Functions for building vectors
colon(:),linspace,logspace

- `v=a:b, w=a:h:b;` default: h=1
- `v=linspace(a,b,N);` **default: N=100**
- `v=logspace(a,b,N);` $10^a, \ldots, 10^b$, **N points**

```
>> 0:10; 0:.1:1;
>> 10:-2:0
  ans =
    10     8     6     4     2     0
>> logspace(0,1,4)
   ans =
    1.0000    2.1544    4.6416   10.0000
>> 10.^linspace(0,1,4)
   ans =
    1.0000    2.1544    4.6416   10.0000
```

Note: Remember semicolon (;) for large N or small h.

Matrices: building, parts, decomposing

```
>> A=[1:3;4:6]  % Basic
```

reshape

- Forms a matrix of given size for given data.
- Data will be placed in "frame" of given size in **column order**. (Matlab is column oriented.)
- Nr. of datapoints (`numel(data)`) has to match product of dimensions.

```
>> A=reshape(1:6,2,3)  % 2x3 matrix from data 1:6 ...
    in column order
>> B=reshape(1:6,3,2)' % Row-order
>> C=reshape(A,1,6)    % Back to vector 1:6
```

Array, matrix,vector,scalar

- Basic data structurte: Matrix (array), elements: complex numbers. Let's limit ourselves at first to two-dimensional arrays.
  - Column vector: `(m,1)`-matrix
  - Row vector: `(1,n)`-matrix
  - Scalar: `(1,1)`-matrix
  - Empty: `(m,0)` or `(0,n)`-matrix
- Matrix and its (`size`)

```
>> A=[1 2 3 4 ;5 6 7 8; 9 10 11 12]
>> [m,n]=size(A)
>> v=-[1 2 3 4 ]
>> length(v)
>> 1:10       % [1,2,3,...,10]
>> size(ans) % ans:previous non-assigned result
>> who, whos % workspace variables
```

Matrices

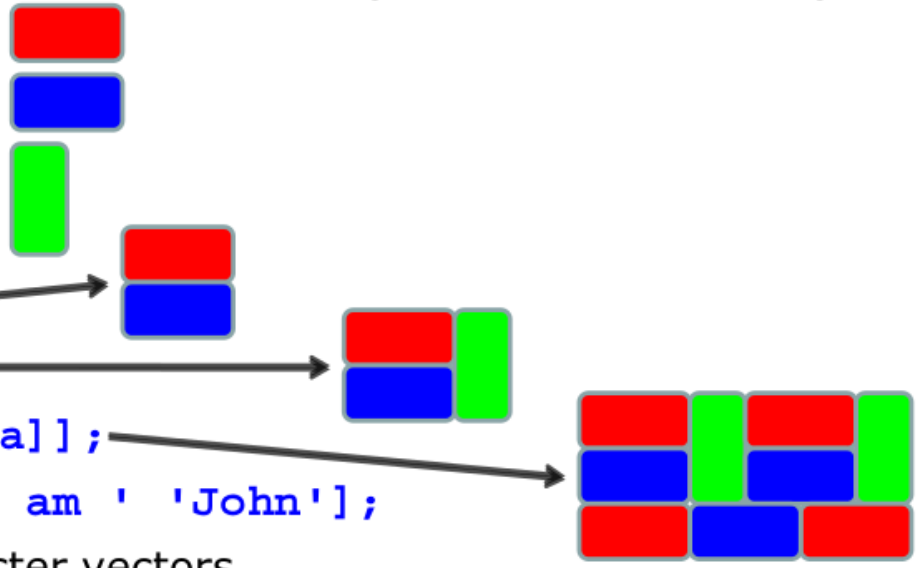
- Make matrices like vectors
- Element by element

```
» a= [1 2;3 4];
```

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- By concatenating vectors or matrices (dimension matters)

```
» a = [1 2];
» b = [3 4];
» c = [5;6];
» d = [a;b];
» e = [d c];
» f = [[e e];[a b a]];
» str = ['Hello, I am ' 'John'];
```



  - Strings are character vectors

Creating arrays

- Square brackets `[...]` to define arrays
- Spaces (and/or commas) to separate columns (elemnts of row vector).
- Semi-colons (;) to separate rows (elements of column vector)
- `>> [ 3 4 5 ; 6 7 8 ]` is a 2-by-3 matrix
- If A and B are arrays with the same number of rows, then `>> C = [ A B ]` is the array formed by stacking A and B side by side

```
 >> A=ones(2,2);B=2*ones(2,3);[A B]
ans =
     1     1     2     2     2
     1     1     2     2     2
```

Creating arrays, continued

- If A and B are arrays with the same number of columns, then `>> [ A ; B ]` is the array formed by stacking A on top of B.
- So, `[[ 3 ; 6 ] [ 4 5 ; 7 8 ] ]` is equal to `[ 3 4 5;6 7 8 ]`

Some functions for building matrices

`eye,vander,hilb,zeros,ones,diag,rand,reshape,magic`

Complete list: `help elmat`

```
>> A = zeros(2,5)
>> B = ones(3)     % or ones(3,3)
>> R = rand(3,2)
>> N = randn(3,2)
>> D = diag(-2:2)
```

Compare `rand` and `randn` Try repeatedly

`>> R = rand(3,2)` Use (↑) in command window

Repeat : `>>rand('twister',0); R = rand(3,2)`

Matrices: building, parts, decomposing

```
>> A=[1:3;4:6]  % Basic
```

reshape

- Forms a matrix of given size for given data.
- Data will be placed in "frame" of given size in **column order**. (Matlab is column oriented.)
- Nr. of datapoints (`numel(data)`) has to match product of dimensions.

```
>> A=reshape(1:6,2,3)  % 2x3 matrix from data 1:6 ...
    in column order
>> B=reshape(1:6,3,2)' % Row-order
>> C=reshape(A,1,6)    % Back to vector 1:6
```

Matrices, building blocks

```
>> A=reshape(1:6,2,3); B=ones(2,2),C=diag(1:3)
>> [A B]  % Side by side.
ans =
     1     3     5     1     1
     2     4     6     1     1
>> [A;C] % On top of each other.
ans =
     1     3     5
     2     4     6
     1     0     0
     0     2     0
     0     0     3
```

Matrix- and array algebra

`A, B` matrices, matching size, `c` scalar.

Matrix algebra

- `A + B, A+c`
- `A*B` matrix product
- `A'` (conjugate) transpose
- `A.'` transpose without conjugation
- `A^p (A*A*...A)` Matrix power (A square matrix.)
- `A\b`
  $Ax = b \iff x = A\backslash b$ (if A is invertible)

Array algebra

- `A + B, A+c`
- `A.*B` Pointwise product
- `A.^p, A.^B` Pointwise power, p scalar, A and B of same size.
- `A./B, c./A` Pointwise divide. Subtle `1.0/A, 1.0./A,1./A`
- **Note:** `c/A` usually leads to an error.

Visualization of matrices

Have fun with some commands of type:

```
>> mesh(ones(30));hold on;mesh(zeros(30));
>> mesh(eye(30));shg; hold off
>> imagesc(diag(-5:5)),colorbar;shg
>> surf(magic(10));colorbar;shg
>> surfc(vander(0:.1:1));colorbar;shg
>> imagesc(reshape(0:24,5,5)),colorbar
```

Modify some parameters, and try to see what kind of matrices the visualizations reveal to you.
In the figure-window you can click the *"rotate-arrow"* and rotate your figure with the mouse.

Accessing single element of a vector

If A is a vector, then

- A(1) is its first element
- A(2) is its second element
- ...
- A(end) is its last element

For matrices either *columnwise linear indexing*, or

- A(1,1) is the element on the first row of the first column
- A(2,1) is the element on the second row of the first column
- A(3,4) is the element on the third row and fourth column
- A(4,end) is the last element of the fourth row

Example

```
    >> A = [ 3   4.2   -7   10.1   0.4   -3.5 ];
    >> A(3)
    >> Index = 5;
    >> A(Index)
>> A(4) = log(8);
>> A
>> A(end)
```

Accessing multiple elements of an array

Index need not be a single number – you can index with a vector.

```
>> A = [ 3  4.2  -7  10.1  0.4  -3.5 ];
>> A([1 4 6]) % 1-by-3, 1st, 4th, 6th entry
>> Index = [3 2 3 5];
>> A(Index) % 1-by-4
```

Index should contain integers. Shape of the index will define the shape of the output array.

Exercise

Using MATLAB indexing, compute the perimeter sum of the matrix "`magic(8)`".

Perimeter sum adds together the elements that are in the first and last rows and columns of the matrix. Try to make your code independent of the matrix dimensions using `end`.

Linear Systems

Linear systems of equations

Given the system of equations:

$$\begin{cases} 6x + 12y + 4z = 70 \\ 7x - 2y + 3z = 5 \\ 2x + 8y - 9z = 64 \end{cases}$$

Solve it!

```
>> A=[6 12 4;7 -2 3;2 8 -9]
>> b=[70;5;64];
>> x=A\b; x'
ans =
     3     5    -2
```

Linear systems of equations, continued

```
>> [A*x b] % Check by multiplication:
ans =
    70    70
     5     5
    64    64

>> b=[70;5;64];
>> x=A\b; x'
ans =
     3     5    -2
>> x=inv(A)*b % Alternatively multiply by inverse
```

- Backslash \ is recommended for efficiency and accuracy.
- Linear systems don't always have a unique solution.
- `det(A)==0` is not a numerically reliable way of testing "almost singularity". See `help cond, rcond`.

Excercise

Solve the system of equations

$$\begin{cases} 2x + y = 3 \\ x - 2y = -1 \end{cases}$$

using the "backslash" operator, and check the result.

Using the same technique, solve below system, and check result.

$$\begin{cases} 35x_1 + 0x_2 + 14x_3 + 16x_4 + 2x_5 = 67 \\ 27x_1 + 7x_2 + 14x_3 + 4x_4 + -7x_5 = 45 \\ -13x_1 - 2x_2 + 6x_3 + 10x_4 + 8x_5 = 9 \\ 30x_1 - 1x_2 - 12x_3 + 7x_4 - 11x_5 = 13 \\ 7x_1 + 14x_2 + 7x_3 - 3x_4 - 10x_5 = 15 \end{cases}$$

Functions

User-defined functions

- Function handles, anonymous functions
  - One-liners, defined in the command window or in a script
    `>> f=@(x)x.^2` to be read: f is the function which "at x" returns the value x^2. (In math: $f = x \to x^2$)
    Several inputs allowed:
    `>> g=@(x,y,z)sqrt(x.^2+y.^2+z.^2).`
- Functions in m-files
  If more lines are needed, local variables, control structures (`for`, `while`, `if - else`, etc.), then write an m-file
- Inline-function is older, more restrictive version of function handle. We will not use them actively, the only reason to know about them, is old Matlab-codes. (`help inline`)

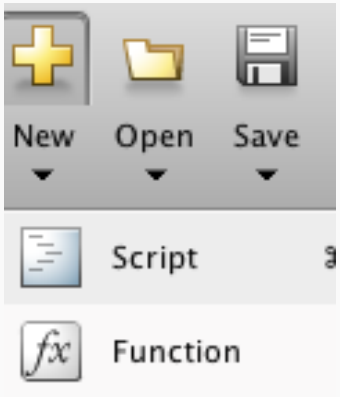
User-defined functions, m.file

function [out1,out2,out3]=funname(in1,in2)

file: funname.m on matlabpath.

- Keyword function
- Each out_k-argument must be assigned a value, the last assignement is the value returned.
- Variable scope: All variables defined in the function body are local, i.e. they are cleared when function stops running. (Note the difference with a script).
- Function needn't have output-arguments it can display text or graphics, write to files etc. In such cases it may often be more natural to use a script, though.

Examples of writing functions

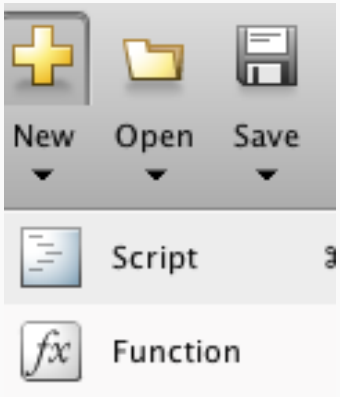


To start editing a function, open the editor on the top left "New"-button. Instead of script, this time click Function. Or on the command line:

```
>> edit myfunction
```

As our first example, let's write a function that computes the mean of the components of the input vector.
Let's first give some thought of the expression.

Examples of writing functions



To start editing a function, open the editor on the top left "New"-button. Instead of script, this time click Function. Or on the command line:

```
>> edit myfunction
```

As our first example, let's write a function that computes the mean of the components of the input vector.
Let's first give some thought of the expression.

```
x=1:10;
avg=sum(x)/length(x)
```

Example 1, mean of a vector

```
function y=mymean(x)
% Compute the mean (average) ox x-values.
% Input: vector x
% Result : mean of x
% Exampe call: r=mymean(1:10)
%
y=sum(x)/length(x);
```

```
>> help mymean
  Compute the mean (average) ox x-values.
  ...
>> r=mymean(1:10)
r =
    5.5000
```

Example 2.: function stats

Standard deviation is given by:

$$\sigma = \sqrt{\frac{1}{N}\sum_{k=1}^{n}(x_k - \mu)^2}.$$

Write the code for the following function file:

```
function [avg,sd,range] = stats(x)
% Returns the average (mean), standard deviation
% and range of input vector x
 N=length(x);
 ...
 ...
```

Calling example function *stats*

Test your function using a script like the following:

```
%% Test script for function stats
x=linspace(0,pi);
y=sin(x);
[a,s,r]=stats(y)  % Function call
plot(x,y,'b')     % 'b' for blue
hold on
plot([0 pi],[a a],'k') % 'k' for blacK
shg               % show graphics
```

Solution: Listing of function stats

```
function [avg,sd,range] = stats(x)
% Returns the average (mean), standard deviation
% and range of input vector x
 N=length(x);
 avg=sum(x)/N;
 sd = sqrt(sum(x - avg).^2)/N);
 range=[min(x),max(x)];
```

Basics of Graphics

Basic 2d-graphics, plot

- *"Matlab has excellent support for data visualization and graphics with over 70 types of plots currently available. We won't be able to go into all of them here, nor will we need to, as they all operate in very similar ways. In fact, by understanding how Matlab plotting works in general, we'll be able to see most plot types as simple variations of each other. Fundamentally, they all use the same basic constructs."*
- Links:
  - https://se.mathworks.com/help/matlab/ref/plot.html
  - http://ubcmatlabguide.github.io/html/plotting.html

Basic 2d-graphics

- If `x` is a 1-by-N (or N-by-1) vector, and `y` is a 1-by-N (or N-by-1) vector, then

  ```
  >> plot(x,y)
  ```

  creates a figure window, and plots the data points with joining line segments in the axes. The points are:

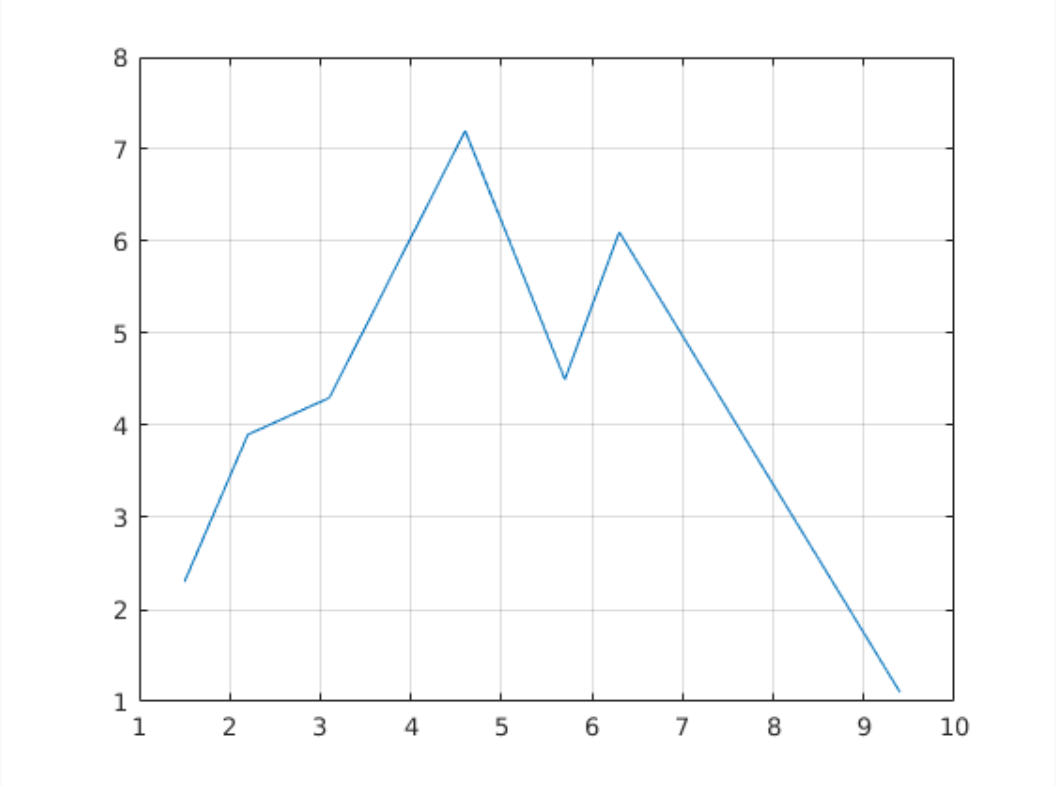
  ```
  (x(1),y(1)), (x(2),y(2)),..., (x(N),y(N))
  ```

- The axes are automatically chosen so that all data just fits into the figure window. This can be changed by the `axis, xlim, ylim`-commands.

Basic 2d-graphics, plot

Function *plot* can be used for simple "join-the-dots" xy-plots.

```
>> x=[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y=[2.3 3.9 4.3 7.2 4.5 6.1 1.1];
>> plot(x,y);grid on
```



Basic 2d-graphics, general form

Continue keeping the previous plot:

```
>> hold on          % Keep the previous lines.
>> plot(x,y,'or')   % Mark datapoints with ...
    'o'-marker, r='red'
>> shg              % show graphics
```

- General form:

  `plot(x1,y1,'string1',x2,y2,'string2', ...)`

  The `'string'`-parts may be missing.

- `plot(x,y,'r*--')`

  Use red *-markers, join with red dashed line segments.

help plot -> table of markers

Various line types, plot symbols and colors: plot(X,Y,S)

S is a character string made from one element from any or all of the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p,h	pentagram, hexagram		

Plotting graphs of functions

Just take enough points to get smoothness.

```
>> x=linspace(0,3*pi);  % Default: 100 points
>> y=sqrt(x).*sin(x);   % Note again: (.*)
>> plot(x,y)
>> figure  % Open a new graphics window.
>> x1=linspace(0,pi,1000);  % More points.
>> y1=cos(4*x1).*sin(x1);
>> m=mean(y1);
>> plot(x1,y1,[0 pi],[m m],'r--') % "red" dashed
>> legend('Function','mean'); grid on
```

Refrences in Finnish:

http://math.aalto.fi/∼apiola/matlab/opas/mini/vektgraf.html

http://math.aalto.fi/∼apiola/matlab/opas/lyhyt/grafiikka.html

Excercise

Let's do some plotting. Do the following:

a) Graph the function $f(x) = sin(x)$ on the interval $x \in [0, 1]$. Try changing the plot colour, and observe your discretization by using different plotting styles.

b) Graph the function $f(x) = \frac{1}{4}x\sin(x)$ on the interval $x \in [0, 40]$ in the same plot with $y_1 = \frac{1}{4}x$ and $y_2 = -\frac{1}{4}x$. Plot the lines with red dashes, and change the line width of f to 3.

c) Plot a curve with x coordinate of $\cos(t)$ and y coordinate of $\sin(t)$ when $t \in [0, 2\pi]$.

d) Plot a parametric curve

$$\begin{cases} x = \sin(t)\left(e^{\cos(t)} - 2\cos(4t) - \sin\left(\frac{t}{12}\right)\right) \\ y = \cos(t)\left(e^{\cos(t)} - 2\cos(4t) - \sin\left(\frac{t}{12}\right)\right) \end{cases}$$

Some plots will propably not look like you expect: try using `axis equal` or `axis square`.

Polynomials

Polynomials, roots, value

Let $p = x^4 - 15x^2 + 45x - 36$. Matlab represents the polynomial as the vector of coefficients starting at the highest power:

```
>> c=[1 0 -15 45 -36]; %Note: 0 for a missing power
>> pzeros=roots(c)
pzeros =
  -5.0355 + 0.0000   % Real root
   1.8680 + 1.4184i  % complex conjugate roots
   1.8680 - 1.4184i  % (always with real polynomial)
   1.2996 + 0.0000i  % Real root
```

Note: One is tempted to use variable names such as `roots` or `zeros`. Both are names of Matlab's built-in functions (we just used `roots`). Check: `>> which roots` `>> which zeros`. Using such names may lead to "nonsense" error messages.

Polynomials, roots, value (continued)

To check how close to zero the values of the polynomial are at the computed zeros, we need the function `polyval`.
Data for plotting will also be created at once.

```
>> polyval(c,pzeros) % Values of p at pzeros
ans =
   1.0e-11 *          % Small enough
   0.1300 + 0.0000i
  -0.0043 - 0.0046i
  -0.0043 + 0.0046i
   0.0000 + 0.0000i
>> x=linspace(-6,6); % 100 equally spaced points on ...
    the interval [-6,6].
>> y=polyval(c,x);
>> plot(x,y)
```

Exercise

- Plot the values of the polynomial $p(x) = x^4 - 3x^3 + 8x + 2$ on the interval $x = [-3, 3]$.
- Find the roots of $p(x)$.
- Find the roots of $z^{12} - 1$ (Yes, there is more than one), and plot them on the complex plane.
- Construct a polynomial of degree 6, with roots $r_k = k$. (i.e., first root is 1, second 2 and so on). How high can you increase the degree, before the root-finding becomes inaccurate?