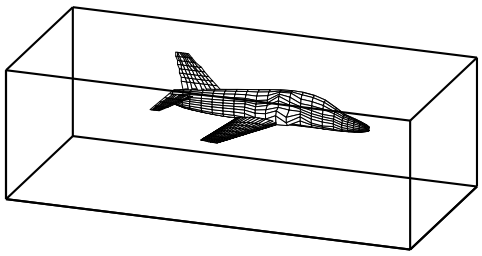
One drawback with this plot is that is hard to know what line corresponds to which $\cos z$-curve. Perhaps we could use some interaction with the mouse, clicking on a line in the left window would make the corresponding curve in the right window blink, change colour or something.

picking



We end with two images where we plot $Re(\cos z)$ and $Im(\cos z)$ in two windows or together in one. [Image].
Several other alternatives remain.

Now to another problem, mesh generation in 3D. The difficulty is not the number of dimensions this time, but the huge amount of data.



Discretize (divide into small volume elements) the air in the box and outside the aircraft. Mesh generation (using `m3d`, an ITM-project, Swedish Institute of Applied Mathematics) on one RS6000 processor:

```
wed may 29 12:54:44 metdst 1996  So this is old stuff
thu may 30 07:05:53 metdst 1996

183463307 may  2 13:46 s2000r.mu

tetrahedrons in structured mesh:    4 520 413
tetrahedrons in unstructured mesh: 4 811 373
```

Does the program work? Does it refine the mesh it the right places? Make nice images for the annual report (and for those supplying the money). [Image] (several).

There are of course many other visulization problems. Here are a few [Image] showing a simulation of an open cavity problem. Others will turn up on the lectures or in the labs.

Starting Matlab

```
> matlab -help

Here is an edited list:


-h|-help          - Display arguments.
-nodisplay        - Do not display any X commands. The
                    MATLAB desktop will not be started.
                    However, unless
                    -nojvm is also provided the Java
                    virtual machine will be started.
-nosplash         - Do not display the splash screen
                    during startup.
-nodesktop        - Do not start the MATLAB desktop.
                    Use the current terminal for
                    commands. The Java virtual
                    machine will be started.
-nojvm            - Shut off all Java support by not
                    starting the Java virtual machine.
                    In particular the MATLAB
                    desktop will not be started.
```

I use `matlab -nodesktop`.
To get short help, type `help command`. For more help use the GUI (Graphical User Interface) or `doc command`. There are thick PDF-manuals available (through the GUI) as well. Start Help and click on MATLAB, choose Printable (PDF) Documentation. The basic graphics manual is 667 pages and the 3D-manual an additional 212 pages.

For this to work you have to tell MATLAB what browser you are using. Netscape is default (and we do not have it). This is one way to fix it:


```
cd
mkdir matlab
cd matlab
cp /chalmers/sw/sup/matlab-2007b/toolbox/
    local/docopt.m .  (I have broken the line)

edit docopt.m and change line 77 in the file
      doccmd = 'netscape';
to
     doccmd = 'mozilla';    or
     doccmd = 'firefox';
```

Programming in Matlab

- A full programming language, if, for,...
- The basic datatype, a double precision matrix in several dimensions. In Matlab7 there are more types, such as single precision and integers.
- No type declarations. variables are created when needed.
- Interactive. Partially interpreted.
- New programming style; vector based.
- Object oriented (to some extent).
- Easy to use graphics.
- Can add toolboxes and compiled code.

A tutorial is available. Look under MATLABs help. You can also see the Matlab-book by Jönsson (Swedish).
One should learn to work with vectors and matrices instead of using loops and elements. Shorter, faster and easier to read. It is convenient to write the labs as m-files (instead of typing commands and using the history mechanism).

On the following pages comes a short and fast review of Matlab. There will probably be new things for you as well. Some of the commands below can be performed using the GUI instead.

```
>> v = [1 -5 7 8 -3]  % or comma as delimiter
v =
     1    -5     7     8    -3

>> a = v(2) + v(5)
a =
    -8

>> v(2) = 25
v =
     1    25     7     8    -3

>> v(2) + v(3)
ans =   % default "answer", % = comment
    32

>> who

Your variables are:

a         ans       v

>> sin(v(1))
ans =
    0.8415

>> format short e
>> sin(v(1))
ans =
   8.4147e-01
```



```
>> format long e
>> sin(v(1))
ans =
     8.414709848078965e-01

>> format short
>> sin(v(1))
ans =
    0.8415

>> format bank
>> sin(v(1))
ans =
          0.84

>> format hex
>> sin(v(1))
ans =
   3feaed548f090cee

>> format compact  % for less space

>> help format

 FORMAT Set output format.
  etc.

>> doc format    % opens Matlab's browser
                 % (or use the GUI)
```

Note that this changes the output format and not the internal binary representation.

```
>> w = 1:6
w =
     1     2     3     4     5     6

>> w = 1:2:8
w =
     1     3     5     7

>> w = 7:-2:-4
w =
     7     5     3     1    -1    -3

>> w = 7:-2:8
w =
   Empty matrix: 1-by-0

>> 1.5:0.856:6.7  % complex numbers do not work
ans =
 1.5000 2.3560 3.2120 4.0680 4.9240 5.7800 6.6360

>> w = [1; 2; 3]
w =
     1
     2
     3

>> w = [1; 2; 3];   % no printing
>> w
w =
     1
     2
     3
```



```
>> w = 1; 2; 3     % ; separates commands
ans =
     3
>> w
w =
     1

>> a = 1:3; b = 5:7;
>> c = a + b
c =
     6     8    10

>> a = 1:3; b = 5:8;
>> c = a + b
??? Error using ==> plus
Matrix dimensions must agree.

>> size(a)
ans =
     1     3    % size(a, 2) is 3 etc.
>> size(b)
ans =
     1     4

>> b = (5:7)'
b =
     5
     6
     7
```



```
>> c = a + b
??? Error using ==> plus
Matrix dimensions must agree.

>> size(b)
ans =
     3     1

>> a = a'
a =
     1
     2
     3
>> c = a + b
c =
     6
     8
    10

>> sqrt(c')
ans =
    2.4495    2.8284    3.1623

>> a = 1:3, b = 10 * (3:-1:1)
a =
     1     2     3
b =
    30    20    10

>> a * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> a .* b
ans =
    30    40    30
```



```
>> a ./ b
ans =
    0.0333    0.1000    0.3000

>> a / b   % something different
ans =
    0.0714

>> a .\ b
ans =
   30.0000   10.0000    3.3333

>> a \ b
ans =
         0         0         0
         0         0         0
   10.0000    6.6667    3.3333


>> a \. b
??? a \.
       |
Error: Unexpected MATLAB operator.

>> a .^ b
ans =
           1     1048576       59049

>> a.^2 .* b.^3
ans =
       27000       32000        9000

>> 1 + a
ans =
     2     3     4
```

```
>> 1 ./ a
ans =
   1.0000e+00   5.0000e-01   3.3333e-01


>> i
ans =
        0 + 1.0000i
>> j
ans =
        0 + 1.0000i
>> sqrt(-1)
ans =
        0 + 1.0000i

>> q = [1+i 2-3*i 6+6*i]   % 2-3i works as well
q =
 1.0000 + 1.0000i  2.0000 - 3.0000i  6.0000 + 6.0000i

>> q'
ans =
   1.0000 - 1.0000i
   2.0000 + 3.0000i
   6.0000 - 6.0000i

>> q.'
ans =
   1.0000 + 1.0000i
   2.0000 - 3.0000i
   6.0000 + 6.0000i

>> real(q)   % is applied on the whole vector
ans =
     1     2     6
```



```
>> imag(q)
ans =
     1    -3     6

>> abs(q)
ans =
    1.4142    3.6056    8.4853

>> exp(i * pi) % pi is predefined
ans =
  -1.0000 + 0.0000i

>> format short e
>> exp(i * pi)
ans =
 -1.0000e+00 + 1.2246e-16i

>> sqrt(2)^2 - 2
ans =
   4.4409e-16

>> sin(pi)
ans =
   1.2246e-16

>> v = 1:10

v =
  1  2  3  4  5  6  7  8  9  10

>> s = 0;
>> for k = 1:10
     s = s + v(k);
   end
>> s
s =
    55
```



```
>> s = sum(v)  % there is prod as well
s =
    55
```

Matrices

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

>> A'
ans =
     1     4
     2     5
     3     6

>> A(2, 3) = 66
A =
     1     2     3
     4     5    66
```



```
>> A(3, 3) = 9   % A is increased dynamically
A =
     1     2     3
     4     5    66
     0     0     9

>> 1 + A(3, 4)
???  Index exceeds matrix dimensions.

>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> B = [3 4; 1 2]
B =
     3     4
     1     2

>> A * B
ans =
     5     8
    13    20

>> A + B
ans =
     4     6
     4     6

>> A .* B
ans =
     3     8
     3     8
```

```
>> A ./ B
ans =
   3.3333e-01   5.0000e-01
   3.0000e+00   2.0000e+00

>> A .\ B
ans =
   3.0000e+00   2.0000e+00
   3.3333e-01   5.0000e-01

>> A / B   % roughly A * inv(B)
ans =
     0     1
     1     0

>> A \ B   % roughly inv(A) * B
ans =
  -5.0000e+00  -6.0000e+00
   4.0000e+00   5.0000e+00

>> A^2
ans =
     7    10
    15    22

>> A.^2
ans =
     1     4
     9    16

>> A.^A
ans =
     1     4
    27   256
```



```
>> sqrt(A)
ans =
   1.0000e+00   1.4142e+00
   1.7321e+00   2.0000e+00

>> sqrt(-A)
ans =
           0 + 1.0000e+00i           0 + 1.4142e+00i
           0 + 1.7321e+00i           0 + 2.0000e+00i

>> R = rand(3)
R =
   9.5013e-01   4.8598e-01   4.5647e-01
   2.3114e-01   8.9130e-01   1.8504e-02
   6.0684e-01   7.6210e-01   8.2141e-01

>> R = rand(3, 2)     % rand
R =
   4.4470e-01   9.2181e-01
   6.1543e-01   7.3821e-01
   7.9194e-01   1.7627e-01

>> R = randn(3, 2)    % NOTE randN
R =
  -1.9790e-02   2.5730e-01
  -1.5672e-01  -1.0565e+00
  -1.6041e+00   1.4151e+00

>> D = diag(1:2:5)  % diag(matrix) returns the
D =                 % diagonal in a vector
     1     0     0
     0     3     0
     0     0     5
```



```
>> D = diag(1:2:5, -1) + diag(1:2:5, 1)
D =
     0     1     0     0
     1     0     3     0
     0     3     0     5
     0     0     5     0

>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1

>> B = magic(3)
B =
     8     1     6
     3     5     7
     4     9     2

>> IB = inv(B)
IB =
   1.4722e-01  -1.4444e-01   6.3889e-02
  -6.1111e-02   2.2222e-02   1.0556e-01
  -1.9444e-02   1.8889e-01  -1.0278e-01

>> B * IB
ans =
   1.0000e+00            0  -1.1102e-16
  -2.7756e-17   1.0000e+00            0
   6.9389e-17            0   1.0000e+00

>> IB * B
ans =
   1.0000e+00            0  -2.7756e-17
            0   1.0000e+00            0
            0   1.1102e-16   1.0000e+00
```

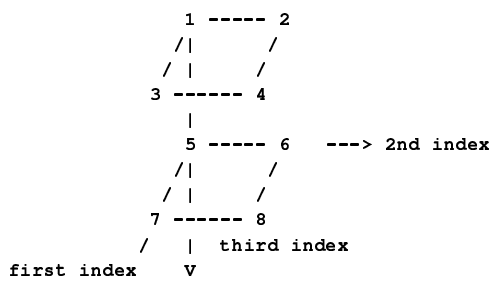


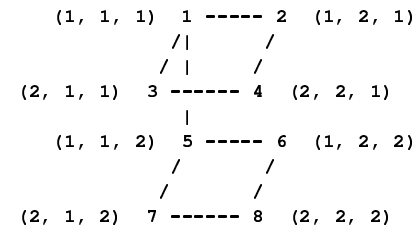
```
>> ones(2, 3)
ans =
     1     1     1
     1     1     1

>> zeros(2)
ans =
     0     0
     0     0

>> S = reshape(1:6, 2, 3)
S =
     1     3     5
     2     4     6

>> sum(S)
ans =
     3     7    11

>> sum(S')
ans =
     9    12

>> sum(S, 2)
ans =
     9
    12

>> sum(sum(S))
ans =
    21

>> cumsum(1:7)
ans =
     1     3     6    10    15    21    28
```

```
>> M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2

>> sort(M)
ans =
     3     1     2
     4     5     6
     8     9     7

>> M(:)'
ans =
     8     3     4     1     5     9     6     7     2

>> s = sort(ans)
s =
     1     2     3     4     5     6     7     8     9
```


There are matrices of higher order:

```
>> A1 = [1 2;3 4]
A1 =
     1     2
     3     4

>> A2 = [5 6; 7 8]
A2 =
     5     6
     7     8

>> A(:,:,1) = A1;
>> A(:,:,2) = A2;

>> A
A(:,:,1) =
     1     2
     3     4
A(:,:,2) =
     5     6
     7     8
```



Index vectors

```
>> v = 0.1 + (1:7)
v =
    1.1    2.1    3.1    4.1    5.1    6.1    7.1

>> v(1:3:7)   % 1:3:7 = [1 4 7]
ans =
    1.1    4.1    7.1

>> M = magic(5)
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> M(:, 2)
ans =
    24
     5
     6
    12
    18

>> M([2 5], :)
ans =
    23     5     7    14    16
    11    18    25     2     9

>> M([2 5], [2 4])
ans =
     5    14
    18     2
```

end is practical in constructions like these:

```
>> M(:, end)
ans =
    15
    16
    22
     3
     9

>> M(end, :)
ans =
    11    18    25     2     9

>> M(end, end)
ans =
     9

>> M([1 3], [end-3:end])
ans =
    24     1     8    15
     6    13    20    22
```

An alternative is of course:

```
>> [m, n] = size(M)
m =
     5
n =
     5

>> M(m, :)
ans =
    11    18    25     2     9
```


A bit more original is:

```
>> M(:, [1 1 2])
ans =
    17    17    24
    23    23     5
     4     4     6
    10    10    12
    11    11    18
```

Is used by **meshgrid**.

```
>> x = 1:3
x =
     1     2     3

>> y = -2:0
y =
    -2    -1     0

>> [X, Y] = meshgrid(x, y)
X =
     1     2     3
     1     2     3
     1     2     3
Y =
    -2    -2    -2
    -1    -1    -1
     0     0     0
```

Can be computed this way:

```
>> x = x(:)'; X = x(ones(length(y), 1), :)
X =
     1     2     3
     1     2     3
     1     2     3

>> y = y(:); Y = y(:, ones(1, length(x)));
```


I used this quite often:

```
>> [X, L] = eig(M)
X =
    0.4472   -0.6780   -0.6330    0.0976    0.2619
    0.4472   -0.3223    0.5895    0.3525    0.1732
    0.4472    0.5501   -0.3915    0.5501   -0.3915
    0.4472    0.3525    0.1732   -0.3223    0.5895
    0.4472    0.0976    0.2619   -0.6780   -0.6330

L =
   65.0000         0         0         0         0
         0   21.2768         0         0         0
         0         0  -13.1263         0         0
         0         0         0  -21.2768         0
         0         0         0         0   13.1263

>> [l, pntr] = sort(diag(L))
l =
  -21.2768
  -13.1263
   13.1263
   21.2768
   65.0000

pntr =
     4
     3
     5
     2
     1

>> X = X(:, pntr);
```


min can return a pointer vector as well. Suppose we would like to find the row- and column indices for the largest element in a matrix (we assume it is unique).

```
>> M
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> [col_max, row_p] = max(M)
col_max =
    23    24    25    21    22
row_p =
     2     1     5     4     3

>> [max_M, col_p] = max(col_max)
max_M =
    25
col_p =
     3

>> M(row_p(col_p), col_p)
ans =
    25
```

```
>> M(1:2, 3:4) = M([2 5], [2 4])
M =
    17    24     5    14    15
    23     5    18     2    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> A = ones(3, 1) * (1:3)
A =
     1     2     3
     1     2     3
     1     2     3

>> B = A(:, 3:-1:1)
B =
     3     2     1
     3     2     1
     3     2     1

>> A = A'
A =
     1     1     1
     2     2     2
     3     3     3

>> C = A(3:-1:1, :)
C =
     3     3     3
     2     2     2
     1     1     1
```

Logical vectors

```
>> v = 0.1 + (1:7)
v =
    1.1    2.1    3.1    4.1    5.1    6.1    7.1
```



```
>> v > 4
ans =
     0     0     0     1     1     1     1
>> v(v > 4)
ans =
    4.1    5.1    6.1    7.1

>> v([0 0 0 1 1 1 1])
??? Subscript indices must either be real positive
    integers or logicals.

>> v(logical([0 0 0 1 1 1 1]))
ans =
    4.1    5.1    6.1    7.1
```

Logical operators:

```
>> v(2 < v & v < 5)
ans =
    2.1    3.1    4.1

>> v(v <= 2.1 | 6 <= v)
ans =
    1.1    2.1    6.1    7.1
```

Count occurrences

```
>> sum(v ~= 3.1)     % == equality, ~= not equal
ans =                % unsafe for floating point
     6

>> any(v ~= 3.1)
ans =
     1

>> all(v ~= 3.1)
ans =
     0
```



```
>> all(v ~= 3.5)
ans =
     1

>> find(v > 4)
ans =
     4     5     6     7

>> M = magic(4)
M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> M > 11
ans =
     1     0     0     1
     0     0     0     0
     0     0     0     1
     0     1     1     0

>> M(M > 11)
ans =
    16
    14
    15
    13
    12

>> i = find(M > 11)
i =
     1
     8
    12
    13
    15
```



```
>> m = M(:);

>> m(i)
ans =
    16
    14
    15
    13
    12

>> [j, k] = find(M > 11)
j =
     1
     4
     4
     1
     3
k =
     1
     2
     3
     4
     4
```

Compute the number of negative numbers in v. Can use loops

```
>> v = randn(100, 1);
>> neg = 0;
>> for k = 1:100
     if v(k) < 0
       neg = neg + 1;
     end
   end
```

```
>> neg
neg =
    55

>> sum(v < 0)  % but this is nicer.
ans =
    55
```

Creating matrices from parts.

```
>> A = magic(2)
A =
     1     3
     4     2

>> b = [1; 3]
b =
     1
     3
>> C = [A, b; b', 7]
C =
     1     3     1
     4     2     3
     1     3     7

>> b = (1:3)';

>> F = [b b(3:-1:1) [b([3 1]); 10]]
F =
     1     3     3
     2     2     1
     3     1    10
```


Three dimensional matrices

```
>> A1 = [1 2; 3 4] + 0.1;
>> A2 = [5 6; 7 8] + 0.1;
>> A(:,:,1) = A1;
>> A(:,:,2) = A2;

>> A
A(:,:,1) =
    1.1000    2.1000
    3.1000    4.1000
A(:,:,2) =
    5.1000    6.1000
    7.1000    8.1000

>> A(A > 3)
ans =
    3.1000
    4.1000
    5.1000
    7.1000
    6.1000
    8.1000

>> i = find(A > 3)
i =
     2
     4
     5
     6
     7
     8

>> A(:)'
ans =
1.100  3.100  2.100  4.100  5.100  7.100  6.100  8.100
```


`[i, j, k] = find...` does not do anything useful in this case. Here is an alternative using loops:

```
i = []; j = []; k = [];

for r = 1:size(A, 3)
  [row, col] = find(A(:, :, r) > 3);
  i = [i; row];
  j = [j; col];
  k = [k; r * ones(size(row))];
end

ind = [i, j, k]

ind =
     2     1     1
     2     2     1
     1     1     2
     2     1     2
     1     2     2
     2     2     2

v = [];
for i = 1:6
  v(i) = A(ind(i, 1), ind(i, 2), ind(i, 3));
end

v
>> v
v =
  3.1000  4.1000  5.1000  7.1000  6.1000  8.1000
```


Linear systems

```
>> A = [1 -1 1; 1 2 3; 4 5 6]
A =
     1    -1     1
     1     2     3
     4     5     6
>> b = [0 1 0]'
b =
     0
     1
     0
>> x = A \ b
x =
   -0.9167
   -0.1667
    0.7500
>> r = b - A * x
r =
   1.0e-15 *
    0.1110
         0
         0
```

Cell arrays

An array where the elements can be of different types:

```
>> c{1, 1} = sqrt(2);
>> c{1, 2} = [1 2; 3 4];
>> c{2, 1} = 'Hejsan';
>> c{2, 2} = 1:5;

>> c
c =
    [1.4142e+00]    [2x2 double]
    'Hejsan'        [1x5 double]

>> c{1, 2}(2, 2)
ans =
     4

>> celldisp(c)
c{1,1} =
   1.4142e+00
c{2,1} =
Hejsan
c{1,2} =
     1     2
     3     4
c{2,2} =
     1     2     3     4     5

>> cc={1:3, 'hej'; c, eye(2)}
cc =
    [1x3 double]    'hej'
    {2x2 cell  }    [2x2 double]

>> cc{2,1}{1,2}(1, :)
ans =
     1     2
```



```
>> C = cell(2)
C =
     []     []
     []     []
```

Another data structure where we can store elements of different types is the struct (record, post in Sw). We name the element with a string and not an index.

```
>> s = struct('type', 'circle',                          ...
              'geom',  struct('c', [1 3], 'r', 1.2), ...
              'color', [1 0 0])
s =
     type: 'circle'
     geom: [1x1 struct]
    color: [1 0 0]

>> s.type
ans =
circle

>> s.geom
ans =
    c: [1 3]
    r: 1.2000e+00

>> s.geom.c
ans =
     1     3

>> s.geom.c(2)
ans =
     3
```



```
>> s.color(2) = 1;

>> s.color
ans =
     1     1     0
```

One can have arrays of structs (of the same kind)

```
>> v(1).fn = 'Thomas';
>> v(1).en = 'Ericsson';
>> v(2).fn = 'Anders';
>> v(2).en = 'Andersson';

>> v(3).new = 'oops'    % a new member
v =
1x3 struct array with fields:
    fn
    en
    new
>> v(1).new         % all the structs in the array
                    % get this new member
ans =
     []
```


There are if-statements etc.

```
>> a = 2.25;
>> if a > 1
     disp('a > 1')
   else
     disp('a <= 1')
   end

a > 1

>> a = 0.2;
>> if a > 1
     disp('a > 1')
   else
     disp('a <= 1')
   end

a <= 1

>> help if   % for elseif etc.
```

`&&`, `||` for lazy scalar and, or.

```
>> a = 2.25;
>> if a, disp('****'), end
****

>> a = 0;
>> if a, disp('****'), end
```

Handling characters

```
>> s = 'AabcDd'
s =
AabcDd
```

```
>> s + 0   % double(s) works as well
ans =
    65    97    98    99    68   100

>> whos
  Name       Size          Bytes  Class

  s          1x6              12  char array

Grand total is 6 elements using 12 bytes

>> S = [s; s(6:-1:1)]
S =
AabcDd
dDcbaA

>> s = 'sirapiparis';
>> palin = all(s == s(end:-1:1))
palin =
     1

>> s(1)='a';
>> palin = all(s == s(end:-1:1))
palin =
     0

>> s1 = 'ABC';
>> s2 = ' 12';

>> s1 + s2
ans =
    97   115   117

>> char(ans)
ans =
asu
```


Tuning Matlab programs

The timings below are for Matlab version 7. Matlab 6.5, and later versions, has a JIT-accelerator (Just In Time) which speeds up for-loops etc.

- Use the builtin compiled routines.
  The Matlab-language is interpreted.
- Work on the matrix/vector-level, not on element-level.
  "New" programming style.
- Take care when using the dynamic memory allocation.
  Preallocate.

Some examples:

```
% Matrix sum. n = 1500 in all examples

for j = 1:n
  for k = 1:n
    A(j, k) = A(j, k) + B(j, k);
  end
end
```

Takes 0.11 s.

`A = A + B;` requires 0.017 s.


```
clear A
for k = 1:n
  A(:, k) = x;  % could have different arrays
end
```

Takes 25 seconds.

```
A = zeros(n);   % preallocate
for k = 1:n
  A(:, k) = x;
end
```

Takes 0.09 s.

`W` is a 8000 × 15-matrix and `x` is a column vector having 8000 elements.

```
y = W * W' * x;          y = W * (W' * x);

Takes 4.1 s              0.0009 s
```

Note that it may be impossible just to form `W * W'` even though `y = W * (W' * x);` gives no problem.

M-files and functions

- For short tests we may type commands by hand and use the history mechanism, arrow keys etc. to modify statements. Possible to use emacs-commands on the command line. `Ctrl-a` beginning of line, `Ctrl-e` end of line, `Ctrl-d` remove character, `Ctrl-k` kill (remove) the rest of the line etc. Can match the beginning of a string: `im`↑ press up-arrow, matches line starting with `im`.
  For those using the GUI there is a Command History window, as well.
- For longer tests (assignments) we create an m-file script (or a function) with an editor (e.g. Matlab's own). If the filename is `name.m` we execute the file by typing `name` in Matlab.

Scripts do not take any parameters. Matlab just reads from the file instead of reading commands from the command window. Sometimes functions are more useful or necessary. Here is a simple example. We disregard the fact that Matlab has a function for computing the median. We store the function on the file `median.m`. If the name of the function and file are different you have to use the filename to invoke the function.

```
function med = median(v)
% med = median(v) computes the median of
%       the elements in the vector v

  n = length(v);         % number of elements in v
  if n == 0
    med = 0;
  else
    s = sort(v);         % s is local to the function
    if rem(n, 2) == 0
      n2  = n / 2;
      med = 0.5 * (s(n2) + s(n2 + 1));   % even
    else
      med = s((n + 1) / 2);              % odd
    end
  end
```

We can think of the parameters as being passed by "call by value", but "call by reference" is used for variables that are not changed. We could have written

```
    v = sort(v);    % replace v
    ...
    med = 0.5 * (v(n2) + v(n2 + 1));    % even
```

This does not change the array in the calling program. The variables `n`, `n2`, `s` and `med` are local to the function. We give the function a value by giving the return-variable, `med`, a value.

```
>> help median

  med = median(v) computes the median of
        the elements in the vector v

>> v = randn(1, 4)
v =
   -1.8092   -0.6337   -0.4533    0.2840

>> median(v)
ans =
   -0.5435

>> median([v, 5])
ans =
   -0.4533
```

There are several types of functions:

- Anonymous functions (short function not stored in a file)
- Subfunctions (several functions in one file)
- Nested functions (functions inside other functions)
- Overloaded functions (polymorphic functions)
- Private functions (functions in `dir_name/private` are only visible to functions in `dir_name`)

Let us look at the first three types. An anonymous function is created by

```
    fhandle = @(argumentlist) expr
```

`expr` is a simple expression and `@` a so-called function handle.

```
>> f = @(x) x .* exp(-x)
f =
    @(x) x .* exp(-x)

>> f([-1 0 1])
ans =
   -2.7183         0    0.3679

>> quadl(f, 0, 1)  % integrate
ans =
    0.2642

>> sin(f(2))
ans =
    0.2674

% A cell array of functions.
% Be careful with blanks. See the manual

>> funcs = {@(x)x.*exp(-x), @(x)x.*sin(-x), ...
            @(x)x.*cos(-x)};

>> for k=1:3, quadl(funcs{k}, 0, 1), end
ans =
    0.2642
ans =
   -0.3012
ans =
    0.3818
```



```
>> comm = @(A, B) A * B - B * A;
>> C = [1 2; 3 4];
>> comm(C, C')
ans =
    -5    -3
    -3     5

% Using "external" variables
>> a = 10;
>> mul_10 = @(z) a * z
mul_10 =
    @(z) a * z

>> mul_10(2)
ans =
    20

>> a = 20;  % does not change the function
>> mul_10(2)
ans =
    20
```

One disadvantage with ordinary m-file functions is that they tend to produce many files. It is possible to put several functions in one file. The first function in the file, the primary function, is visible from outside, but the functions coming after, the subfunctions, are only visible to the primary function or to other subfunctions in the same file. So something like this:

```
function w = f(x, y, z)
 w = x + g(z);
 ...

function s = g(t)
  ...
  s = ...
```


Another alternative is to use nested functions,

```
function w = f(x, y, z)
 w = x + g(z);
 ...

  function s = g(t)
    s = ...
    ...
  end  % necessary

end  % necessary
```

Read more in the manual about scope for variables and functions.
A function can take zero or more input arguments and return zero or more output arguments.

```
function [b_plus_c, sum_A] = func(A, b, c)

  b_plus_c = b + c;
  sum_A    = sum(A(:));


>> F = [1 2;3 4]
F =
     1     2
     3     4
>> h = [1 3]; g = [2 5];

>> [uu, vv] = func(F, h, g)
uu =
     3     8
vv =
    10

>> z = func(F, h, g)
z =
     3     8
```

It is possible to, inside the function, see the number of arguments.

```
  function [out1, out2, out3] = func(in1, in2, in3, in4)
  n_in_arg = nargin;
  n_out_arg = nargout;

  if n_in_arg == 4
    ...
  elseif n_in_arg == 3
   ...
etc.
```

It is possible to have optional input (output) parameters, so the number of parameters of a function may change between calls. See the documentation for **varargin** and **varargout** for details.

Global variables

Variables in functions are local to the function. We use the parameters to communicate with other routines. Another way is to use global variables.

```
>> global a b   % In Matlab, or the calling routine
>> type func

function func

global a b      % A matching global declaration

a = a + 1;
b = b * 10;

>> a = 1; b = 2;
>> func
>> a

a =
     2

>> b

b =
    20
```


Persistent variables

A variable which is local to a function does not keep its value between calls. To make it keep the value, we use a persistent declaration. A persistent variable is initialised to the empty matrix.

```
>> type pers
function num_calls = pers
  persistent k  % persistent num_calls does not work

  if isempty(k)
    k = 0;
  end

  k = k + 1;

  num_calls = k;

>> pers
ans =
     1
>> pers
ans =
     2
>> pers
ans =
     3


>> clear pers
>> pers
ans =
     1
```


A few tips

Debugging: there is a Matlab-debugger, but it is usually sufficient to remove semi-colons (to print variables).
The **keyboard**-command is convenient when we want to stop in functions. Resume execution by typing the letters **return** .

```
>> y = cos(0)
y =
     1
>> cos = 8
cos =
     8
>> y = cos(0)
??? Subscript indices must either be real
    positive integers or logicals.

>> which cos
cos is a variable.  % checks variable first
                    % then function
>> clear cos        % remove definition

>> which cos
built-in (/chalmers/sw/ ... /cos)  % double method

% Even more amusing
>> cos = 1:4
cos =
     1     2     3     4

>> cos(1)
ans =
     1
```

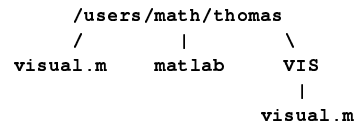
The **clear**-command takes several parameters. Here are a few. For a full description, see the documentation.

clear removes all variables from the workspace.
clear variables does the same thing.
clear global removes all global variables.
clear functions removes all compiled M- and MEX-functions.
clear all removes all variables, globals, functions and MEX links.
clear var1 var2 ... clears the variables specified.
clear fun clears the function specified.
Clear does not affect the amount of memory allocated to the Matlab process under unix.

Some commands have been written in C while others are m-files,

```
>> type cos
cos is a built-in function.

>> which ls
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/general/ls.m

>> type ls   % lists the m-file (not included)
>> dir       % (DOS-command) faster
```

More unix-like stuff. **cd**, **path** etc. **matlab** and **VIS** are directories.




```
>> cd ~             % ~ home dir
>> cd               % print current directory
/users/math/thomas

>> pwd              % an alternative
ans =
/users/math/thomas

>> which visual              % one visual.m here
/users/math/thomas/visual.m

>> cd matlab
>> pwd
ans =
/users/math/thomas/matlab
```



```
>> which visual              % but no one here
visual not found.

>> cd ../VIS

>> pwd
ans =
/users/math/thomas/VIS

>> which visual
/users/math/thomas/VIS/visual.m  % and one here

>> cd ../matlab
>> which visual
visual not found.

>> path(path, '../VIS')
>> which visual
../VIS/visual.

>> path    % lists the path

        MATLABPATH

/users/math/thomas/matlab
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/general
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/ops
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/lang
/chalmers/sw/sup/matlab-7.1/toolbox/matlab/elmat

etc.
```


Handling files

save filename saves all workspace variables to a binary file **filename.mat**. The data may be retrieved with **load**.

If **filename** has no extension, **.mat** is assumed. **save**, by itself, creates **matlab.mat**.

save filename var saves only **var**.
save filename var1 var2 var3 saves only **var1**, **var2** and **var3**.

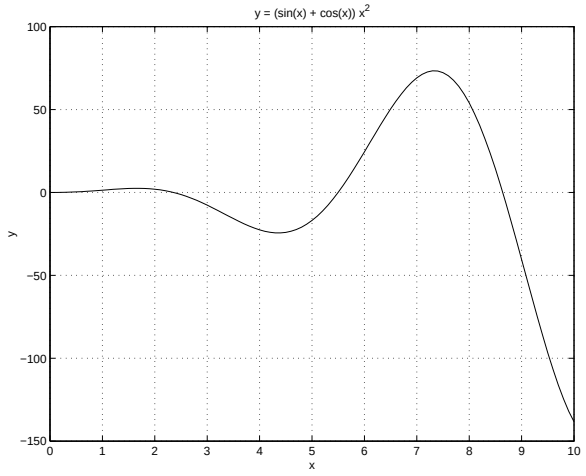
save filename var -ascii or **save -ascii filename var** saves in human readable form, 8-digit ASCII.
save -ascii -double filename var saves in 16-digit ASCII.

If one needs more control over the format, **fprintf** can be used. This routine accepts the same type of formatting codes as C. **fscanf** is a more general routine for reading data. **help iofun** gives a long list of I/O-related routines.

Graphics in Matlab

I assume you have seen some basic plotting, but here comes a few simple examples. First 2D-plots:

```
>> x = 0:0.1:10; % or linspace
>> plot(x, (sin(x) + cos(x)) .* x.^2)
>> grid on
>> xlabel('x')
>> ylabel('y')
>> title('y = (sin(x) + cos(x)) x^2')
```

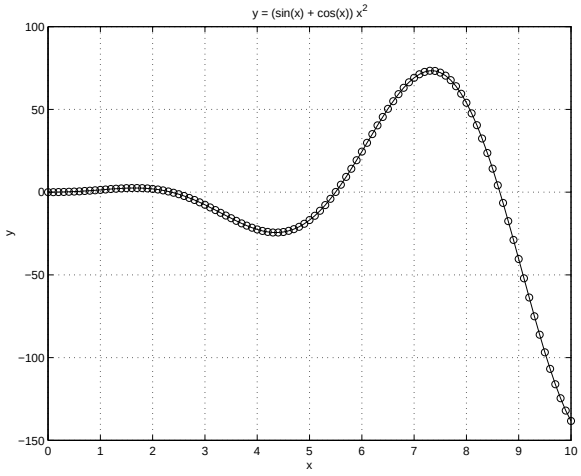


I usually make the lines thicker, increase the size of numbers and letters when showing transparencies, but we have not learnt that yet, and I do not want to give the wrong impression. So that is why it is hard to read the text in the plots.


```
>> hold on
>> plot(x, (sin(x) + cos(x)) .* x.^2, 'o')
>> help plot

 PLOT   Linear plot.
    PLOT(X,Y) plots vector X versus vector Y.
  ....
```

An alternative to hold on/off:

```
>> y = (sin(x) + cos(x)) .* x.^2;
>> plot(x, y, '-', x, y, 'o')
```



A good sequence is:

```
figure(1)  % create otherwise put on top
hold off
plot...
hold on
plot ...

figure(2) % makes a new figure window
```

`plot(1, 1:3, 'o')` is equivalent to `plot([1 1 1], 1:3, 'o')`.
`plot(1:3, 1, 'o')` is equivalent to `plot(1:3, [1 1 1], 'o')`.

`plot(x, y)` works as expected even if `x` is a row vector and `y` is a column vector (or vice-versa).

If `x` is a vector and `Y` is a matrix, then `plot(x, Y)` is equivalent to plotting `plot(x, Y(:, 1))`, ..., `plot(x, Y(:, end))` or `plot(x, Y(1, :))`, ..., `plot(x, Y(end, :))` whichever lines up. If the matrix is quadratic, the columns are used.

`x` cannot be a scalar.

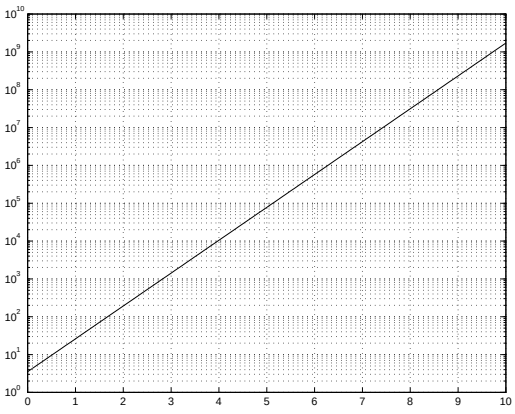
It is analogous for `plot(Y, x)`.

`plot(X, Y)` where also `X` is a matrix plots `Y(:, k)` as a function of `X(:, k)`.

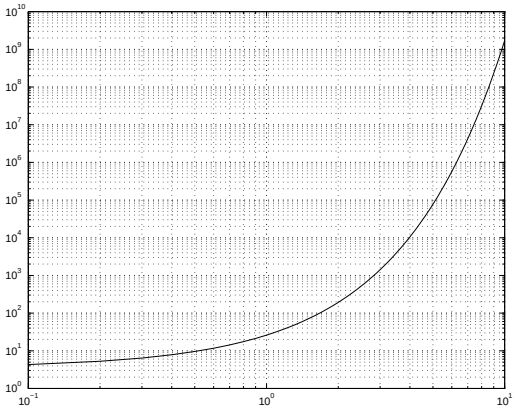
Do not forget the x-values. `plot(y)` is equivalent to `plot(1:length(y), y)`.

`plot(Y)` is equivalent to `plot(1:m, Y(:, 1))`, ..., `plot(1:m, Y(:, end))` where `m = size(Y, 1)`.


```
>> x = 0:0.1:10;
>> y = 3.52441 * exp(2 * x);
>> semilogy(x, y)
>> grid on
>> print -deps semilogy.eps

% head semilogy.eps   In unix
%!PS-Adobe-2.0 EPSF-1.2
%%Creator: MATLAB, The Mathworks, Inc.
%%Title: semilogy.eps
%%CreationDate: 09/01/ 0  22:34:29
%%DocumentNeededFonts: Helvetica
%%DocumentProcessColors: Cyan Magenta Yellow Black
%%Pages: 1
%%BoundingBox:    66   210   548   592    <--- NOTE
%%EndComments
```



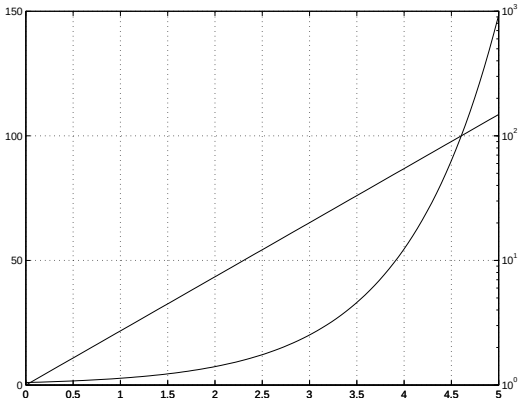
```
>> loglog(x, y)
>> grid on
```



One can plot with different styles, e.g. `plot(x, y, 'r:')`, plot with a red dotted line. Type `help plot` for details.

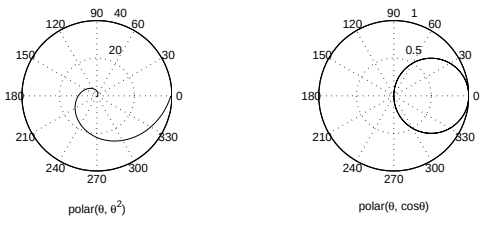
Two y-axes.

```
>> x = linspace(0, 5);
>> y = exp(x);
>> plotyy(x, y, x, y, 'plot', 'semilogy')
>> grid on
```



Polar coordinates and subplot

```
theta = linspace(0, 2 * pi);
subplot(2, 2, 1)    % 2 x 2-matrix of plots
polar(theta, theta.^2)
xlabel('polar(\theta, \theta^2)')

subplot(2, 2, 2)
polar(theta, cos(theta))
xlabel('polar(\theta, cos\theta)')

subplot(2, 2, 3)
polar(theta, cos(theta).^2)
xlabel('polar(\theta, cos^2\theta)')

subplot(2, 2, 4)
polar(theta, sin(theta).*cos(theta))
xlabel('polar(\theta, sin\theta cos\theta)')
```



Matlab understands simple TEX/LATEX-expressions such as:

```
Greek letters: \alpha, \beta, ..., \Gamma, \Xi

Index: \alpha_2^3, \alpha^{m + n}

Integrals: \int_a^b f(x) dx
```

Here is the LATEX-code:

$$\alpha, \beta, ..., \Gamma, \Xi$$
$$\alpha_2^3, \alpha^{m+n}$$
$$\int_a^b f(x)dx$$

Matlab cannot cope with more complicate expressions, such as:

```
\sum_{k=0}^{n-1}\ ax^k = a\ \frac{x^n-1}{x-1},\ x\ne 1
```

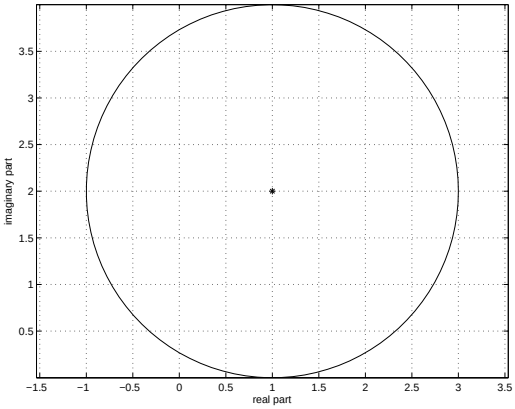
$$\sum_{k=0}^{n-1}\ ax^k = a\ \frac{x^n-1}{x-1},\ x \neq 1$$

unless ones changes the string's `Interpreter`-property to `latex` and surrounds the string with `$ $`.
It does not always seem to work properly though, I had problems with minus-signs, for example.

`text` and `gtext` can be used to place text in a plot (as can the menu in the plot window). `ginput` can be used to read the position of the mouse.

One can plot complex numbers

```
>> theta = linspace(0, 2 * pi);
>> iu = sqrt(-1)
iu =
        0+ 1.0000i
>> circle = 1 + 2 * iu + 2 * exp(iu * theta);
>> plot(circle)
>> axis equal % or axis('equal'). NOT axis square
>> grid on
>> hold
Current plot held
>> plot(1 + 2 * iu, '*')
>> xlabel('real part')
>> ylabel('imaginary part')
```

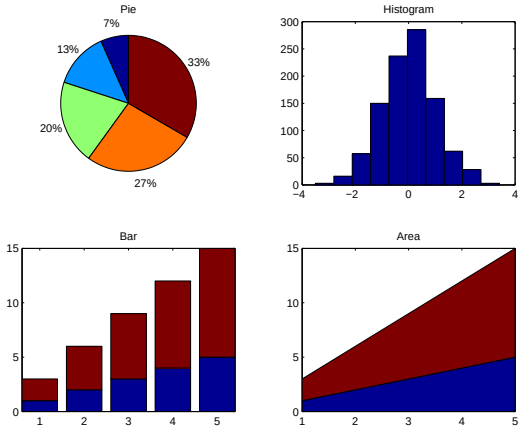


Some business graphics

Matlab can produce, bar- and area graphs. **help bar**, **help area**. There are pie charts and histograms (**help pie**, **help hist**) and a few others. Read the documentation to see the available options.

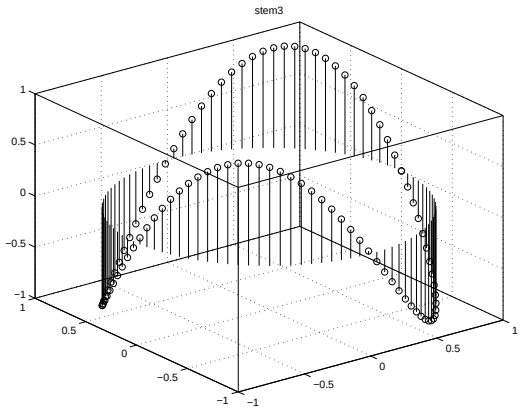
This code produces the plot on the next page:

```
figure(1)
subplot(221)
pie(1:5)
title('Pie')

subplot(222)
hist(randn(1000, 1))
title('Histogram')

subplot(223)
bar(1:5, [(1:5)', 2*(1:5)'], 'stacked')
axis tight
title('Bar')

subplot(224)
x = (1:5)';
Y = [(1:5)', 2*(1:5)'];
area(x, Y)
title('Area')
```

Here is a plot made by **stem3**:

```
>> phi = linspace(0, 2 * pi);
>> stem3(cos(phi), sin(phi), sin(2 * phi))
>> title('stem3')
```



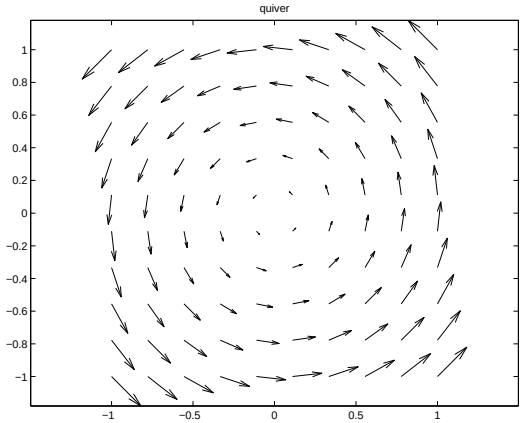
Matlab has several commands for drawing arrows, **compass**, **feather**, **quiver** and **quiver3**. These are used e.g. when drawing flow fields. Here is a **quiver**-example.

We start by creating a grid in the x-y-plane, using the **meshgrid**-command. First a word on how **meshgrid** works:

```
>> [X, Y] = meshgrid(linspace(-1, 1, 3))
X =
    -1     0     1
    -1     0     1
    -1     0     1

Y =
    -1    -1    -1
     0     0     0
     1     1     1
```

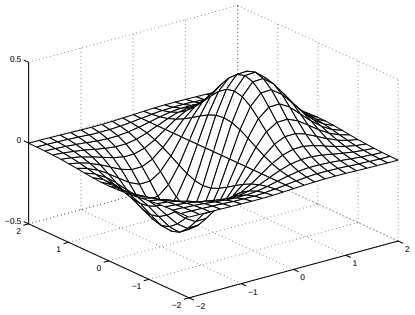
`[X, Y] = meshgrid(x_vec, y_vec);` is another alternative. In this example we draw an arrow, [u, v], that is orthogonal to the vector going from the origin to [x, y]. it should have the same length as well. So one choice is taking [u, v]=[-y, x]. here is the code:

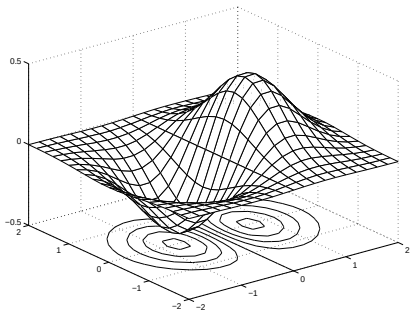
```
>> [X, Y] = meshgrid(linspace(-1, 1, 10));
>> quiver(X, Y, -Y, X)
>> axis equal
>> title('quiver')
```



The `meshgrid`-command is used when drawing simple surfaces as well, such as when we have a function $z = f(x, y)$. Here is an example.

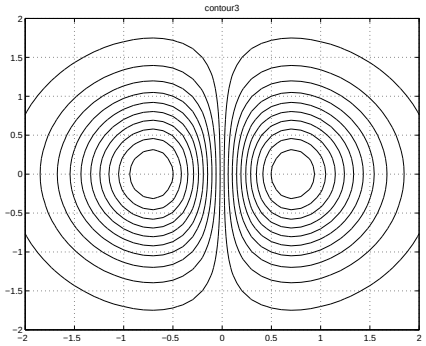
```
>> [X,Y] = meshgrid(-2:0.2:2);
>> Z = X .* exp(-X.^2 - Y.^2);  % Note elementwise
>> figure          % new plotwindow
>> mesh(X, Y, Z)
>> figure
>> meshc(X, Y, Z)  % Note the c in meshc
```



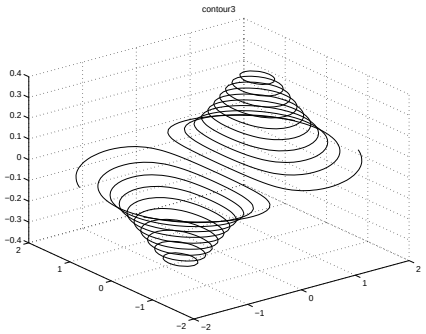


`meshc` draws the surface and contour lines, i.e. curves in the x-y-plane where $f(x, y)$ is constant. It is possible to just draw the contours using the command `contour(X, Y, Z)`. One can specify the number of contour lines or give the exact values where a contour line should be drawn. using `contour3` it is possible to put a contour line at the correct z-level.

```
>> [X,Y] = meshgrid(-2:0.1:2);
>> Z = X .* exp(-X.^2 - Y.^2);
>> contour(X, Y, Z,  20, 'k')
>> grid on
>> title('contour')
```

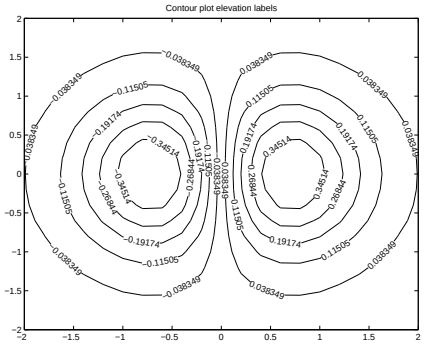



```
>> contour3(X, Y, Z,  20, 'k')
>> title('contour3')
```

One can label the contour lines

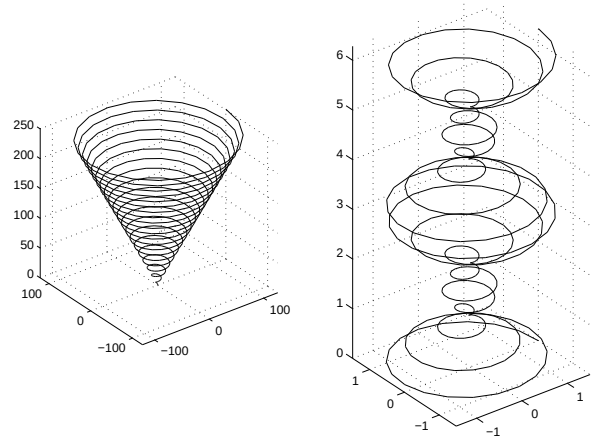
```
>> [C, h] = contour(X, Y, Z,  10, 'k');
>> clabel(C, h)
>> title('Contour plot elevation labels')
```



A rather nice contour-command is `contourf`, which fills the area between contour lines with different colours. Try it!

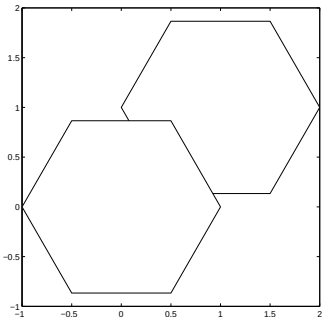
Lines in 3D:

```
>> phi = linspace(0, 40 * pi, 400);
>> subplot(121)
>> plot3(phi .* cos(phi), phi .* sin(phi), 2 * phi)
>> axis equal
>> grid on
>>
>> subplot(122)
>> r = 0.5 + cos(0.1 * phi);
>> plot3(r .* cos(phi), r .* sin(phi), phi / 20)
>> axis equal
>> grid on
```



Polygons

```
>> phi = linspace(0, 2 * pi, 7);
>> c = exp(sqrt(-1) * phi(1:end-1));  % need not close
>> x = real(c)';
>> y = imag(c)';
>> X = [1+x, x];  % one polygon per column
>> Y = [1+y, y];
>> fill(X, Y, 'w')  % not to waste tuner
>> axis([-1 2 -1 2])
>> axis square
```

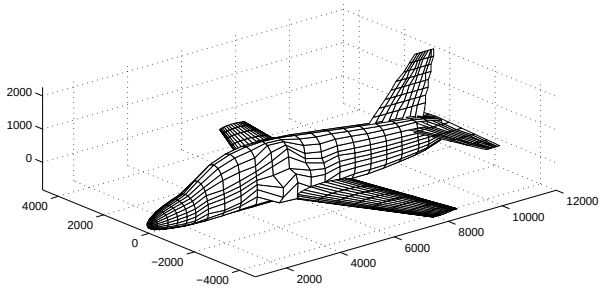


The second polygon is painted on top of the first. To see the edges we can do (much more about such things later):

```
>> h = fill(X, Y, 'w')           % a vector of handles
>> set(h, 'FaceColor','None');   % change the FaceColor-
                                 % property
```


Polygons in 3D

```
>> C = Z;
>> fill3(X, Y, Z, 'w')
>> grid on
>> axis equal
>> view([320, 20])
>> whos
  Name        Size              Bytes  Class

  C           4x1214            38848  double array
  X           4x1214            38848  double array
  Y           4x1214            38848  double array
  Z           4x1214            38848  double array

Grand total is 19424 elements using 155392 bytes
```

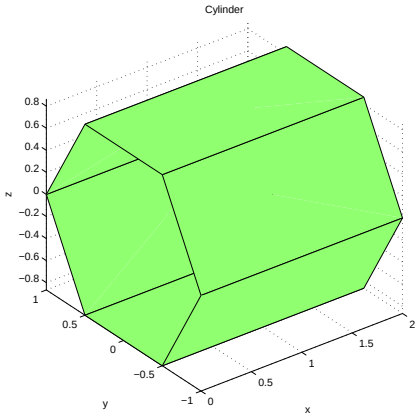


```
>> fill3(X, Y, Z, C)
```


The `surf`-command.
We have used the `mesh`-command to draw surfaces (arising from $z = f(x, y)$). When we have more general surfaces the mesh-command may not work, the surface may not be the graph of a function. The sphere is a simple example, $z = \pm\sqrt{1 - x^2 - y^2}$ does not define a function from (x, y) to z, although $z = \sqrt{1 - x^2 - y^2}$ and $z = -\sqrt{1 - x^2 - y^2}$ do. So one (perhaps not very good) way to draw a sphere is to use the mesh-command twice.

Another example is given by the following cylinder. The cylinder is centered on the x-axis and has an hexagonal cross-section.

```
>> phi = linspace(0, 2*pi, 7); % 6 corners; must close
>> surf([zeros(1,7); 2*ones(1,7)], [1;1]*cos(phi), ...
       [1;1]*sin(phi), ones(1,7)) % we could transpose
>> axis equal                    % everything
>> xlabel('x'); ylabel('y'); zlabel('z')
>> title('Cylinder')
```



To understand this better we can read the documentation. This is a quote from the manual:

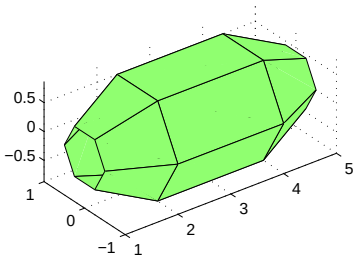
Algorithm
Abstractly, a parametric surface is parametrized by two independent variables, i and j, which vary continuously over a rectangle; for example, $1 \leq i \leq m$ and $1 \leq j \leq n$. The three functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ specify the surface. When i and j are integer values, they define a rectangular grid with integer grid points. The functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ become three $m \times n$matrices, X, Y, and Z. Surface color is a fourth function, $c(i,j)$, denoted by matrix C.
Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

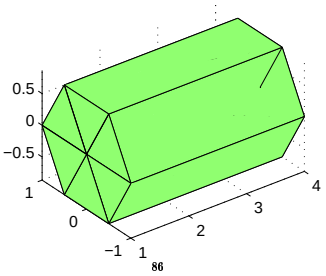
```
          i-1,j
            |
i,j-1 - i,j - i,j+1
            |
          i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, $[X(:)\ \ Y(:)\ \ Z(:)]$ returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a quadmesh.

Let us take the cylinder and close the ends. First an example where the ends are partially closed. Just to show that it is possible, we transpose all the arrays. Here

```
>> phi = linspace(0, 2*pi, 7)'; % Note transpose
>> z = zeros(7, 1); o = ones(7, 1);
>> c = cos(phi);    s = sin(phi);
>> subplot(211)
>> surf([o 2*o 4*o 5*o], [0.5*c c c 0.5*c], ...
        [0.5*s s s 0.5*s], ones(7,4))
>> axis equal
>> subplot(212)
>> surf([o o 4*o 4*o], [z c c z], [z s s z], ones(7,4))
>> axis equal
```





The line in the right part is not visible on the monitor.

It is possible to draw the cylinders using the `fill3`-command as well. That would, however, require more points.

The first cylinder we drew was defined by 14 points (two times seven edges). Using polygons we would need 24 points (6 polygons with four corners).

When we come to shading (colouring polygons with light present) we will notice a difference as well (with the normals). Six polygons are six different objects while the `surf`-cylinder is one object. We have 24 normals for the polygons and 14 for the `surf`-command.

Polygons do not have to be planar (all point in a plane). Consider the following polygon with four corners:

```
>> X = [0 1 1 0]';
>> Y = [0 0 1 1]';
>> Z = [0 0 0 1]';
>> C = ones(size(X));
>> h = fill3(X, Y, Z, C)
```

Matlab breaks up till polygon in two triangles (i.e. two planar polygons). This is a special case of tessellation:

Etymology: Late Latin tessellatus, past participle of tessellare to pave with tesserae, from Latin tessella, diminutive of tessera : to form into or adorn with mosaic

There is an image-toolbox. Here I am covering a surface with an image (usually called a texture, in this context, and the process is called texture-mapping). We will be using textures in the OpnGL-lab.

```
>> B = imread('te.jpg', 'jpg');
>> image(B)    % to look at the image
>> axis image  % correct scaling
>> [X, Y] = meshgrid(linspace(-1, 1, 10));
>> warp(X, Y, (X.^2 - Y.^2) .* cos(0.1 * Y), B)
>> axis off
```

In the upper part of the windows there are buttons for zooming, rotation etc. Have a look at the Tools- and View-menus as well. Some of the remaining buttons and menus are used for editing an image (adding text, arrows etc).

There are several other plot-commands, but before we get back to those we need to have a look at Matlab's handle graphics.