Unit 2

Matrices and Advanced Plotting/Scripting

MATLAB *matrices*

- every variable in MATLAB is a matrix

```
>> a=1                        1x1 matrix
>> b=[1, 2]                   1x2 matrix
>> c=[1; 2]                   2x1 matrix
>> d=[1, 2; 3, 4]             2x2 matrix
>> A=[10, -3, 7; 2, 12, 0]    2x3 matrix
```

`A(n,m)` n=rown, m=column

$$\mathtt{A} = \begin{pmatrix} \mathtt{A}(1,1) & \mathtt{A}(1,2) & \mathtt{A}(1,3) \\ \mathtt{A}(2,1) & \mathtt{A}(2,2) & \mathtt{A}(2,3) \end{pmatrix}$$

➢ **exercise:**

- define various matrices (1x1, 1x3, 3x1, 3x3) and check the results of…

```
>> transpose(d)
>>d’
>> size(d)
>> diag(d)
```

- Note that `diag()` can extract and generate diagonal matrix elements!
- use `help` to learn about the commands `diag()`, `zeros()`, `eye()`, `ones()`, `numel()`

- matrix multiplication:

```
• A(n,m)  * B(m,k) = C(n,k)        mathematical
• A(n,m) .* B(n,m) = C(n,m)        component-wise
```

➢ **exercise:**

- perform the following operations on `A=[1,2;3,4]` and `B=[5,6;7,8]`

```
>> A+B
>> A-B
>> A*B
>> A.*B
>> A./B
```

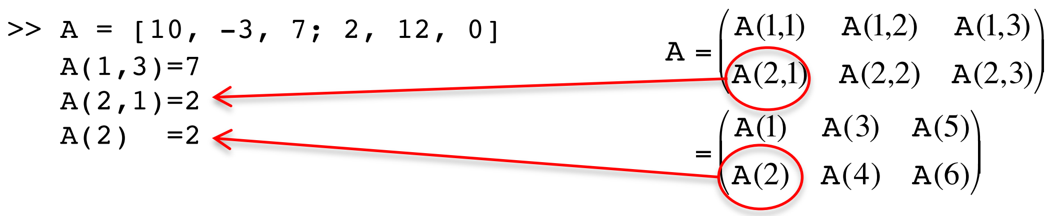
- **Note**: the operations `A/B` and `A\B` will be explained later!

➢ **exercise:**

- use `A` and `B` from the previous exercise to generate the following matrix `C` with one command

```
>> C = ???
C =
       1  2  0  0
       3  4  0  0
       0  0  5  6
       0  0  7  8
```

- matrix elements can be accessed individually



```
>> A = [10, -3, 7; 2, 12, 0]
   A(1,3)=7
   A(2,1)=2
   A(2)  =2
```

$$A = \begin{pmatrix} A(1,1) & A(1,2) & A(1,3) \\ A(2,1) & A(2,2) & A(2,3) \end{pmatrix} = \begin{pmatrix} A(1) & A(3) & A(5) \\ A(2) & A(4) & A(6) \end{pmatrix}$$

➢ **exercise:**

- extract the second column of `A` into a vector `c` with one command
- extract the second row of `A` into a vector `r` with one command

- **Note**: do *not* write [A(1,2),A(2,2)] or [A(2,1),A(2,2),A(2,3)] but use the colon operator ':' instead

- useful function `find()`:

```
>> a = [0.1, 7.3, 0.5, 3.2, 2.8, 6.9]
>> find(a>3.5)
>> find(a<3.5)
```

(more about conditions like "<" and ">" later on page 16)

`>> use help find` to learn more about `find()` and its mode of operation!

➢ **exercise:**

- fill all zero elements of the following matrix `A` with `-1`

```
>> a = [1, 2, 3, 4, 5, 6]
>> A = diag(a)

>> ???
 A =
      1  -1  -1  -1  -1  -1
     -1   2  -1  -1  -1  -1
     -1  -1   3  -1  -1  -1
     -1  -1  -1   4  -1  -1
     -1  -1  -1  -1   5  -1
     -1  -1  -1  -1  -1   6
```

hint: you have to use `find()`

- just like with vectors, you can easily remove columns and/or rows from a matrix, e.g.

```
>> A(:,1) = []
>> A(end,:) = []
```

➢ **exercise:**

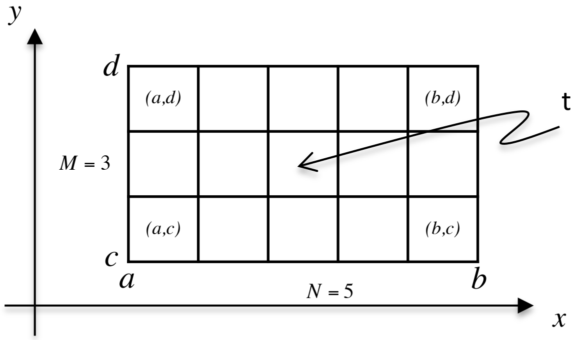
- adjust the script for the cannonball trajectory to plot the ascending trajectory in blue (vy>0) and the descending in red (vy<0)
- ***hint***: you have to use `find()` again…

MATLAB *plotting scalar fields*

- we intend to visualize a function of multiple variables, e.g.

$$f(x,y) = x^2 + y^2 \quad \text{with} \quad \begin{matrix} x \in [a,b] \\ y \in [c,d] \end{matrix}$$

- we need to cover the following area in the xy-plane:



this memory element has index (`i,j`), but needs to give **both** x and y

- we need to generate ***two*** matrices of dimension `MxN`:

```
>> xm = linspace(a,b,N)
>> ym = linspace(c,d,M)
>> [x,y] = meshgrid(xm,ym)
```

where now index (`i,j`) will give the corresponding x and y values:

`x =`

a	…	…	…	b
a	…	…	…	b
a	…	…	…	b

`y =`

c	c	c	c	c
…	…	…	…	…
d	d	d	d	d

- the 2D mesh covered by `x` and `y` can then be used to calculate $f(x,y)$, i.e. generate another `MxN` matrix that contains the function values

```
>> f = x.^2+y.^2
```

`f =`

f(a,c)	…	…	…	f(b,c)
…	…	…	…	…
f(a,d)	…	…	…	f(b,d)

- the matrix `f()` can then be visualized using one of the following MATLAB functions:

```
>> contour(x,y,f)
>> mesh(x,y,f)
>> surf(x,y,f)
>> surfc(x,y,f)
>> surfl(x,y,f)
```

MATLAB *plotting scalar fields*

- **exercise:**

- write a script **x2+y2.m** that plots $f(x,y)=x^2+y^2$ within the range [-100,100]x[-100,100]
- use `subplot()` or `figure()` to view all possible contours and surfaces simultaneously
- use `colorbar`, `axis`, and `shading` to modify the figure
- use `help` to find out more about `mesh()`, `waterfall()`, `surf()`, `surfc()`, `surfl()`
- use `help` to learn more about `colorbar`, `axis`, `shading`

- **exercise:**

- write a script **sinxcosx.m** that visualizes $f(x,y)=\sin(x)\cos(y)$ within the range [0,2π]x[0,2π]

- **exercise:**

- write a script **potential2D.m** that visualizes the potential of an electric charge:
  - place the charge at position (x_0,y_0) within the range [-1.25,+1.00]x[-0.75,+1.15]
  - generate a 2D mesh covering this x-y range using `meshgrid()`
  - use the following formula for the potential where e = -1 is the charge:

$$U = \frac{e}{\sqrt{(x-x_0)^2+(y-y_0)^2}}$$

  - visualize the potential using `contour()`, `mesh()`, `surf()`, etc. either in multiple figures (`figure()`) or in one figure (`subplot()`)

- **exercise:**

- write a new script **potentials2D.m** where you add a second charge e = +1 at position ($-x_0$, $-y_0$)
- Note: the potential is additive, i.e. $U_{total} = U_- + U_+$

MATLAB *plotting vector fields*

- MATLAB can attach vectors to (a grid of) points with `quiver(x,y,Vx,Vy)`:

```
>> quiver(x,y, Vx,Vy)
```

- example script **vectorfield2D.m**:

```
%================================
% vectorfield.m: 2D random vector field
%================================

% range and # of points
Nmesh = 10;

xmin  = -2.6;
xmax  = +3.2;
ymin  = -1.6;
ymax  = +2.8;

% generate a linearly spaced mesh in x and y
xmesh = linspace(xmin,xmax,Nmesh);
ymesh = linspace(ymin,ymax,Nmesh);
[x,y] = meshgrid(xmesh,ymesh);

% generate a 2D random vector field
vx = rand(Nmesh,Nmesh)-0.5;
vy = rand(Nmesh,Nmesh)-0.5;

quiver(x,y,vx,vy)
axis image
```

➢ **exercise:**

- generate the example script **vectorfield2D.m** as given above
- what is the output of `rand()` and `rand()-0.5`, respectively?
- how can you change the length of the vectors?
- ***hint***: `help quiver`

➢ **exercise:**

- write a script **vectorfield3D.m** that plots a random vector field in 3D using `quiver3()`

voluntary exercise!

MATLAB *gradients*

- recall the exercises on numerical derivatives, e.g. the calculation of dx and df for df/dx

```
>> il = [1:N-1], ir = [2:N];
>> dx = x(ir)-x(il);
>> df = f(ir)-f(il);
```

- as this is a rather important operation MATLAB has a simple command for this that does not need the index vectors:

```
>> dx = diff(x);
>> df = diff(f);
```

➢ **exercise:**

- adjust your **derivation.m** scipt to now use `diff()` instead of index vectors

- for functions of multiple variables we have derivatives with respect to every variable, e.g. the force is the gradient of the potential:

$$\vec{F} = -\vec{\nabla}U = -\left(\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}, \frac{\partial U}{\partial z}\right)$$

- MATLAB can calculate the gradient of a given scalar field

```
>> [Fx, Fy]     = gradient(U)     2D gradient
>> [Fx, Fy, Fz] = gradient(U)     3D gradient
```

➢ **exercise:**

- write a new script **force2D.m** by adjusting **potential2D.m** that now plots the force field.

➢ **exercise:**

- write a script **force3D.m** that now plots the 3D force field of the same electric charge

- ***hint***: you now need to add a 3rd dimension (i.e. z) to all calculations including `meshgrid()`, the actual potential and the gradient.

voluntary exercise!

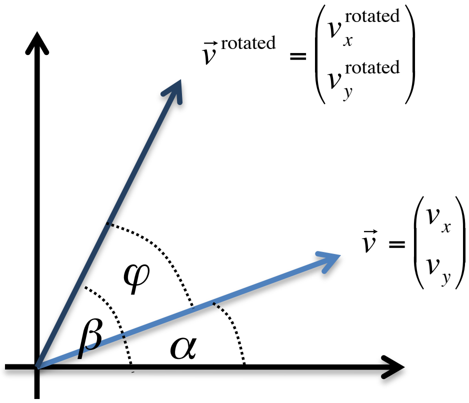
MATLAB

rotation via matrices

- the rotation of a 2D vector can be described by a matrix operation

$$\vec{v}^{\,\text{rotated}} = \hat{M}\,\vec{v}$$

- the matrix M is determined as follows:



$$v_x = v\cos\alpha \quad ; \quad v_x^{\text{rotated}} = v\cos\beta \quad ; \quad \beta = \alpha + \varphi$$
$$v_y = v\sin\alpha \quad ; \quad v_y^{\text{rotated}} = v\sin\beta$$

$$\Rightarrow \quad \begin{aligned} v_x^{\text{rotated}} &= v\cos(\alpha+\varphi) = v\left[\cos\alpha\cos\varphi - \sin\alpha\sin\varphi\right] = v_x\cos\varphi - v_y\sin\varphi \\ v_y^{\text{rotated}} &= v\sin(\alpha+\varphi) = v\left[\sin\alpha\cos\varphi + \cos\alpha\sin\varphi\right] = v_y\cos\varphi + v_x\sin\varphi \end{aligned} \quad \Rightarrow \quad \hat{M} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix}$$

➢ **exercise:**

- write a script that rotates a given 2D vector about a pre-defined angle (given in degrees!) using the rotation matrix
- proof that the original and rotated vectors have the same norm.
- graphically display the two vectors using the MATLAB function `quiver()`

➢ **exercise:**

- write a script that rotates $x = \sin(t)$ (for $t \in [0, 2\pi]$) about 32°
- ***hints***:
  - put the vectors `t()` and `x()` into a matrix `S=[t; x]`
  - rotate that matrix via `R*S` where `R` is the rotation matrix
  - extract the new vectors `t()` and `x()` from the rotated matrix and plot them

- the rotation of a 3D vector can be described by successive matrix operations

<u>rotation about x-axis</u>

$$M_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix}$$

<u>rotation about y-axis</u>

$$M_y = \begin{pmatrix} \cos\varphi & 0 & -\sin\varphi \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{pmatrix}$$

<u>rotation about z-axis</u>

$$M_z = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

➢ **exercise:**

- write a script that rotates a given 3D vector about two pre-defined angles (given in degrees!), i.e. one rotation about the x-axis and another rotation about the z-axis.
- show that rotations are non-permutative, i.e. first rotating about the x- and then the z-axis is not the same as first rotating about the z- and then the x-axis.
- proof that all (rotated) vectors have the same norm.
- graphically display all vectors using the MATLAB function `quiver3()`

MATLAB *functions*

- MATLAB comes with a suite of pre-defined and ready to use functions

```
>> sin(), cos(), exp(), log(), plot(), linspace(), meshgrid(), …
```

- we can also define our own functions, e.g. generate a file **statistic.m**

```
%==============================================================
% statistic(x): calculate median, mean and standard deviation of all elements in x
%==============================================================
function [med, avg, stddev] = statistic(x)
% Calculate the median, mean, and standard deviation of all elements in vector x
med     = median(x);
avg     = mean(x);
stddev  = std(x);
```

- to use the function we need to write a script **use-statistic.m** that, for instance, generates a vector filled with random numbers and calculates the median, mean and standard deviation of the elements of that vector by calling the function `statistic()`

```
%======================================================================
% use-statistic.m: calculate the median, mean and standard deviation of random numbers
%======================================================================
% generate a vector h filled with 1000 random numbers
h = 100*rand(1000,1);

% call our own function statistic()
[a, b, c] = statistic(h);

% print the result
a, b, c
```

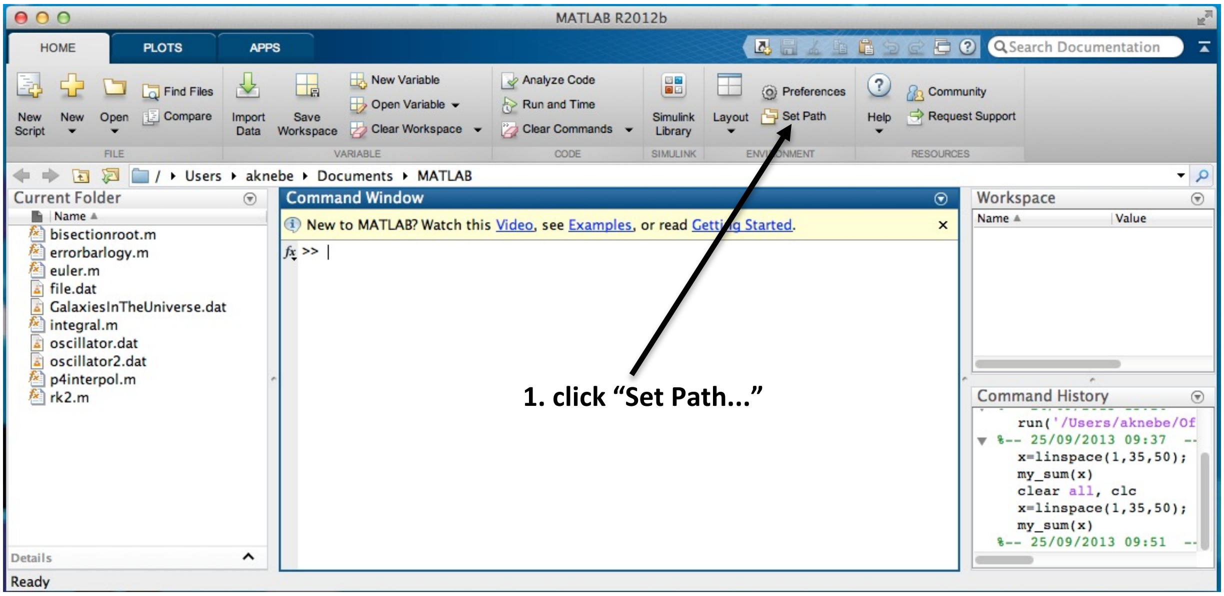
- **Note**:
  - avoid using names that already exist in MATLAB
  - functions can return single or multiple variables (or even no variable at all)
    - [a] = your_function(x) => returns a single variable "a"
    - [a,b] = your_function(x) => returns two variables "a" and "b"
  - functions can depend on a single or multiple variables
    - [a] = your_function(x,y,z) => makes use of x, y and z (but only returns "a"!)
  - a,b,x,y,z can be variables, but also vectors or multi-dimensional matrices
  - the return value(s) must be assigned in the function
  - the names of the variables inside your function do not need to be the same as the names of the variables you pass to the function!
  - if you modify "x" in your_function() this will not be known by the program calling your_function()

➢ **exercise:**

- use both **statistic.m** and **use-statistic.m**, and understand these scripts…
- what happens if you type `help statistics` in the command window?
- ***hint***:
  - check next page to better understand how to use your own function `statistic()`
  - use `help median`, `help mean`, `help std`, `help rand`

MATLAB *functions*

- before being able to use a function you must tell MATLAB where that function can be found:

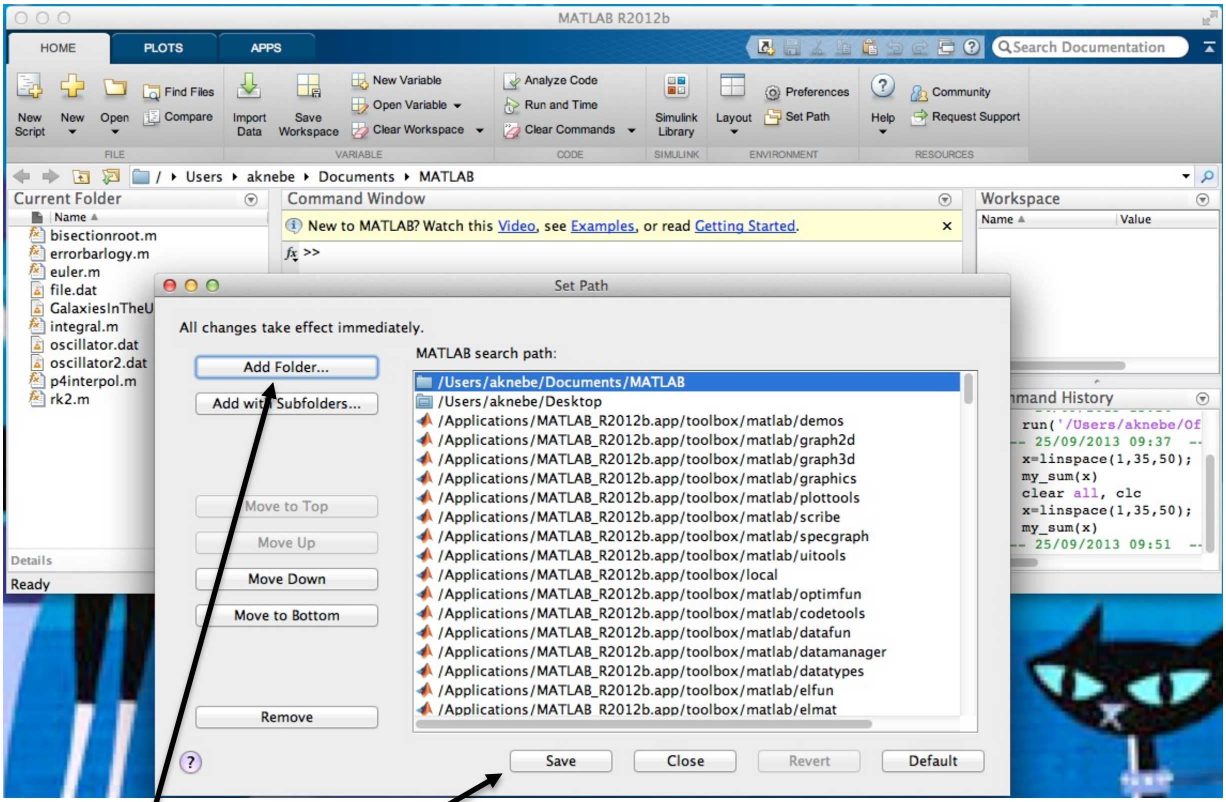


New to MATLAB? Watch this Video, see Examples, or read Getting Started.

```
fx >> |
```

```
   run('/Users/aknebe/Of
▼ %-- 25/09/2013 09:37 --
   x=linspace(1,35,50);
   my_sum(x)
   clear all, clc
   x=linspace(1,35,50);
   my_sum(x)
  %-- 25/09/2013 09:51 --
```

1. click “Set Path...”



All changes take effect immediately.

MATLAB search path:

```
/Users/aknebe/Documents/MATLAB
/Users/aknebe/Desktop
/Applications/MATLAB_R2012b.app/toolbox/matlab/demos
/Applications/MATLAB_R2012b.app/toolbox/matlab/graph2d
/Applications/MATLAB_R2012b.app/toolbox/matlab/graph3d
/Applications/MATLAB_R2012b.app/toolbox/matlab/graphics
/Applications/MATLAB_R2012b.app/toolbox/matlab/plottools
/Applications/MATLAB_R2012b.app/toolbox/matlab/scribe
/Applications/MATLAB_R2012b.app/toolbox/matlab/specgraph
/Applications/MATLAB_R2012b.app/toolbox/matlab/uitools
/Applications/MATLAB_R2012b.app/toolbox/local
/Applications/MATLAB_R2012b.app/toolbox/matlab/optimfun
/Applications/MATLAB_R2012b.app/toolbox/matlab/codetools
/Applications/MATLAB_R2012b.app/toolbox/matlab/datafun
/Applications/MATLAB_R2012b.app/toolbox/matlab/datamanager
/Applications/MATLAB_R2012b.app/toolbox/matlab/datatypes
/Applications/MATLAB_R2012b.app/toolbox/matlab/elfun
/Applications/MATLAB_R2012b.app/toolbox/matlab/elmat
```

2. “Add Folder...” and select the folder where your your_function.m files is located

3. “Save” to save the changes you just made

- **exercise:**

- write a script **use-oplot.m** that calls your own function `oplot(x,y)` defined in **oplot.m**
- your function oplot() is supposed to “overplot” some data (x,y) in an existing plot, e.g.

```
%==================================================================
% use-oplot.m: plot two functions in the same figure using oplot()
%==================================================================
x = linspace(0,2*pi,100);
figure(1)
plot(x,sin(x))      % use MATLAB’s built-in function plot() to initiate the plot
oplot(x,cos(x))     % use your own function oplot() to add another curve to the plot
```

- ***hints***:
  - the function should look like this:

```
function [] = oplot(a,b)
    % ensure that we can add a new plot to the existing figure
    command?
    % plot a on the x-axis and b on the y-axis
    command?
    % return to the situation where plot() does not add to the existing figure
    command?
end
```

  - remember `hold on` and `hold off`
  - the function `oplot()` does not return anything!

- **exercise:**

- write a script **use-ang2rad.m** that calls your own function `ang2rad(x)` defined in **ang2rad.m** converting degrees to radians, i.e.

```
function [y] = ang2rad(x)
    % command to convert x in degrees to y in radian
    command?
end
```

- use this function to plot a full period of sin(x)

- ***Note***: this function already exists in MATLAB, but when you write your own version that will be the one used by MATLAB!

- **exercise:**

- write a new script **force2D-dist2D.m** by adjusting your script **force2D.m** to now utilize a function `dist2D()` that calculates

$$dist2D = \sqrt{(x-x_0)^2 + (y-y_0)^2}$$

- ***hint***: `dist2D()` has to accept 4 arguments (x,y,x0,y0) and return 1 result (the distance)

- **there are two different types of functions in MATLAB:**
  - script functions

```
function [I] = integrate(g, x0, xend, N)
```

  - anonymous functions

```
g = @(x)(x.^2-exp(-x))
```

1) script functions:

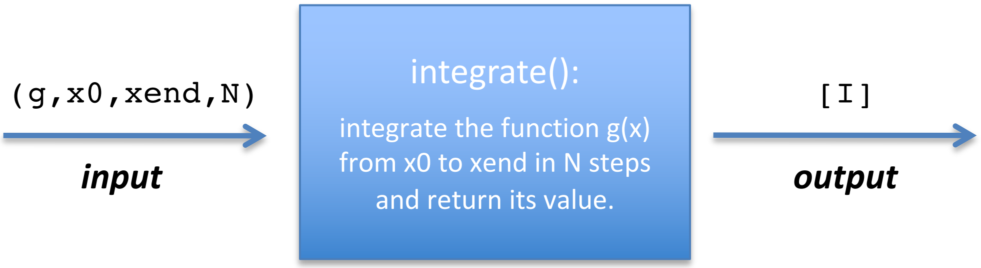
- script functions require you to write an m-file with the same name as the function
- script functions can return multiple values of different types, e.g.

```
function [E,V] = ElectricFields(r)
```

where

`E` is a 3-component vector (electric field),
`V` is a 1-component scalar (potential field), and
`r` the 3-component vector (3D position of electric charge)

- all variables declared as return values must be set inside the function
- a script function can be a block of certain operations that you plan to do repeatedly, e.g.



`(g,x0,xend,N)`

input

integrate():

integrate the function g(x) from x0 to xend in N steps and return its value.

`[I]`

output

- **there are two different types of functions in MATLAB:**
  - script functions

```
function [I] = integrate(g, x0, xend, N)
```

  - anonymous functions

```
g = @(x)(x.^2-exp(-x))
```

2) anonymous functions:

- anonymous functions can be defined anywhere in a script
- an anonymous function can be passed to a script function (see example above)
- an anonymous function rather defines a mathematical function than a block of operations

- **Note:**
  - you can pass more arguments to a function than actually used, e.g.

```
g = @(x,v,t)(-1/x.^2)
```

➔ *this can be very helpful to know when programming general purpose routines!* 🡐

- but when using `g(x,v,t)` you **must** call it with all arguments, e.g.

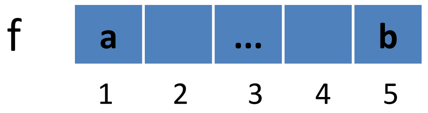
```
x = linspace(5,10,100);
plot(x, g(x,v,t))
```

…even though `v` and `t` are not used in this particular case!

- **vectors vs. functions:**

```
%========================
% f as a vector
%========================
a = 1.5;
b = 7.8;
n = 5;

x = linspace(a,b,n);

f = x.^2+5.*x;

plot(x,f)
```

```
%========================
% f as a function
%========================
a = 1.5;
b = 7.8;
n = 20;

x = linspace(a,b,n);

f = @(x)(x.^2+5.*x);

plot(x,f(x))
```



f(x)

f is a vector whose values f(1)=a, ..., f(n)=b can be read and used (and even over-written). Note again, a vector can only be accessed at the integer values i=1,...,n as they indicate the position in the vector (=vector index).

f(x) is an anonymous function that can be used to evaluate f at any given value for x. Note, the “plot(x,f(x))” command also generates a vector that contains f(x) at n points, but this vector will not be stored under any name in the computer’s memory; it will only be plotted.

- **Note:**
  - MATLAB does ***not*** distinguish syntax-wise between accessing a vector and evaluating a function
  - both commands are written as f():
    - if f is a vector, f(i) accesses element i in f()
    - if f is a function, f(i) evaluates f() at the argument i
  - other programming languages (like C) use, for instance, f[] for accessing vectors and f() for evaluating functions to distinguish between these cases...

➢ **exercise:**

- return to your script **force2D-dist2D.m** and use an anonymous function for `dist2D()` now.

➢ **exercise:**

- write a script function for log3() and use it on the command line to calculate log3(108).
- write an anonymous function for log3() and use it on the command line to calculate log3(108).

MATLAB

conditions

- it is possible to compare the content of two variables, vectors, or even matrices:

• x > a	x is greater than a
• y >= z	y is greater or equal z
• q < 5.3	q is smaller than 5.3
• p <= b	p is smaller or equal b
• m == n	m is equal n
• z ~= c	z is not equal c

- the result of any comparison is either 1 (true) or 0 (false), e.g.

```
>> 5 == 3
ans = 0
>> 7 > 2
ans = 1
```

- if you compare vectors (matrices) the result will be a vector (matrix) containing the results of a component-wise comparison, e.g.

```
>> a = [1:2:10]; b = [10:-2:1];
>> a > b
ans = 0 0 0 1 1

>> A = [1,2; 3,4]; B = [1,1; 4,4];
>> A == B
ans = 1 0
      0 1
```

- logical conditions can be combined:

& :	condition #1 AND condition #2 are true
| :	condition #1 OR condition #2 is true

- example:

```
x = input(’please give a number x = ’);
if(1 < x & x < 10)
   disp(‘the number you entered lies between 1 and 10’)
if(x < 0 | x > 2^32)
   disp('very large or negative number’)
end
```

- a common application of conditions is to use them together with MATLAB’s function `find()`

```
>> x=rand(1,10);
x = 0.8147   0.9058   0.1270   0.9134   0.6324   0.0975   0.2785   0.5469   0.9575   0.9649

>> i=find(x>0.5);
i =  1    2    4    5    8    9    10
```

→ `i()` now contains all the positions of the vector `x()` whose values are larger than 0.5

- **exercise:**

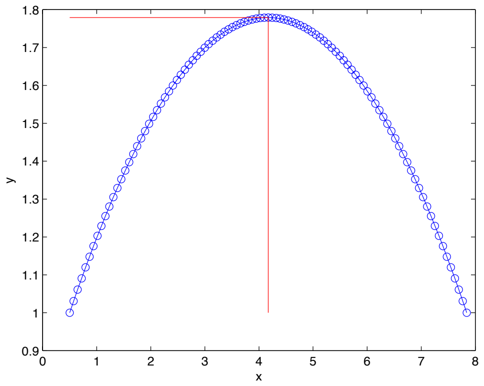
- write a script **sine-positive.m** by adjusting **sine.m** that sets all negative values of sin() to zero.

- **exercise:**

- write a script that generates a vector containing 10^6 uniformly distributed random numbers on the intervall [1,100] and calculate what fraction of numbers lies on the intervall [20,30].

- **exercise:**

- write a script **cannonball-maximum.m** by adjusting the **cannonball.m** script to also calculate the maximum height `ymax` of the cannonball
- how long does it take to reach this height (i.e. calculate the corresponding `tmax`, too)?
- at what x-position `xmax` does the cannonball reach this height?
- generate a plot that indicates the maximum by red lines on top of the actual trajectory:



- ***hints***:
  - use the following idea to finding the maximum in vector `y()`:

    for increasing values of `y()` the difference (calculated with `diff()`) between two neighbouring points in `y()` is greater than zero and less than zero for decreasing values of `y()`

MATLAB

if-else-end clause

- execute different commands depending on some (combination of) logical *condition* again, e.g.

```
%============
% if-clause
%============
if condition
    command;
end
```

```
%================
% if-else clause
%================
if condition
        command;
else
        some other command;
 end
```

```
%================
% if-elseif clause
%================
if condition
        command;
elseif condition
        some other command;
else
        another command;
end
```

- **Note**:
  - a condition used in an if(-else)-end clause should only compare scalar values and not vectors!
  - **but**: you can also compare two string variables (see exercise below)

➢ **exercise:**

- write a script function **my_abs.m** that returns the absolute value of a scalar input argument.

➢ **exercise:**

- write a function `calculation.m` that calculates either a+b, a-b, a*b or a/b depending on a variable `action` that either contains 1 (for ‘add’), 2 (for ‘subtract’), 3 (for ‘multiply’) or 4 (for ‘divide’). The function should work like this:

```
function [result] = calculation(a,b,action)
```

- **Note**: you should use a combination of `if-elseif-else-end` that also checks if the `action` is valid (i.e. valid means `action`∈`[1,4]`).

- **Note**: never compare floating variables (i.e. real numbers) using == or ~= ; use the following instead:

do not use:	`x==a`	instead use:	`abs(x-a)` < *e*	for ‘is equal’
do not use:	`x~=a`	instead use:	`abs(x-a)` > *e*	for ‘is not equal’

where *e* defines your desired accuracy, e.g.

will not work

```
if tan(0.7) == sin(0.7)/cos(0.7)
        disp(‘success’)
end
```

will work

```
if abs(tan(0.7)-sin(0.7)/cos(0.7)) < 1e-10
        disp(‘success’)
end
```

(check `help disp` to learn more about `disp()`)

MATLAB *while-loops*

- you want to repeat a certain operation *while* some logical condition remains true:

```
while condition
      command;
end
```

- example: we want to determine how often a number can be divided by 2

```
%=====================================
% simple log2() function
%=====================================
f = 32
n = 0;
while f > 1
   f = f / 2;
   n = n + 1;
end
2^n
```

- **Note**: as the title of the script suggests, this is a very simple (and crude!) form for calculating n=log2(f)
- **Note**: a condition used in while-loop should only compare scalar values and not vectors!

➢ **exercise:**

- use the above idea to write a script that evaluates log3() for several values of f (e.g. 27, 243, 531411) and compare to the real log3(f)

➢ **exercise:**

- write a script that evaluates whether or not a natural number is a prime number.

- ***hints***:
  - a prime number is a number that can only be divided by 1 and by itself
  - use `mod(n,div)` or `rem(n,div)` to evaluate the remainder of the division n/div
  - if `rem(n,div)==0` for any `1<div<n` then `n` cannot be a prime number

- ***advanced scripting hints***:
  - use `input()` to let the user input the natural `number:`

```
n = input('give a natural number n = ')
```

  - use `disp()` to print whether or not `n` is a prime number:

```
if your_condition_for_prime_number
      disp('prime number')
else
      disp('not a prime number')
end
```

MATLAB *for-loops*

- imagine you want to do same operation with every element of a vector, e.g.
  - x() and f() are vectors of the same length and you want to store in f() the numbers x^2

➔ for every i: f(i) = x(i).^2

  - MATLAB is doing this operation automatically when you type

```
>> f = x.^2
```

  - MATLAB is hiding from you a so-called *for-loop*:

```
%========================
% example for-loop
%========================
x = linspace(0,2*pi,5);

f = zeros(1, length(x));
for i=1:length(x)
   f(i) = x(i)^2;
end
```

```
%========================
% example without for-loop
%========================
x = linspace(0,2*pi,5);


f = x.^2;
```

  - **Note**:
    - f = zeros(1, length(x)) generates a vector f() with the same length as x() filling it with zeros
    - the '.' in front of '/', '*', and '^' always means that MATLAB will perform a for-loop for you

- an example where MATLAB does not provide a simplified syntax for you is:

The Fibonacci Series: $f_n = f_{n-1} + f_{n-2}$ with $f_1 = 1\ ,\ f_2 = 1$

➢ **exercise:**

- write a script **fibonacci.m** that calculates the first N Fibonacci numbers using a for-loop
- show in the same script that Binet's formula for the Fibonacci numbers is correct:

$$f_n = \frac{\varphi^n - \psi^n}{\sqrt{5}} \quad \text{with} \quad \varphi = \frac{1+\sqrt{5}}{2}\ ,\psi = \frac{1-\sqrt{5}}{2}$$

- ***Advise***:
  - you can use the loop-index to access the elements of a vector/matrix
  - you can use the loop-index as a variable in formulae…
  - …but **never** change the value of the loop-index within the loop!
  - always use integer values for the loop-index

- **exercise:**

- write a function my_sum() that calculates the sum of all elements in a vector using a for-loop: input argument: a vector x, output: the sum of all elements in x

- **exercise:**

- write a function my_find() that works like MATLAB's "find(x>0)".

- **exercise:**

- write a script **fac.m** that calculates f = n!
- remember: n! is an expression for n*(n-1)*(n-2)*(n-3)*…*2*1
- ***hints***:
    - store the result in a variable f that needs to be initialized to f=1 prior to the loop
    - you can loop from 2:n

- compare your result to MATLAB's in-built function `factorial()`

- **exercise:**

- remember MATLAB's two different (matrix) multiplication operators `*` and `.*`

```
>> A  *  B   =  C   ; mathematical multiplication
>> A .*  B   =  D   ; component-wise multiplication
```

- use for-loops instead of the operator `*` to calculate `C` for
  `A=[1,2; 3,4; 5,6]` and `B=[7,8,9; 10,11,12]`
- compare your results to the results when using the `*` operator
- **hints**:
    - the formula for a matrix multiplication is $C_{i,j} = \sum_k A_{i,k} B_{k,j}$
    - you need to use 3(!) nested for-loops
    - you need to use MATLAB's function `size()`

- use for-loops instead of the operator `.*` to calculate `D` for
  `A=[1,2; 3,4; 5,6]` and `B=[7,8; 9,10; 11,12]`
- compare your results again to the results when using the `.*` operator

▪ **Note**: both the `for`- and `while`-loop can be terminated with a `break` statement:

```
x = linspace(0,2*pi,100);
y = cos(x);                    % generated vector containing cosine curve on [0,2π]
for i=1:length(y)                         % loop through whole vector
   if(y(i)<0)                  % at first negative value...
      break;                   % ...terminate the for-loop
   end
end
plot(x(1:i-1),y(1:i-1)         % only plot the (first) positive part of the cosine-curve
```

MATLAB *switch statement*

- in case you have multiple options, there exist the *switch* statement

```
switch expression:
    case A,
          command,
          command,
          ...
    case B,
          command,
          command,
          ...
    ...
          ...
    otherwise,
          command,
          command,
          ...
end
```

➢ **exercise:**

- write a script that uses `input()` to take a number between 1 and 7 from the user
- use the switch statement to display (using `disp()`)
    - 'Monday' if that number was 1,
    - 'Tuesday' if that number was 2,
    - 'Wednesday' if that number was 3,
    - etc.
- use the 'otherwise' statement to display an error message in case the number is not in the allowed range

- the switch expression can also be a string!

➢ **exercise:**

- write a script that uses `input()` to take both a number and a string from the user:
    - `x` = variable for the number
    - `unit` = string variable for either 'meter' or 'inch'
- use a switch for `unit` to decide whether to convert `x` to meter or inch
    - case 'inch', y = x/39.3701 (converion to meter)
    - case 'meter', y = x*39.3701 (conversion to inch)
- use the 'otherwise' statement to display an error message in case `unit` does neither contain 'meter' nor 'inch'

- **exercise:**

- write a script that generates a matrix M that contains the following elements

$$M_{i,j} = \begin{cases} 0 \;, & i < j \\ \binom{i-1}{j-1} & , i \geq j \end{cases}$$

where the non-zero elements are defined as follows

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

You can use MATLAB's function `factorial()` to calculate $n!$

The matrix should have the dimensions NxN with N=15.

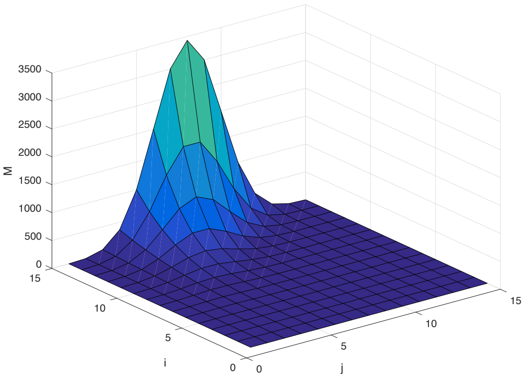
- **exercise:**

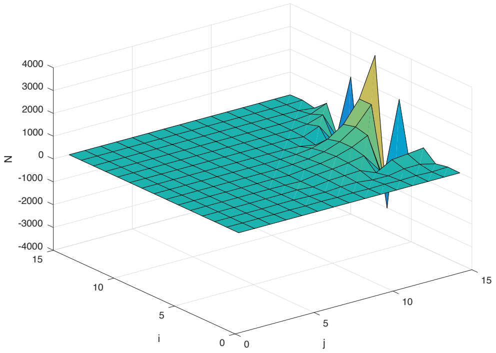
- use the matrix M to create a new matrix N with the following elements

$$N_{i,j} = \begin{cases} M_{j,i} \;, & \text{when } M_{j,i} \text{ is even} \\ -M_{j,i} \;, & \text{when } M_{j,i} \text{ is odd} \end{cases}$$

- **exercise:**

- visualize both matrices, e.g. generate plots similar to these one:





MATLAB

application – gravity

We want to determine the value of the gravitational constant g on Earth. For that we have obtained the following experimental data:

```
h = [2.0000, 4.0161, 5.5282, 6.5363, 7.0403, 7.0403, 6.5363, 5.5282, 4.0161, 2.0000]
t = [0,      0.2268, 0.4536, 0.6804, 0.9073, 1.1341, 1.3609, 1.5877, 1.8145, 2.0413]
```

where `h` measures the height above the Earth and `t` the time of the respective measurement.

➢ **exercise:**

- write a script with the name **gravity.m** in which you plot $h(t)$.

➢ **exercise:**

- write a script function `derivative(f,x)` that calculates the numerical derivative of $h(t)$. This function should work as follows:

```
function [dfdx,xmid] = derivative(f,x)
```

⟹ input values: `(f,x)`
`x` = vector containing the values of x
`f` = vector containing the values of $f(x)$ at the positions stored in `x`

⟸ return values: `[dfdx, xmid]`
`xmid` = vector containing the mid-points of `x`
`dfdx` = vector containing the numerical derivative of $f(x)$ at the mid-points

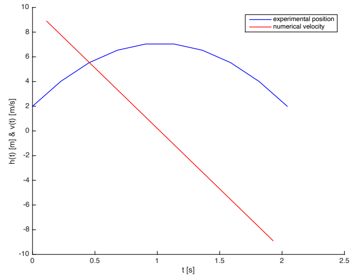
➢ **exercise:**

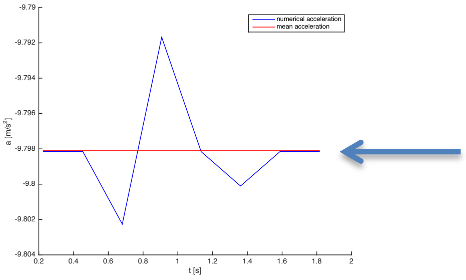
- use your script function `derivative()` to calculate the numerical velocity $v=dh/dt$ and plot it into the same figure as $h(t)$.

➢ **exercise:**

- use `derivative()` again to calculate the numerical acceleration $a=dv/dt$ and plot it into a new figure.
- add another line to this plot that shows the mean value of $a(t)$ as a straight line.

- **Note:** your final plots should look like this



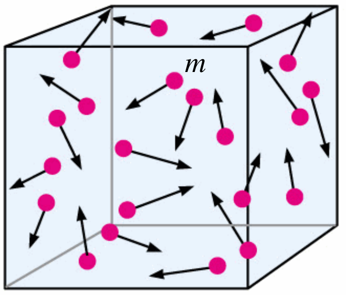


this is the sought-after value of g

MATLAB *application – Maxwell-Boltzmann distribution*

- **The Maxwell-Boltzmann distribution:**
  - $f(v)$ = distribution of velocities of atoms with mass m at temperature T

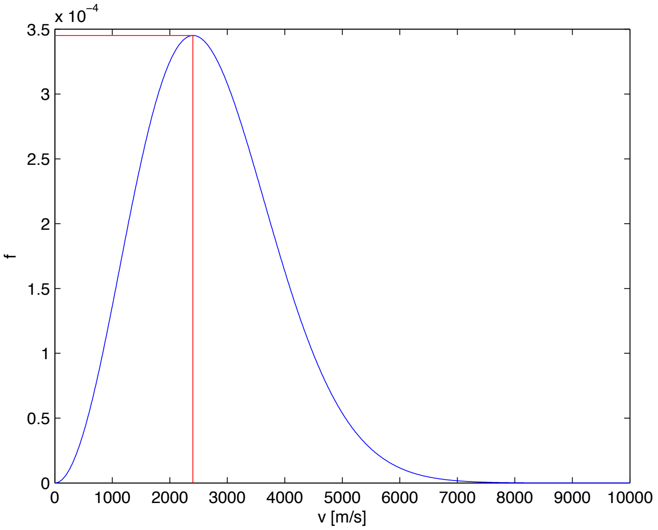
$$f(v) = 4\pi\left(\frac{m}{2\pi k_B T}\right)^{3/2} v^2 e^{-\frac{mv^2}{2k_B T}}$$



temperature T

➢ **exercise:**

- write the script **MaxwellBoltzmann.m** that plots the distribution function $f(v)$ for a proton



…and determines the maximum `vmax` by using a function

```
function [xmax, imax] = vecmax(x)
```

where `x()` is a vector and `xmax` the maximum value of that vector found at index `imax`. The function should also work in case there are multiple maxima/minima in `x()` and hence you need to use a `for`-loop in combination with an `if`-statement.

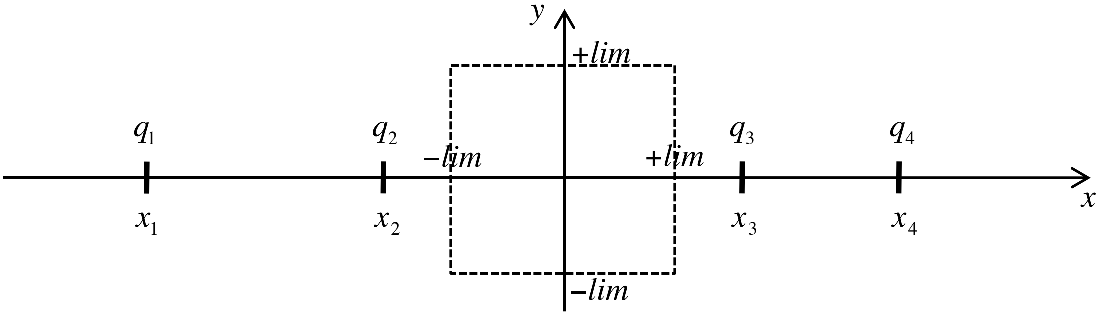
Note:

- MATLAB has in-built functions to determine max- and min-values that you can use from now on:

```
>> help max
>> help min
```

- **Coulomb charge distribution:**

Four charges q_1, q_2, q_3 y q_4 are placed on a 2D plate, but only along the x-axis:

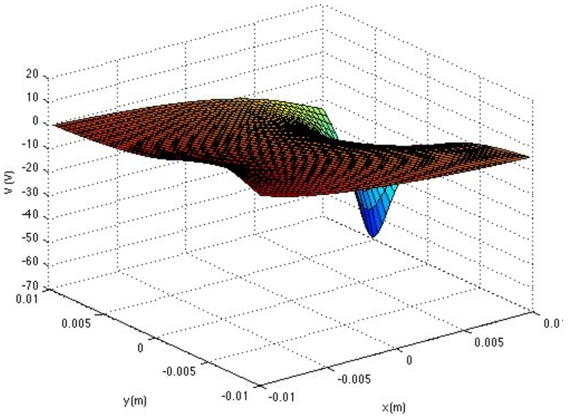


The electric potential is given as follows:

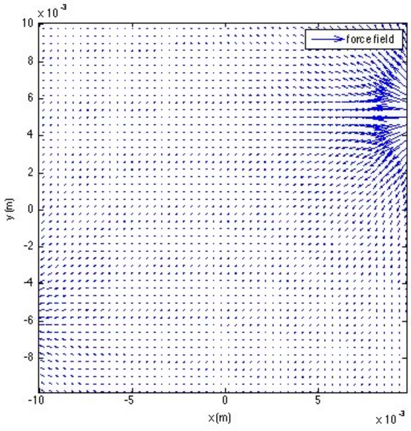
$$V(\vec{r}) = \frac{1}{4\pi\varepsilon_0}\sum_i \frac{q_i}{|\vec{r}_i - \vec{r}|}$$

➢ **exercise:**

- visualize the potential for the region x ∈ [-lim, lim] and y ∈ [-lim, lim].
- calculate and visualize the force field in the same region.
- rotate the charge distribution by 23.5° counter-clockwise and repeat the two plots.



potential of rotated distribution



force field of rotated distribution

(the relevant values are
$x_1=-4.3\cdot10^{-2}$ m, $x_2=-1.5\cdot10^{-2}$ m, $x_3=1.29\cdot10^{-2}$ m, $x_4=4.7\cdot10^{-2}$ m,
$q_1=-2.07\cdot10^{-12}$ C, $q_2=9.08\cdot10^{-12}$ C, $q_3=-16.99\cdot10^{-12}$ C, $q_4=12.48\cdot10^{-12}$ C,
lim=$0.98\cdot10^{-2}$ m)

- **Lissajous curves**

The Lissajous curves are described by the following parametric equations

$$x = A \cdot \cos(a \cdot \theta)$$
$$y = B \cdot \sin(b \cdot \theta)$$

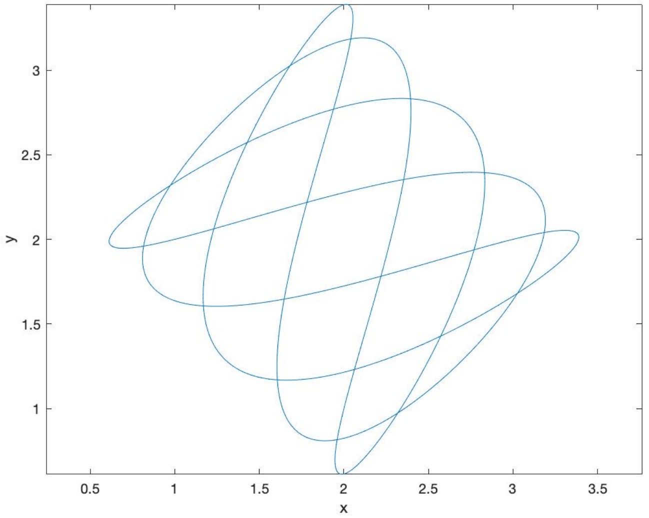
where the curves are centered on the origin of the coordinate system.

➢ **exercise:**

- write a script that calculates the Lissajous curves for the following values

  A=1, B=1, a=5, b=3, but centered at (2,2)

- plot them rotated by 45º:



MATLAB *application – particle trajectory*

A particle moves in one dimension along the x-axis with the velocity

$$v(t) = v_0 + A_0 e^{-t/\tau}\left(\cos(\omega t) - (\omega t) sin(\omega t)\right)$$

- **exercises:**

- calculate the velocity for a given interval $t \in [t_{min}, t_{max}]$ using an anonymous function for $v(t)$; this anonymous function has to take v_0, A_0, τ, and ω as arguments, too!
- plot $v(t)$ into a new figure.
- numerically calculate the acceleration and plot it into a new figure.
- numerically calculate the particle trajectory $x(t)$ and plot it into a new figure.

Use the following data:

t_{min}=1.8 s, t_{max}=4.3 s, v_0=1.5 m/s, A_0=-1 m/s, ω=4.5 s^{-1}, τ=3.3 s, x_0=-7.5 m

MATLAB *application – damped harmonic oscillator*

The Newtonian equation for the damped harmonic oscillator reads as

$$m\frac{d^2x(t)}{dt^2}+c\frac{dx(t)}{dt}+kx(t)=0$$

where m is the mass, c the friction constant, and k the spring constant. The exact solution (for $v(t=0)=v_0=0$) is given as follows

$$x(t)=\frac{x_0}{\sqrt{1-\zeta^2}}e^{-\gamma t}\cos(\sqrt{1-\zeta^2}\,\omega_0 t-\varphi)$$

with

$$\gamma=\frac{c}{2m} \qquad \omega_0=\sqrt{\frac{k}{m}}$$

$$\zeta=\frac{c}{2\sqrt{mk}} \qquad \varphi=\arccos(\sqrt{1-\zeta^2})$$

- **exercise #1:**

  - create an external function `[x]=dho_x(k,m,c,x0,t)` that calculates the solution of the damped harmonic oscillator on the time interval specified by input time vector `t()`.

- **exercise #2:**

  - plot $x(t)$ using m=1.4kg, k=6.5kg/s^2, c=0.8kg/s, x_0=2.8m, t_0=0s, t_{end}=18s.

- **exercise #3:**

  - calculate by numerical differentiation $v(t)$ and plot into the same figure of exercise #2.

- **exercise #4:**

  - numerically calculate $a(t)=dv/dt$ and plot into the same figure of exercise #2.

- **exercise #5:**

  - calculate by numerical integration of $dW/dt = - c\ v^2$ the frictional work of the oscillator.

- **exercise #6:**

  - plot the total energy $E(t)=1/2\ (mv^2+kx^2)$ into the same figure of exercise #5.
  - **Note**: to match the frictional work with the total energy you need to add E_0 to it.

MATLAB *application – Leibniz series*

Consider the following numerical series (Leibniz formula) defining π

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

- **exercise #1:**

- evaluate the finite sum

$$f(N) = \sum_{n=0}^{N} \frac{(-1)^n}{2n+1}$$

and plot it as a function of N and checking that it converges to $\pi/4$.

- **exercise #2:**

- consider now the equivalent form

$$g(N) = \sum_{n=0}^{N} \frac{2}{(4n+1)(4n+3)}$$

plotting it into the same figure as for exercise #1.

Examples

application – Monte Carlo integration

- Monte Carlo integration is a technique for numerical integration using random numbers. Here we will see it in action using a Lissajous curve as the function to be integrated:

The points of a Lissajous curve with period ratio 1:2 and phase p/2 can be described as follows:

$$4(x^4 - x^2) + y^2 = 0$$

➢ **exercise:**

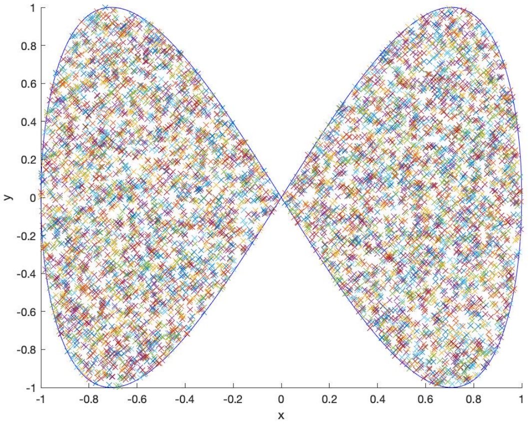
- Plot the Lissajous curve on the interval $x \in [-1, +1]$.

➢ **exercise:**

- Calculate via numerical integration (as learnt in Unit 1) the area covered by the curve.

➢ **exercise:**

- Calculate the area via Monte Carlo integration:
  - generate a pair of random numbers (r_x,r_y) from a uniform distribution
  - so that $r_x \in [x_{min}, x_{max}]$ and $r_y \in [y_{min}, y_{max}]$.
  - check, if that point lies within $y(x)$.
  - repeat this process N times where N_{in} will count how often (r_x,r_y) lies inside $y(x)$
  - the area will then be $A \approx \frac{N_{in}}{N}(x_{min} - x_{max})(y_{min} - y_{max})$
- Compare the values for the areas
- If you add the points inside $y(x)$ the plot will eventually look like this:



matrix functions							
size	diff	gradient	max	min	prod	diag	sort
length	size	numel	transpose	/	inv	gradient	
ones	zeros	eye	meshgrid	norm			
plotting							
meshgrid	mesh	surf	surfc	surfl	contour	quiver	quiver3
colormap	colorbar	shading	waterfall				
script commands							
for	while	if	else	end	function	return	switch/case
input	disp	break					
useful functions							
rand	randn	rem	mod	median	mean	mode	std
factorial	find						

- **please familiarize yourself with all of these commands, functions, variables, etc., even if they have not been discussed in class:**

from now on you must know how to use them all!

- you further need to know the following...
  - basic matrix operations
  - plotting surfaces and multi-dimensional functions, respectively
  - defining and using own functions with one or multiple arguments
  - using for-loops
  - using if-then-else statements