Neuroimmunology &
Imaging Group

DZNE
German Center for
Neurodegenerative Diseases
within the Helmholtz Association

MATLAB Workshop

A beginner’s guide into scientific programming and data analysis

4th DZNE Doctoral Retreat
23 to 25 October 2019 Dresden

Fabrizio Musacchio

Neuroimmunology and Imaging Group
DZNE Bonn

What we will learn today

Introduction into MATLAB

We become familiar with MATLAB. We learn basic commands as well as MATLAB scripts and we write our first program.

Basic Steps of Data Analysis and Presentation

We learn and apply basic data processing, analysis and visualization steps. We apply our methods to artificially generated data first (we want to learn, how data ‘work‘ and how signals are generated).

Data I/O

We see how to import import data and finally we apply what we have learned so far to real data set. We also learn how to save our results.

Advanced Steps of Data Analysis

At the end, we see how to handle sets of data at once and we take a glimpse on clustering data.

Time Schedule

09:00-11:10: **Workshop Part 1**

11:10-11:30: Free time and question session

11:30-12:30: Lunch break

12:40-14:30: **Workshop Part 2**

14:30-15:00: Coffee break

15:00-16:30: **Workshop Part 3**

16:30-17:00: Coffee break

17:00-19:00: Invited Speakers Talks

What is Scientific Programming?

In natural sciences, we observe processes in nature that follow specific laws, the laws of nature. Some of these laws are known and can be described mathematically. Others are unknown: We only know a part of the whole image or we simply have not understood what is going on. We then try to approximate the underlying laws by phenomenological approaches or axioms.

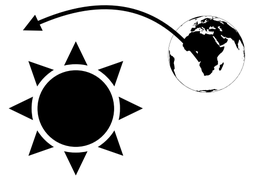
To do this, we make a hypothesis and do experiments which must be reproducible and validate the hypothesis. The measurements of these experiments have to be analyzed, which requires tools: mathematics. Therefore, we need calculators like our brains and a lot of sheets of paper and a pen. For the more complex problems, we use a computer.

First of all, we have to tell the computer what we actually want from it. For this we write a program in a language that both, the computer and we, understand. This language should also be capable of entering our mathematical operations, algorithms and models in order to validate our theories and to make further predictions.

Therefore, *Scientific Programming* is not just "coding" or giving a machine some commands. It is the computational processing of mathematical formulations which are directly related to the laws of nature (physics, biology, chemistry, neuroscience, ...).

Observation: Lighter (*m*) objects are attracted by heavier (*M*) ones:

Copyright protected material



Hypothesis: There is a force of attraction (gravitation), which can be mathematically described:

$$\mathbf{F} = -G\frac{m_s M}{r^2}$$

Copyright protected material

Experiments: Proof of hypothesis by observations and measurements:

Copyright protected material

Expand the theory and apply and proof it to other questions. Perform simulations for further predictions:

Simulation of the collision of two black holes (Max Planck Institute for Gravitational Physics, 2012)

Copyright protected material

What is Scientific Programming?

Observation: Intracellular recording of the squid giant axon action potential (AP)

Hodgkin & Huxley (1945)

Hypothesis: The recorded AP follows a certain excitation model

$$I = C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} + \bar{g}_{\mathrm{K}} n^4 (V_m - V_K) + \bar{g}_{\mathrm{Na}} m^3 h (V_m - V_{Na}) + \bar{g}_l (V_m - V_l),$$

$$\frac{dn}{dt} = \alpha_n(V_m)(1-n) - \beta_n(V_m)n$$

$$\frac{dm}{dt} = \alpha_m(V_m)(1-m) - \beta_m(V_m)m$$

$$\frac{dh}{dt} = \alpha_h(V_m)(1-h) - \beta_h(V_m)h$$

Hodgkin & Huxley (1952)

Huxley (left) and Hodgkin (right)
Cover of the 1963 Nobel Prize Programme

Model predictions compared to experimental results

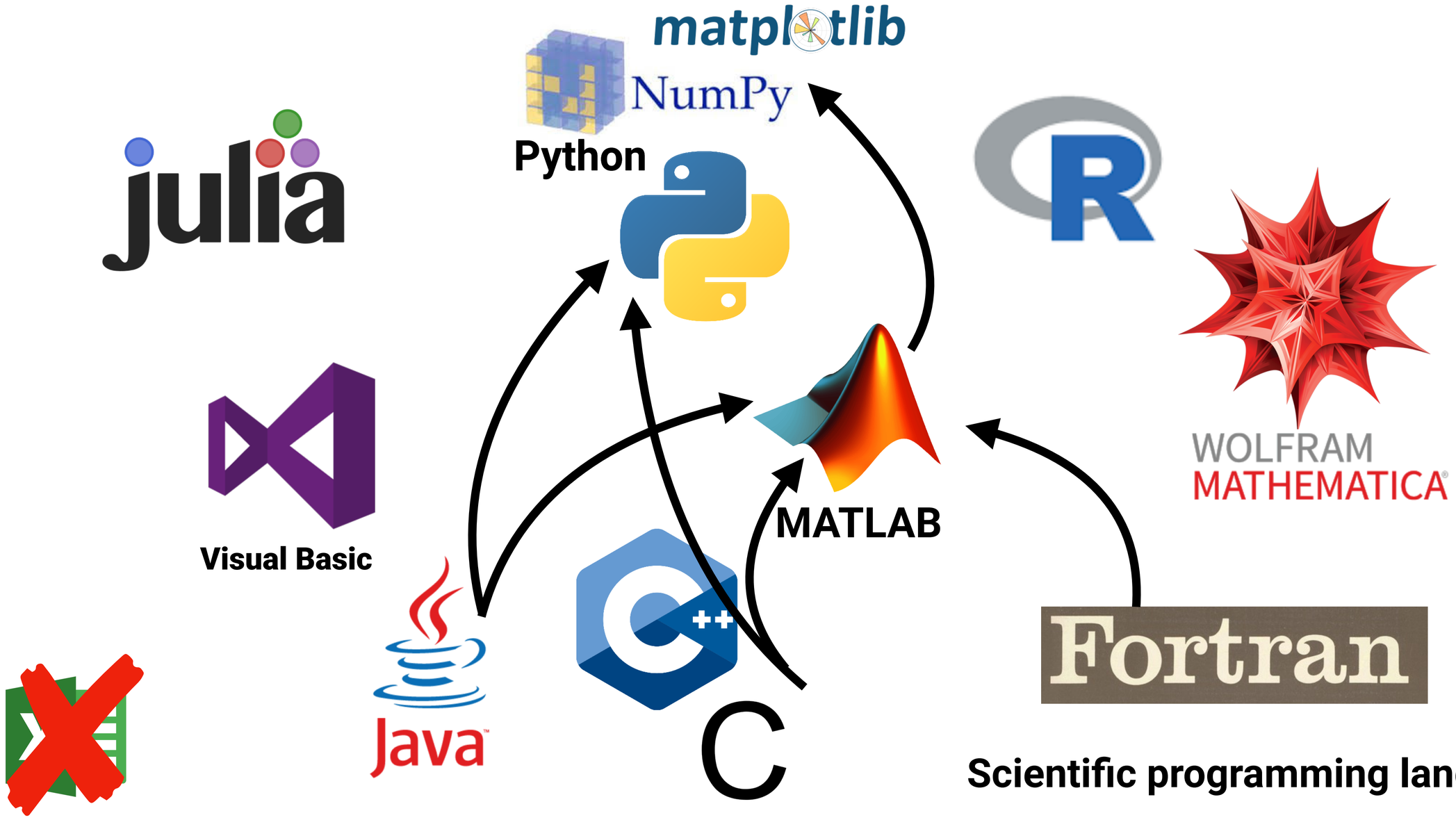
"Computer"

Hodgkin & Huxley (1952)

Model also explains spike trains:

J Physiol. 2012 Jun 1; 590(Pt 11): 2571–2575.
Published online 2012 May 31. doi: 10.1113/jphysiol.2012.230458

Why MATLAB?


matplotlib
NumPy
Python
julia
R
WOLFRAM
MATHEMATICA
Visual Basic
MATLAB
Java
C++
C
Fortran
Scientific programming languages

Why MATLAB?

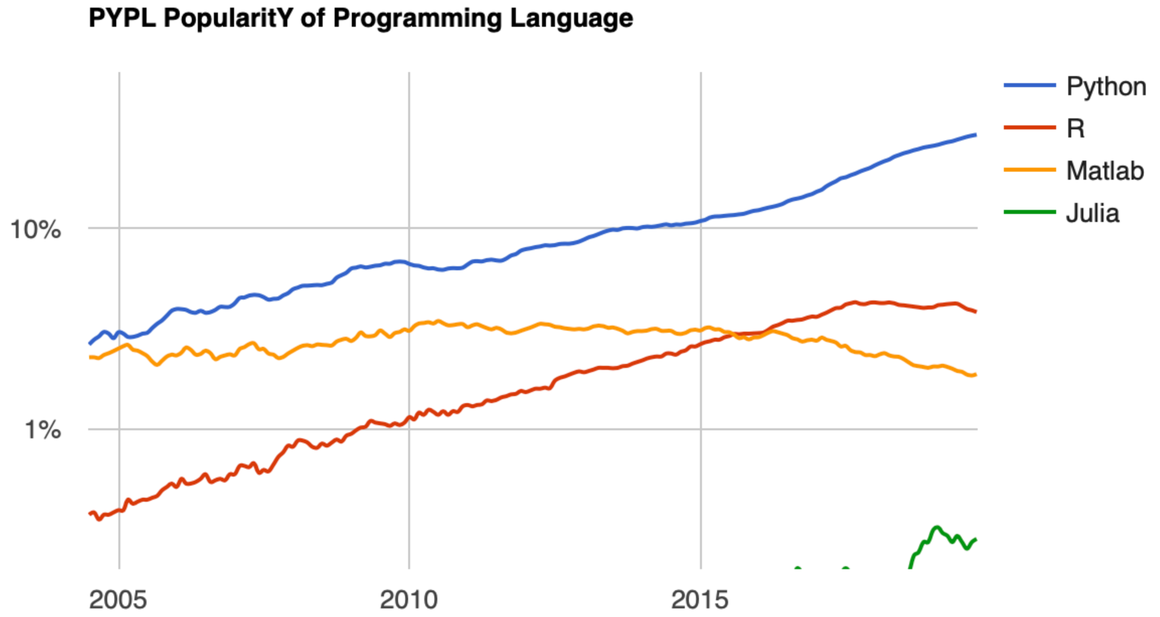
PYPL PopularitY of Programming Language:

Worldwide, Oct 2019 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.49 %	+4.5 %
2		Java	19.57 %	-2.4 %
3		Javascript	8.4 %	+0.1 %
4		C#	7.35 %	-0.4 %
5		PHP	6.34 %	-1.2 %
6		C/C++	5.87 %	-0.4 %
7		R	3.82 %	-0.2 %
8		Objective-C	2.6 %	-0.7 %
9		Swift	2.57 %	-0.1 %
10		Matlab	1.87 %	-0.2 %

pypl.github.io/PYPL.html

> “**Worldwide**, Python is the most popular language, Python grew the most in the last 5 years (18.9%) and Java lost the most (-7.0%)”



Why MATLAB?

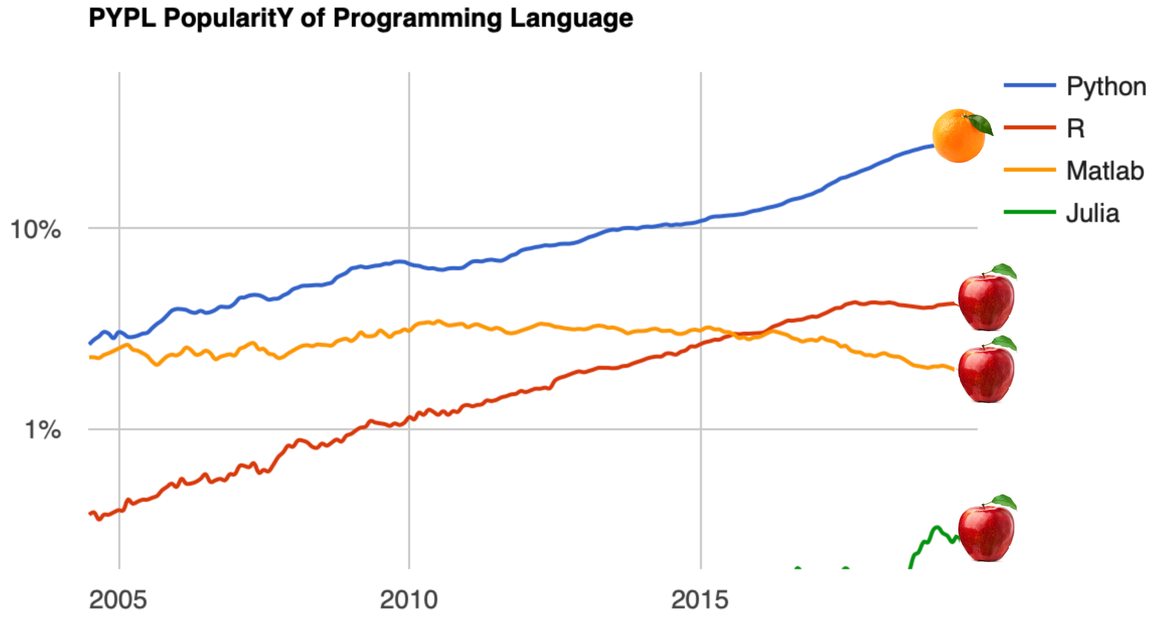
PYPL PopularitY of Programming Language:

Worldwide, Oct 2019 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.49 %	+4.5 %
2		Java	19.57 %	-2.4 %
3		Javascript	8.4 %	+0.1 %
4		C#	7.35 %	-0.4 %
5		PHP	6.34 %	-1.2 %
6		C/C++	5.87 %	-0.4 %
7		R	3.82 %	-0.2 %
8		Objective-C	2.6 %	-0.7 %
9		Swift	2.57 %	-0.1 %
10		Matlab	1.87 %	-0.2 %

pypl.github.io/PYPL.html

> "**Worldwide**, Python is the most popular language, Python grew the most in the last 5 years (18.9%) and Java lost the most (-7.0%)"



Caution, don't compare apples with oranges:

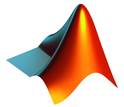
/ interpreted, high-level, general-purpose programming language (but highly expandable with numerical computation modules!)

/ fully scientific, numerical computing language (e.g., MATrix LABoratory)

matplotlib NumPy

Why MATLAB?

MATLAB is a little bit easier at the beginning. But what you learn in MATLAB, can be easily transferred to Python and vice versa.



and ~~**s.**~~

Define variables and arrays:

```
a=1.0;
B=[1 2 3];
```

or

```
B=[1, 2, 3];
```

```
a=1.0
B=[1, 2, 3]
```

***for*-loop:**

```
for i=0:10
  fprintf(1, ,%i\n', i)
end
```

```
for i in range(0,10,1):
  print(i)
```

Initialize large arrays:

```
A = eye(100);
```

```
import numpy as np
A = np.identity(100)
```

Quick data print out:

```
B(1)
```

```
print(B[1])
```

Show all rows of column 1 of a 2D arrays:

```
C=[1 2 3 ; 4 5 6; 7 8 9];
C(:,1)
```

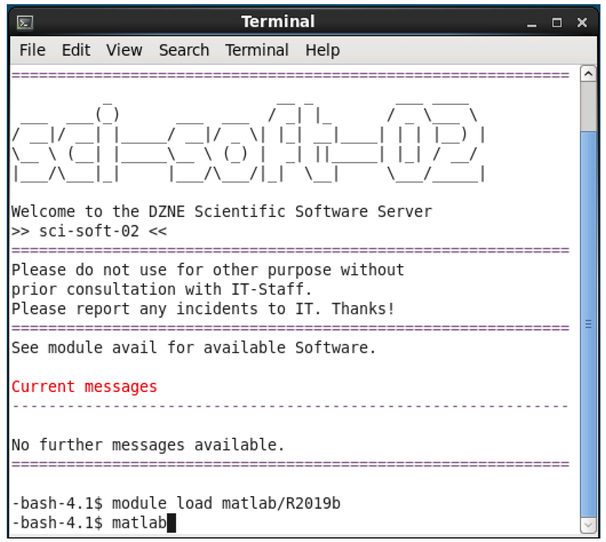
```
import numpy as np
D = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
C_sub = C[0:,0:1]
print(C_sub)
```

Logical expressions:

```
d=1;
if d==1
  fprintf(1, 'Match!\n')
end
```

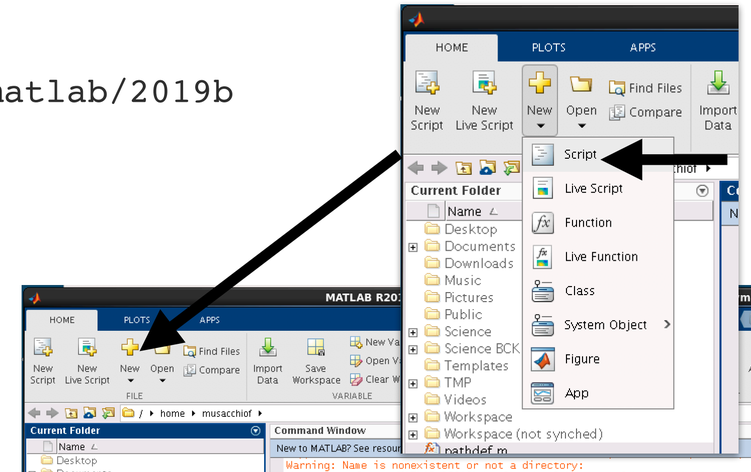
```
d=1;
if d==1
  print('Match!')
```

Starting MATLAB

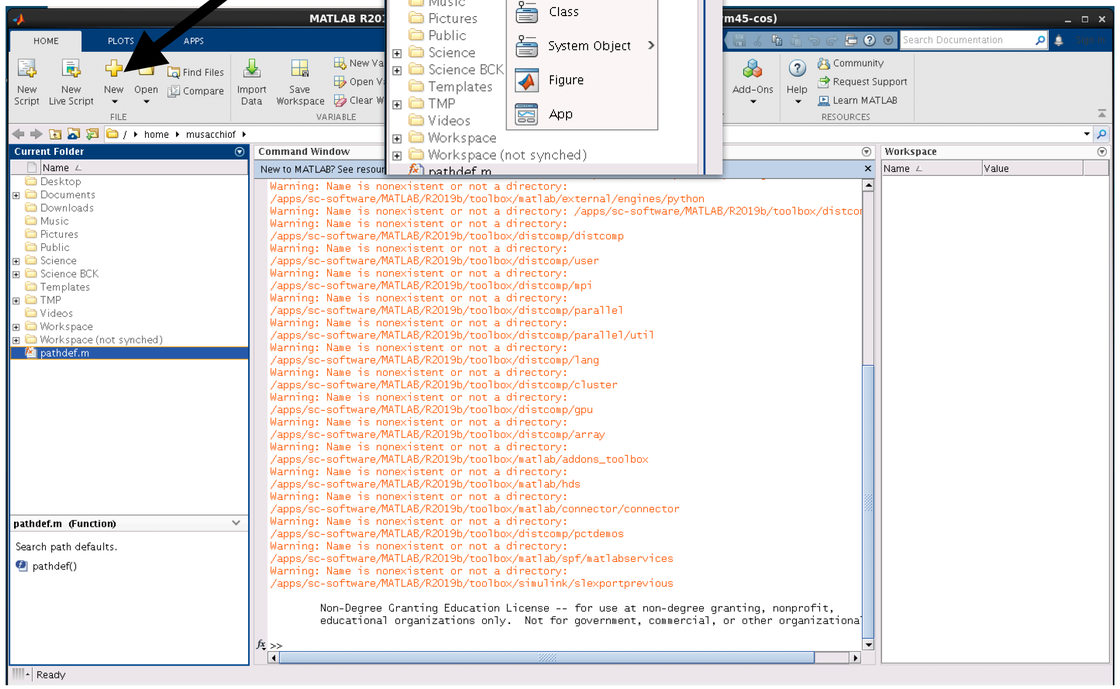


1. At the DZNE, open a terminal (on a linux remote), and enter the following commands:

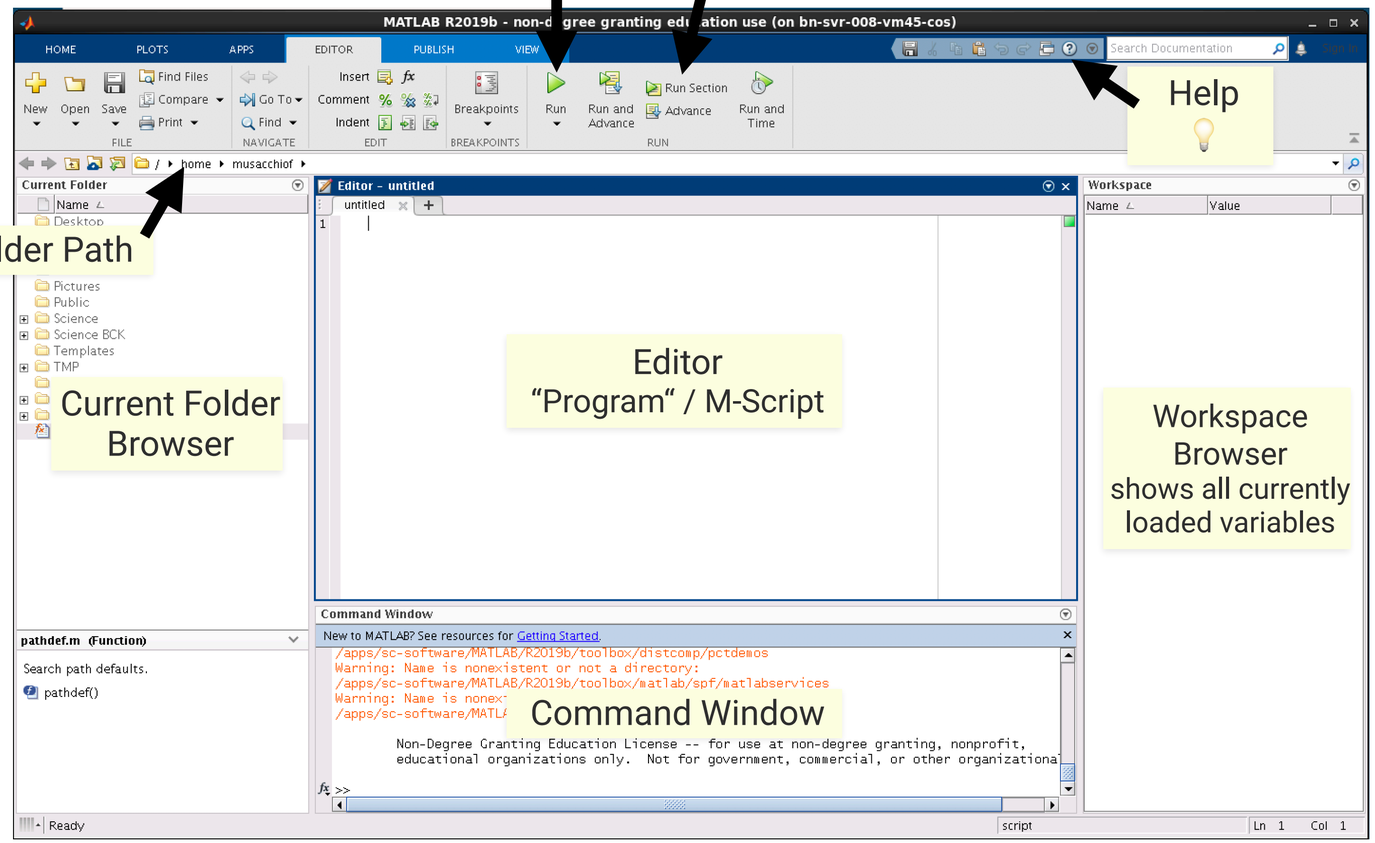
```
module load matlab/2019b
matlab
```



2. Please, click on the “New“ button and open a new “Script“
This is only necessary on our very first start today.



The Integrated Development Environment (IDE) or Graphical User Interface (GUI)



Command Window:

- like a terminal in Linux
- commands are directly executed:

```
>> 5+5
ans =
    10
```

```
>> 5*5
ans =
    25
```

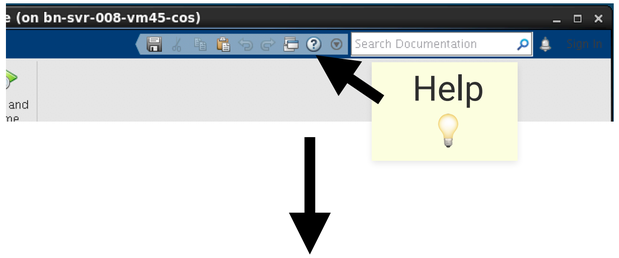
```
>> fprintf(1, 'Hello world\n')
Hello world
```

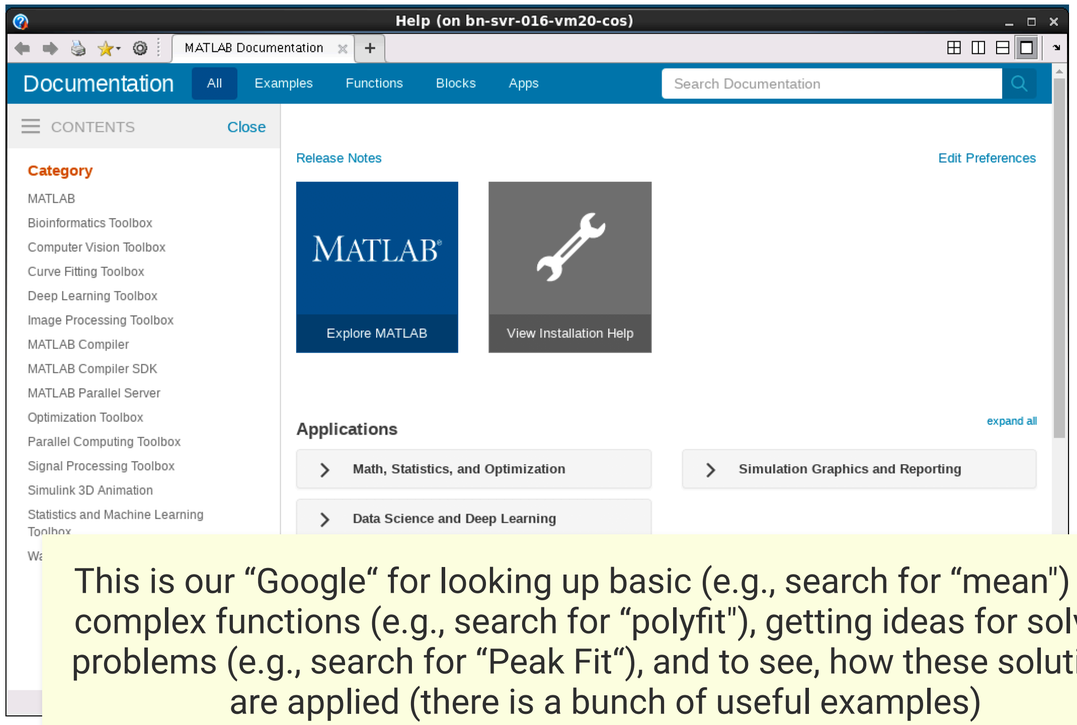
- but: once MATLAB is closed, all commands are gone
- and: a program consists of more than just one line of code

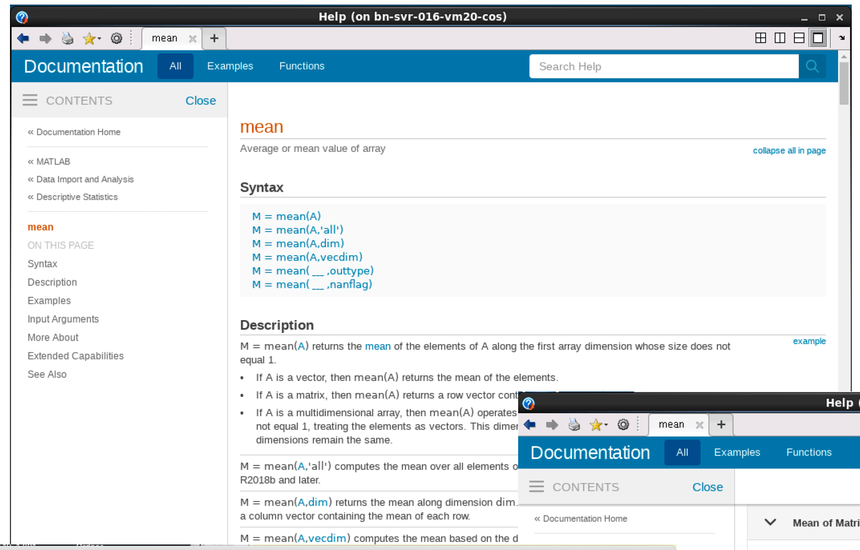
Editor:

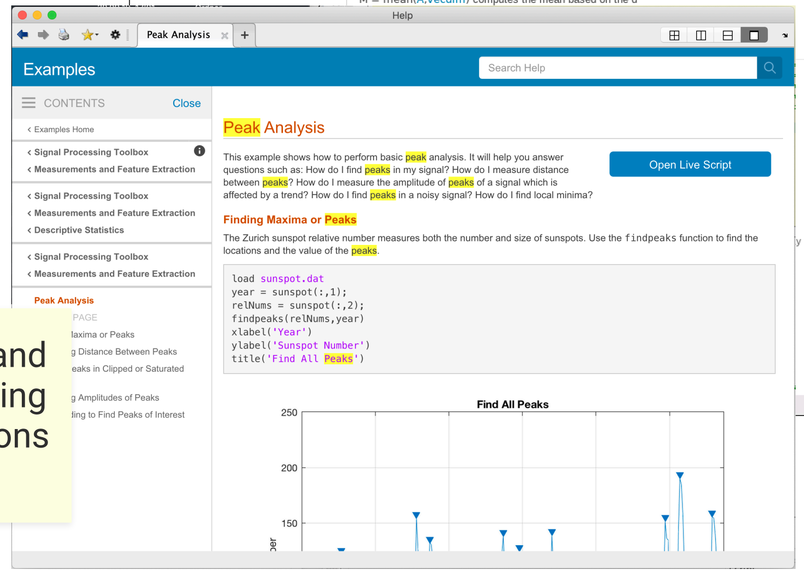
- a MATLAB program is a “script“ (being more precise: an M-script with file ending *.m)
- this is our Swiss army knife with a huge range of useful functions and tools (like code folding, autocomplete, syntax highlighting)

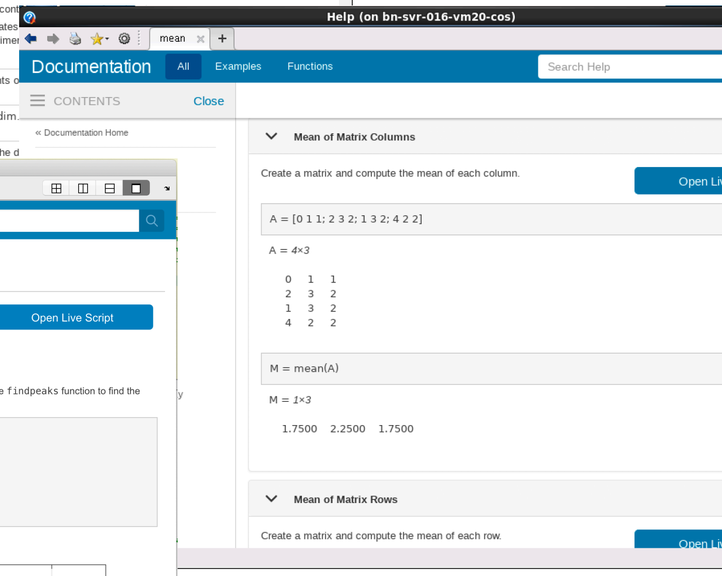
The MATLAB Help Browser





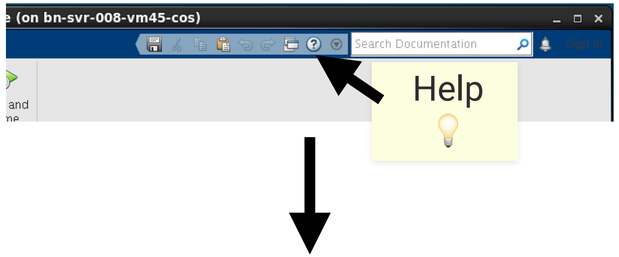


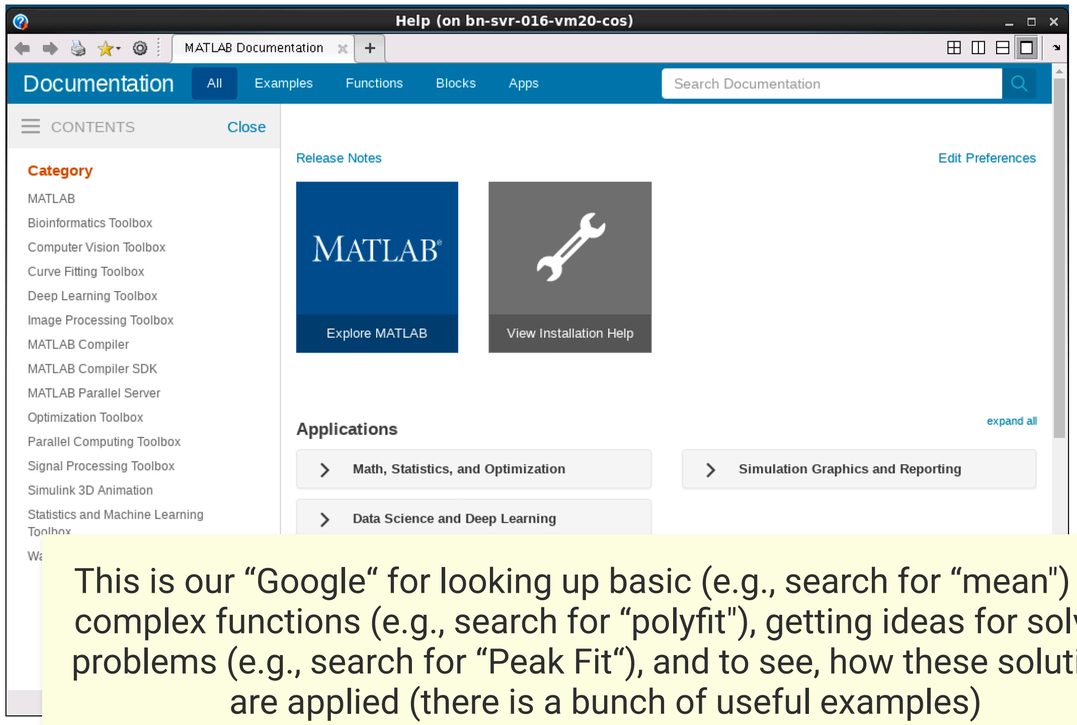




This is our “Google“ for looking up basic (e.g., search for “mean") and complex functions (e.g., search for “polyfit"), getting ideas for solving problems (e.g., search for “Peak Fit“), and to see, how these solutions are applied (there is a bunch of useful examples)

The MATLAB Help Browser





This is our “Google“ for looking up basic (e.g., search for “mean") and complex functions (e.g., search for “polyfit"), getting ideas for solving problems (e.g., search for “Peak Fit“), and to see, how these solutions are applied (there is a bunch of useful examples)

Each command or function often has a mathematical explanation.

Use cross-references: If the name of the function you are currently searching for is unknown, look for related terms.

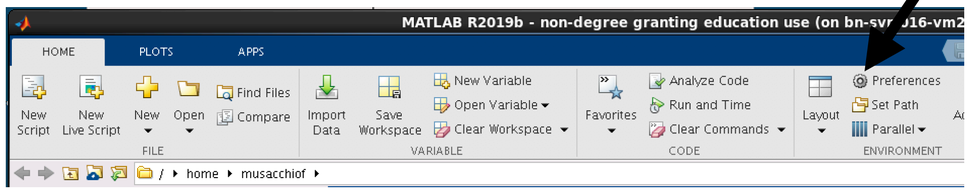
For any thing else, ask the community:

- de.mathworks.com/matlabcentral/answers/index
- stackoverflow.com/questions
- de.mathworks.com/matlabcentral/fileexchange
- many other academic websites (universities, institutes…)

Before we start, two initial settings

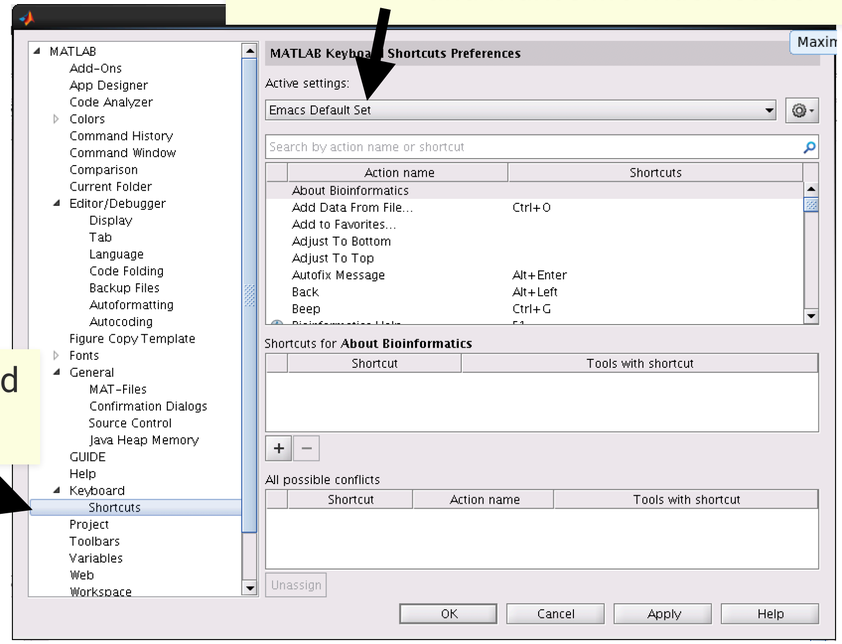
(just for the very first start of MATLAB)

1. Go to preferences



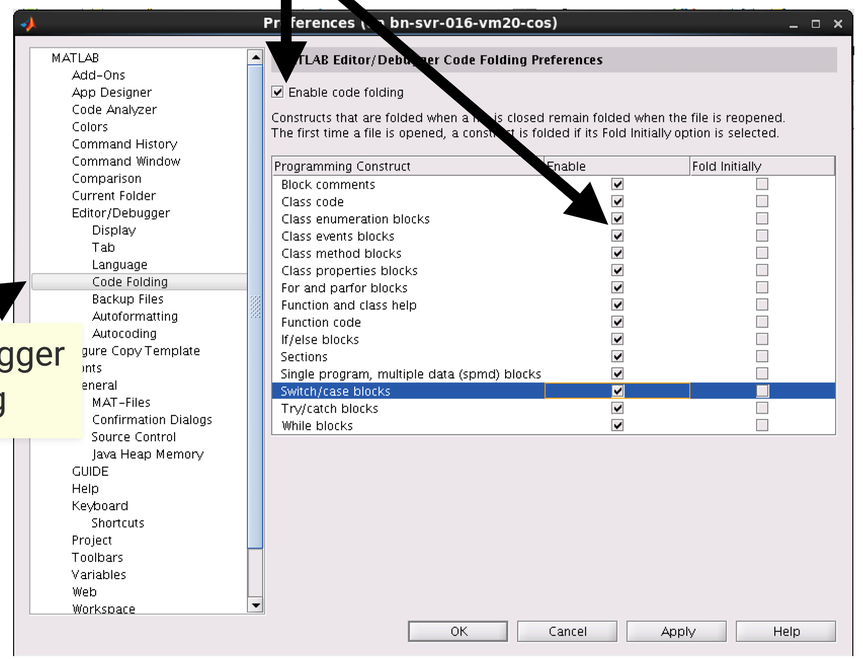
3. Set Active settings to “Windows Default Set“
This enables default shortcuts like Ctrl-C and Ctrl-V and makes life much easier

2. Go to Keyboard ⟶ Shortcuts



4. Go to Editor/Debugger ⟶ Code Folding

5. Mark Enable code folding + enable all
We will profit from this very soon



Command Window: Input of Commands

Use for basic arithmetic:

```
>> 3+5
>> 3-5
>> 3/5
>> 3*5
>> c=4;
>> c^2
>> c^(1/2)
>> 3/5+6
>> 3/(5+6)
>> (3/5)+6
```

We did it already :-D

Many built-in functions and constants:

```
>> e
>> pi
>> sqrt(c)
>> exp(0)
>> exp(4)
>> log(exp(4))
```

general use of function: `function(arguments)`

```
>> sin(0)
>> sin(90)
>> sin(180)
>> sin(270)
>> sin(360)
```

```
>> sind(0)
>> sind(90)
>> sind(180)
>> sind(270)
>> sind(360)
```

What is the difference between `sin` and `sind`?

angles in radians: a=α*π/180 [rad]

angles in degree: α=a*180/π [deg] or [°]

```
>> sin(0)
>> sin(pi/2)
>> sin(pi)
>> sin(3*pi/2)
>> sin(2*pi)
```

Take care in which units you are calculating. The same accounts for `cos` and `cosd`, `tan` and `tand`…

What do you notice? Which gives the correct result?

π in MATLAB is not the real *π*, but an approximation and therefore a floating-point number close to '*π*'. Hence, trigonometric functions around *π* may have errors close to machine precision.

Input of variables and arrays:

```
>> a=6;
>> b=4;
>> a+b
>> A=1/6;
>> a+A
>> data_value = 99;
```

(nearly) always use "`;`" at the end of array definitions

Never use built-in variables as variable names! (`e=7` ⚡)

```
>> d=[1 2 3];
>> D=[1;2;3]
```

or `d=[1, 2, 3];`

What is the difference between `d` and `D`?

`>> D'` or `>> transpose(D)` transpose operator: `'` or `transpose()`

Command Window: Input of Commands

Working with arrays:

```
>> M=[1 2 3; 4 5 6; 7 8 9]
>> M(1,3)
>> M(1:3,3)
>> M(:,3)
>> M(:,end)
>> M(end-1,end)
```

(*m*×*n*)-matrix, *m* rows, *n* columns
1st row & 3rd column value
all rows of 3rd column
all rows of 3rd column
all rows of 3rd (=last) column
pre-last row & last column value

```
>> M'
>> inv(M)
>> diag(M)
>> mean(M)
>> mean(mean(M))
```

The same accounts for `std`, `var` or `sum`.

```
>> M+8
>> M*8
>> M(:,1)+M(:,3)
>> M(1,:)+M(3,:)
>> M+d
>> M+d'
>> M.*d
```

What is the difference?

Two useful commands:

```
>> size(M)
>> numel(M)
```

or

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

Generating arrays automatically:

```
>> Z  =zeros(5)
>> ZZ =zeros(5,5)
>> ZZZ=zeros(5,3)
```

What is the difference?
`zeros(m,n)` creates an (*m*×*n*)-matrix whose elements are equal to 0.
`zeros(m)` creates squared (*m*×*m*)-matrix

```
>> I =eye(5,5)
>> II=ones(5,5)
>> R =rand(5,5)
```

identity matrix
array of all ones
matrix with random numbers between [0,1]

The same works for column and row vectors:

```
>> z =zeros(1,5)
>> zz=zeros(5,1)
>> i =eye(5,1)
>> ii=ones(5,1)
>> r =rand(5,1)
```

The colon operator

```
>> f=1:10
>> f=0:10
>> f=0:0.5:10
```

or better:

```
>> df=0.5
>> f=0:df:10
```

a) Create a 5x5 random numbers matrix and let the values range between [0,100].
b) Use the colon operator to create a (1x5)-vector and set all values equal to 13.

Command Window: Input of Commands

Let's make use of what we have got so far:

```
>> sind(0)
>> sind(90)
>> sind(180)
>> sind(270)
>> sind(360)
```

```
>> alpha = [0 90 180 270 360]
```

or ?

```
>> dalpha=90;
>> alpha =0:dalpha:360
>> y=sind(alpha)
```

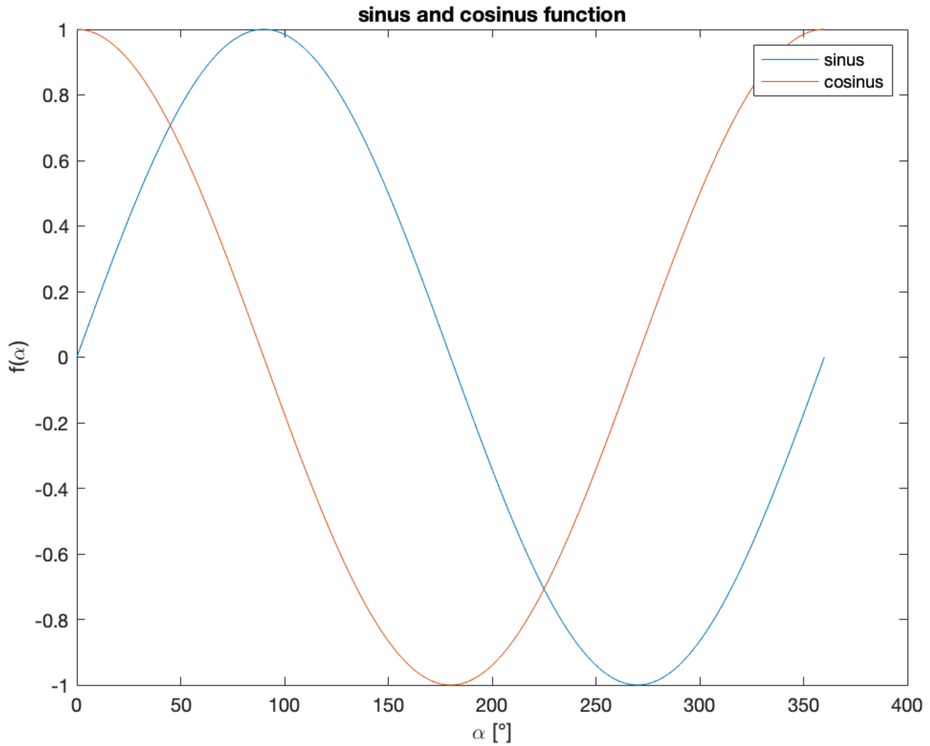
It's time to see something:

```
>> plot(alpha,y)
```
creates a plot

```
>> figure(1)
>> plot(alpha,y)
```
creates a plot in the specific figure #1

```
>> plot(alpha,y, '-k')
```
defining line color and style

```
>> hold on
>> plot(alpha+dalpha,y, '--m')
```
`hold on` allows to plot more than one plot into one figure



a) Plot the Sinus and Cosinus functions, both ranging from 0° to 360° in one figure with an adequate step-size.
b) Look up the help entry for `plot` and adjust the plots' Line Style and Color (hint: "LineSpec").
c) Add Title and Axis Labels to your figure.
d) Search for the `legend` command and ad a legend to your figure.

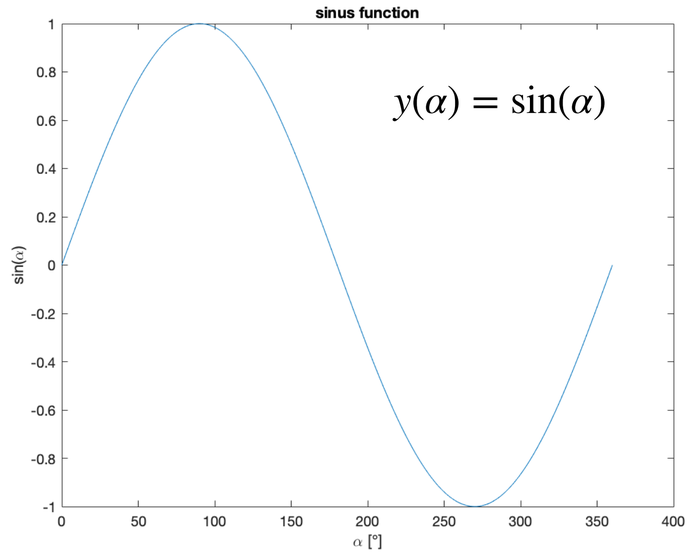
Useful command:

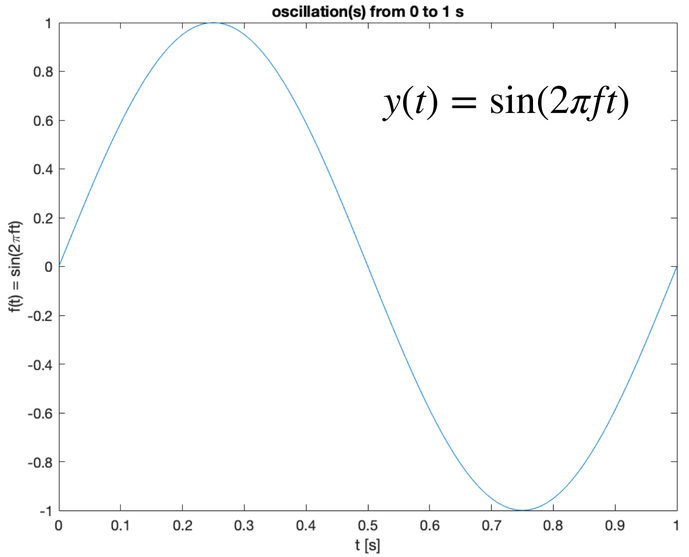
```
>> clf
```
or
```
>> figure(1);clf
```

The `clf` command clears your current figure

Command Window: Input of Commands

Instead of plotting the sinus as a function of angles from 0° to 360°, let's express it in terms of time:





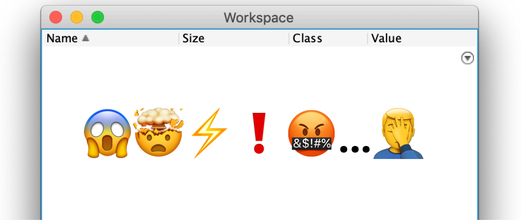

```
>> figure(1);clf
>> dalpha=1;
>> x=0:dalphha:360;
>> y=sind(x);
>> plot(x,y)
>> title('sinus function')
>> xlabel('\alpha [°]')
>> ylabel('sin(\alpha)')
```

```
>> figure(2);clf
>> dt=0.01;
>> t=0:dt:1;
>> f=1;
>> y=sin(f*pi*2*t);
>> plot(t,y)
>> title('oscillation(s) from 0 to 1 s')
>> xlabel('t [s]')
>> ylabel('f(t) = sin(2\pift)')
```

What is `f`?

Command Window: Input of Commands

Take a look at your Workspace Browser: What happened so far?



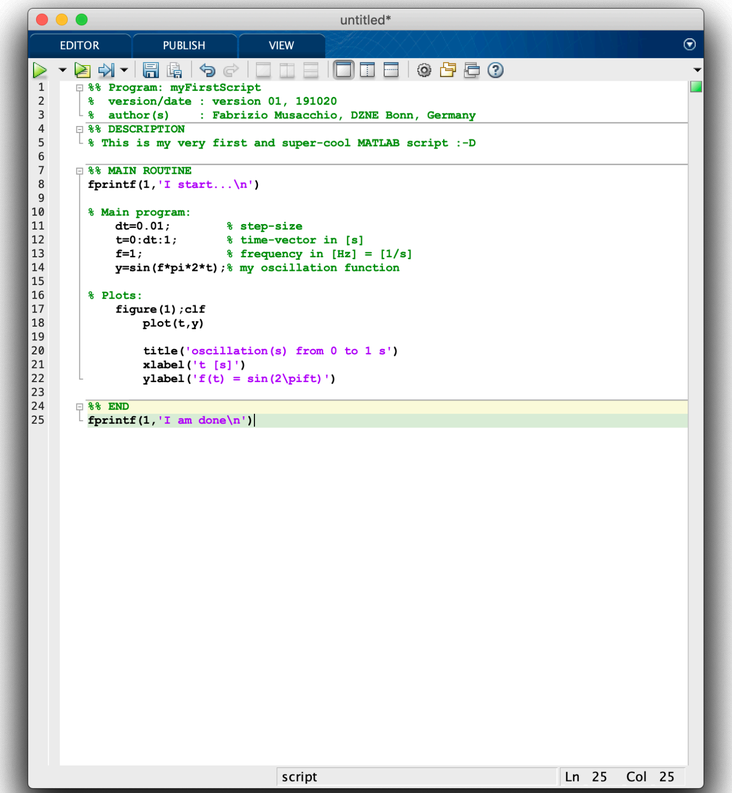
Type:

```
>> clear
```

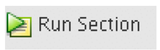
Every time you close MATLAB, MATLAB crashes or when you start a new program (initialized with the `clear` command), the entire workspace is erased.

Now it's time to make use of the MATLAB editor and to write our first program! 🙉🤓

The MATLAB Editor



Comment/Uncomment a line with Ctrl+R / Ctrl+T.



You can run cells separately with the “Run Section“ button (good, e.g., for debugging)

All commands we wrote into the Command Window can also be written into the Editor. The advantage of the editor is: We can save our commands as a MATLAB-script (M-script) and re-open and modify it as many times as we want and need.

There are some conventions regarding the filenames of an M-script:

- no umlauts or special characters (äöüß?% …) are allowed
- no “-“ (*minus*) or “␣“ (*space*) in the filename is allowed
- filename must not start with a number
- file-ending is *.m* (should be set automatically)
- don’t use built-in function names as filenames (e.g., don’t use *mean.m*, *sin.m*, ….)

Some words on good programming practices:

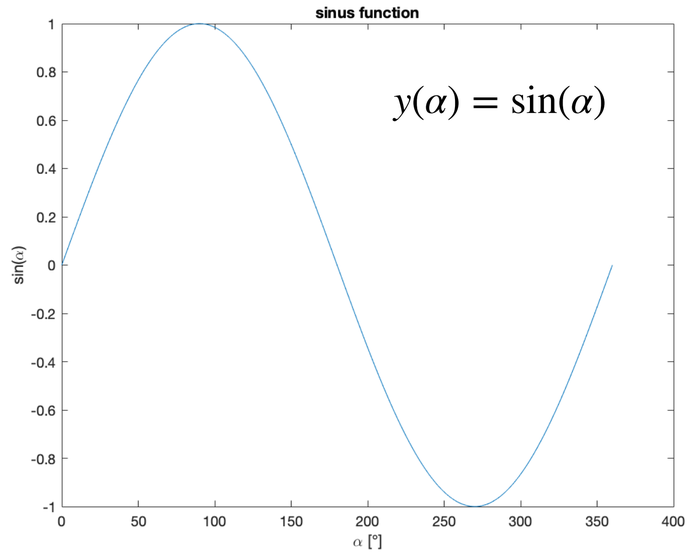
Indention: Code-indention makes your life much easier, e.g., to identify which code blocks belong together or which parts of your code or algorithm are logically connected. Always (!!!) use indention (unfortunately, MATLAB doesn’t ident text automatically, it’s up to you)!

Comments: It’s not only a nice-to-have, comments are just as much part of your program as the code itself. In MATLAB comments are initialized with “%“. Always (!!!) comment your code!

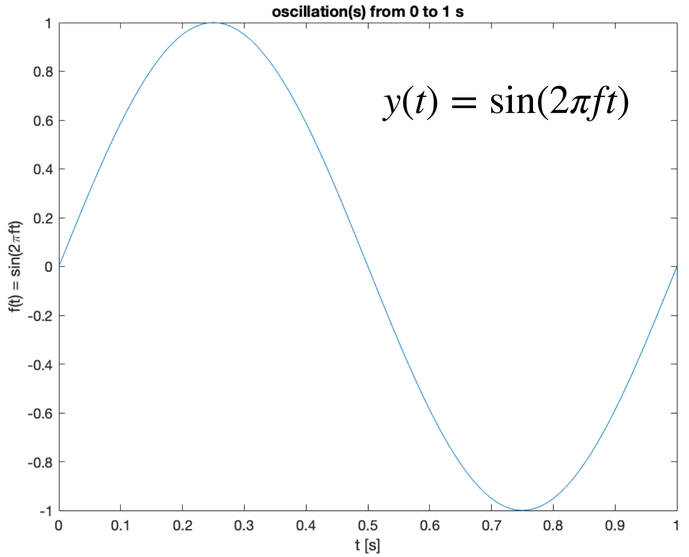
Cells: One very useful tool of the Editor is the concept of cells. Initialized with “%%“, it separates your code. Make use of it.

The MATLAB Editor: Our first script

Instead of plotting the sinus as a function of angles from 0° to 360°, let's express it in terms of time:




```
>> figure(1);clf
>> dalpha=1;
>> x=0:dalphha:360;
>> y=sind(x);
>> plot(x,y)
>> title('sinus function')
>> xlabel('\alpha [°]')
>> ylabel('sin(\alpha)')
```




```
>> figure(2);clf
>> dt=0.01;
>> t=0:dt:1;
>> f=1;
>> y=sin(f*pi*2*t);
>> plot(t,y)
>> title('oscillation(s) from 0 to 1 s')
>> xlabel('t [s]')
>> ylabel('f(t) = sin(2\pift)')
```

The MATLAB Editor: Our first script

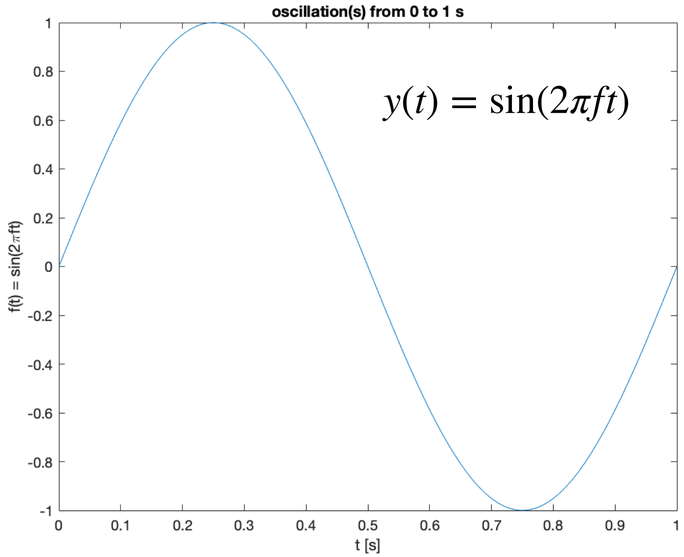
Instead of plotting the sinus as a function of angles from 0° to 360°, let's express it in terms of time:

a) Write a program, which calculates and plots an *f*=1 Hz oscillation in *t*=[0,1] s. Make sure, that
   - you use comments, code indention and cells,
   - you save your script in the correct manner, and
   - you make use of axis labels, title description and a legend.

b) Edit your code, so that some more oscillation functions with different frequencies are plotted in the same figure. (Hint: write several definitions of y=…, y1=…, y2=… with multiples of *f*)

c) Change the amplitude *A* of some of your extra oscillation functions ( ) and plot them.

d) In a new figure window, plot the sum of your oscillations (ySum=y+y1+y2…).




```
>> figure(2);clf
>> dt=0.01;
>> t=0:dt:1;
>> f=1;
>> y=sin(f*pi*2*t);
>> plot(t,y)
>> title('oscillation(s) from 0 to 1 s')
>> xlabel('t [s]')
>> ylabel('f(t) = sin(2\pift)')
```

Save your plots, e.g., as png or pdf file:

```
>> print( '-dpng', '-r600', ['myPlot.png']);
```

```
>> print( '-dpdf', '-r600', ['myPlot.pdf']);
```

The MATLAB Editor: Our first script

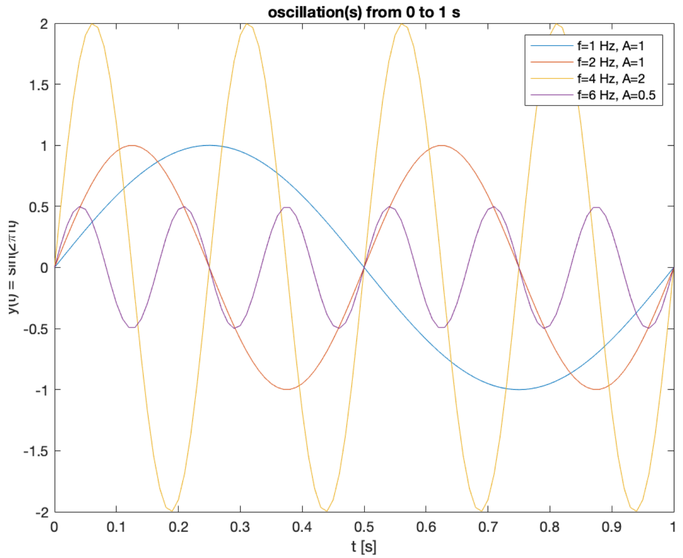
Instead of plotting the sinus as a function of angles from 0° to 360°, let's express it in terms of time:

a) Write a program, which calculates and plots an *f*=1 Hz oscillation in *t*=[0,1] s. Make sure, that
   - you use comments, code indention and cells,
   - you save your script in the correct manner, and
   - you make use of axis labels, title description and a legend.

b) Edit your code, so that some more oscillation functions with different frequencies are plotted in the same figure. (Hint: write several definitions of y=…, y1=…, y2=… with multiples of *f*)

c) Change the amplitude *A* of some of your extra oscillation functions ( ) and plot them.

d) In a new figure window, plot the sum of your oscillations (ySum=y+y1+y2…).



$$y(t) = A\sin(2\pi ft)$$

Save your plots, e.g., as png or pdf file:

```
>> print( '-dpng', '-r600', ['myPlot.png']);
```

```
>> print( '-dpdf', '-r600', ['myPlot.pdf']);
```

```
>> figure(2);clf
>> dt=0.01;
>> t=0:dt:1;
>> f=1;
>> y=sin(f*pi*2*t);
>> plot(t,y)
>> title('oscillation(s) from 0 to 1 s')
>> xlabel('t [s]')
>> ylabel('f(t) = sin(2\pift)')
```

The MATLAB Editor: Our first script

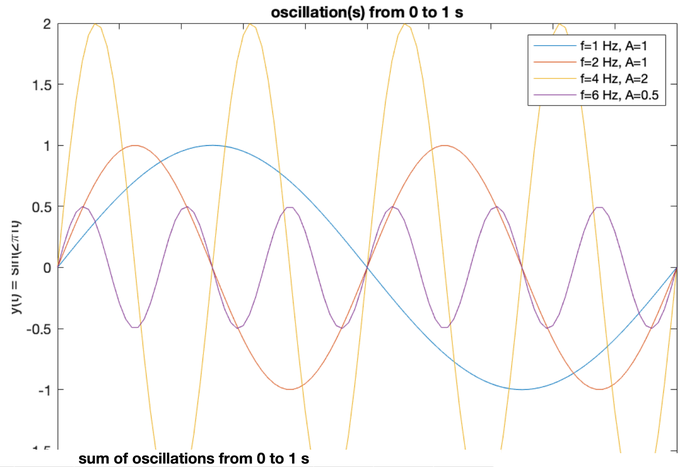
Instead of plotting the sinus as a function of angles from 0° to 360°, let's express it in terms of time:

a) Write a program, which calculates and plots an *f*=1 Hz oscillation in *t*=[0,1] s. Make sure, that
   - you use comments, code indention and cells,
   - you save your script in the correct manner, and
   - you make use of axis labels, title description and a legend.

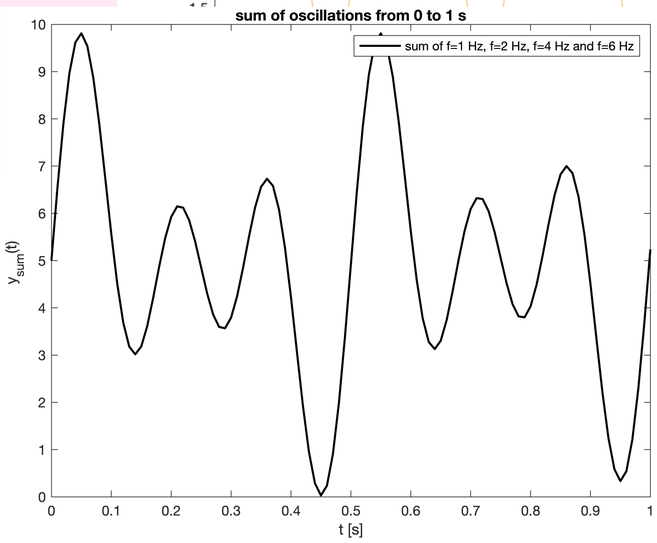
b) Edit your code, so that some more oscillation functions with different frequencies are plotted in the same figure. (Hint: write several definitions of y=…, y1=…, y2=… with multiples of *f*)

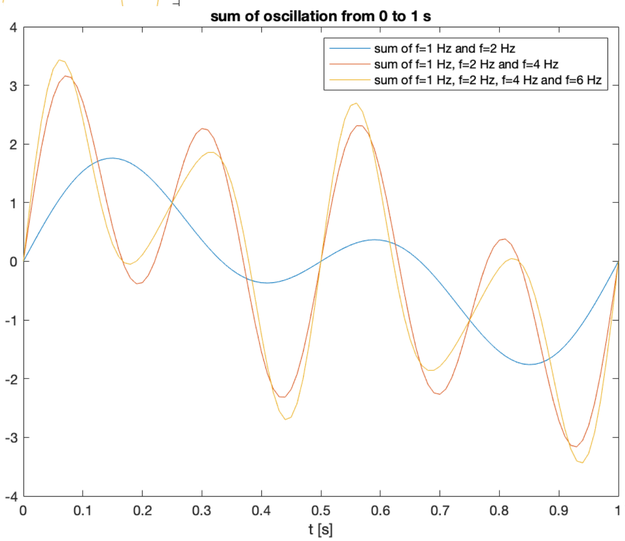
c) Change the amplitude *A* of some of your extra oscillation functions ( ) and plot them.

d) In a new figure window, plot the sum of your oscillations (ySum=y+y1+y2…).



$$y(t) = A\sin(2\pi ft)$$





Save your plots, e.g., as png or pdf file:

```
>> print( '-dpng', '-r600', ['myPlot.png']);
```

```
>> print( '-dpdf', '-r600', ['myPlot.pdf']);
```

The *for*-Loop

Instead of writing individual lines of code for each single oscillation, we can automatize this step:

```
% Main program:
    dt=0.01;          % step-size
    t=0:dt:1;         % time-vector in [s]
    f=1;              % frequency in [Hz] = [1/s]
    % my oscillation functions:
      y=sin(f*pi*2*t);
      y2=sin(2*f*pi*2*t);
      y3=2*sin(4*f*pi*2*t);
      y4=2.5*sin(6*f*pi*2*t);

% Plots:
    figure(1);clf
        plot(t,y)
        hold on
        plot(t,y2)
        plot(t,y3)
        plot(t,y4)
```

```
% Parameters:
    dt=0.01;          % step-size
    t=0:dt:1;         % time-vector in [s]

    figure(1);clf
      hold on
      for f=1:4
         y=sin(f*pi*2*t);
         plot(t,y)
      end
```

or

```
% Parameters:
    dt=0.01;          % step-size
    t=0:dt:1;         % time-vector in [s]
    f_array = [1 2 4 6];

    figure(2);clf
      hold on
      for i=1:numel(f_array)
         y=sin(f_array(i)*pi*2*t);
         plot(t,y)
      end
```

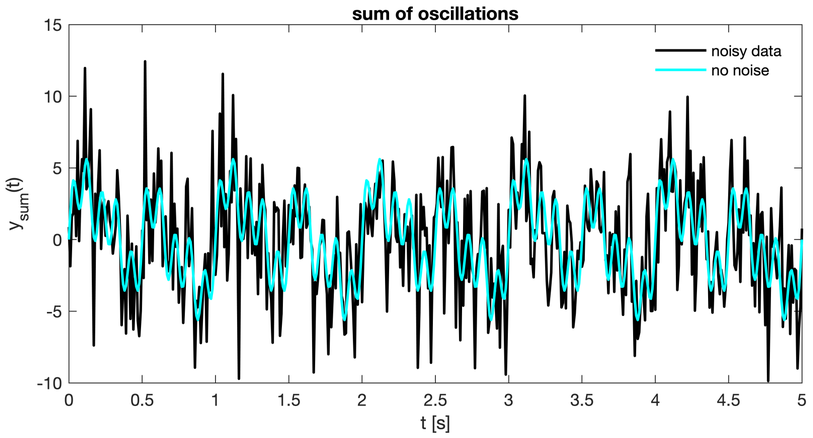
a) Write a new program, which calculates and plots oscillations with a fixed frequency of *f*=1 Hz and different amplitudes A=0.5, 1, 2, 3 using one for-loop.
b) Calculate and plot the sum of the individual oscillations (`ySum = zeros(1,numel(t)); for i=… y=…; ySum=ySum+y; …`).
c) Modify the amplitudes and re-run your script. What happens?

Basic Steps of Time Series Analysis

d) Copy your code to a new script and extend the code so that it calculates oscillations also with different frequencies (e.g., `f_array = [1 2 4 10]`).
e) Add some noise to your summed oscillations array (`ySum`) and plot it.

f) Calculate the spectrum of your summed signal *ySum*.

```
noise          = 3.0*randn(1, numel(ySum));
ySum_withNoise = ySum+noise;
```



As we have already learned on the previous slides, the signal consisting of individual Sinus function with different frequencies is “hidden“ inside our artificial, summed signal *ySum*. Is there a way, to figure out the individual frequencies in *ySum*?

Yes! ⟶ Spectral Analysis (using Fast Fourier Transformation):

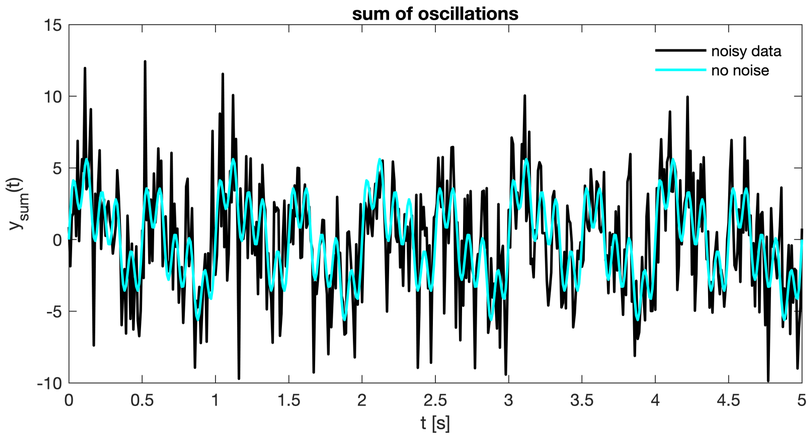
```
% Calculate some parameters for the Fast Fourier Transformation (FFT):
  Fs =numel(t)/t(end); % Sampling frequency
  L = numel(t);        % Length of signal
  f = Fs*(0:(L/2))/L;  % Frequency domain
% Do the FFT:
  Y = fft(ySum_withNoise);       % FFT
  P2 = abs(Y/L);                 % Two-sided amplitude spectrum
  P1 = P2(1:floor(L/2)+1);
  P1(2:end-1) = 2*P1(2:end-1);   % Single-sided amplitude spectrum
% Plot the spectrum:
  plot(f,P1, '-k', 'LineWidth',1.5)
```

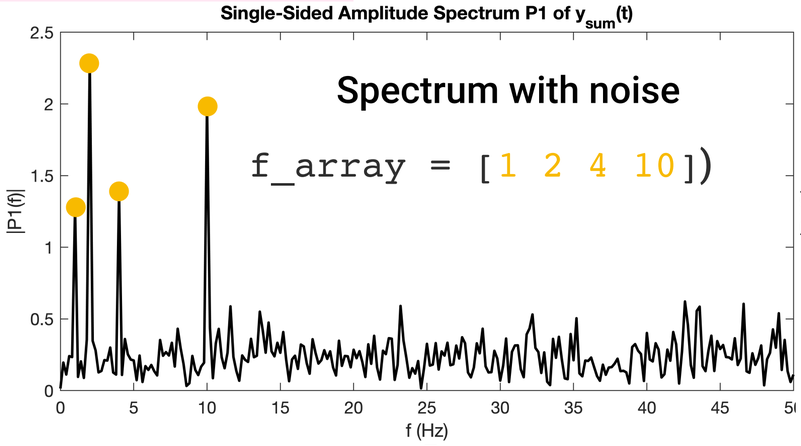
Basic Steps of Time Series Analysis

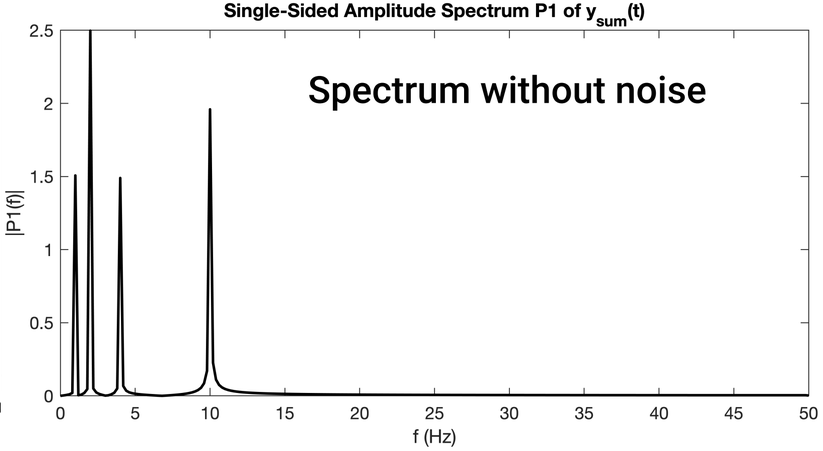
d) Copy your code to a new script and extend the code so that it calculates oscillations also with different frequencies (e.g., `f_array = [1 2 4 10]`).
e) Add some noise to your summed oscillations array (`ySum`) and plot it.

f) Calculate the spectrum of your summed signal *ySum*.
g) Vary the noise factor (3.0) and amplitudes. What happens, e.g., if you set the first amplitude to 0.5 and the noise factor to 4? What happens, if you set the noise factor to 0?

```
noise          = 3.0*randn(1, numel(ySum));
ySum_withNoise = ySum+noise;
```







useful, e.g., to identify frequencies of your interest (e.g., theta or gamma oscillations) and to separate them from unwanted frequencies (e.g., noise). We do this later.

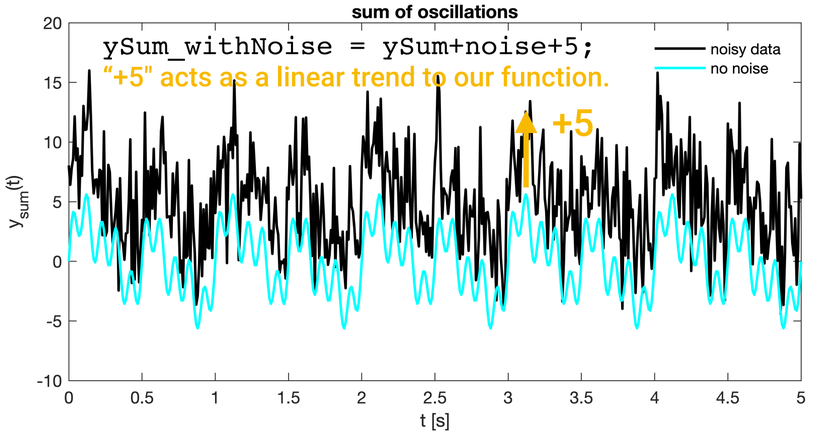
Basic Steps of Time Series Analysis

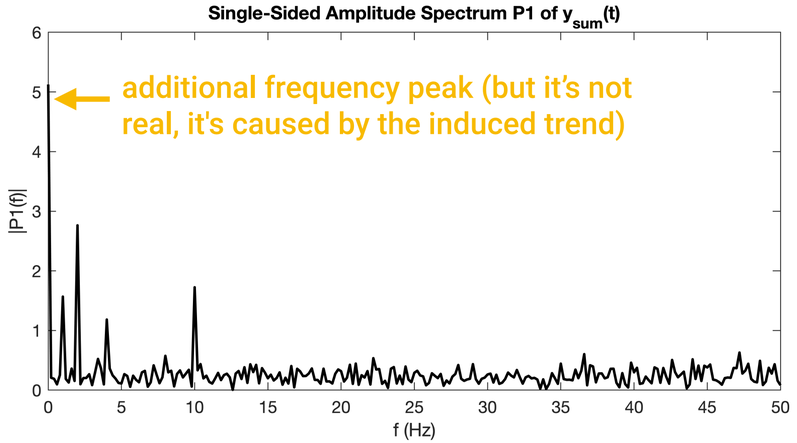
d) Copy your code to a new script and extend the code so that it calculates oscillations also with different frequencies (e.g., `f_array = [1 2 4 10]`).
e) Add some noise to your summed oscillations array (`ySum`) and plot it.

f) Calculate the spectrum of your summed signal *ySum*.
g) Vary the noise factor (3.0) and amplitudes. What happens, e.g., if you set the first amplitude to 0.5 and the noise factor to 4? What happens, if you set the noise factor to 0?
h) Add a constant factor to ySum and re-run your script. What happens to the spectrum?

```
noise          = 3.0*randn(1, numel(ySum));
ySum_withNoise = ySum+noise;
```

```
b=5;
yTrend = b;
ySum_withNoise = ySum+noise+yTrend;
```





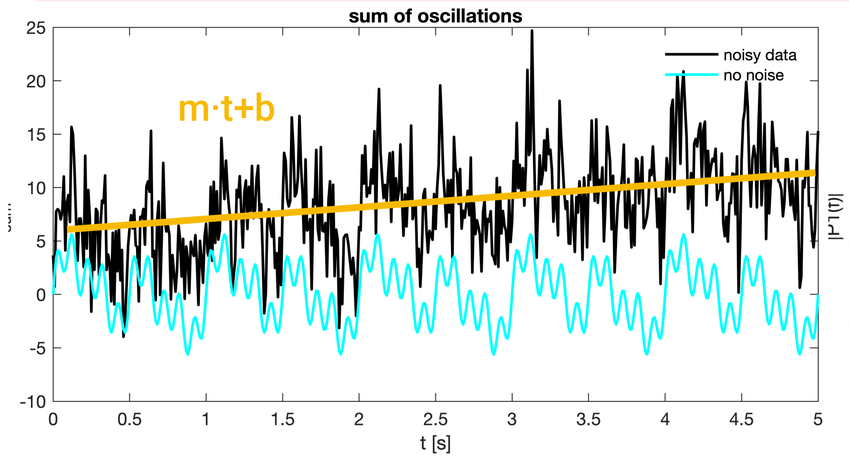
Basic Steps of Time Series Analysis

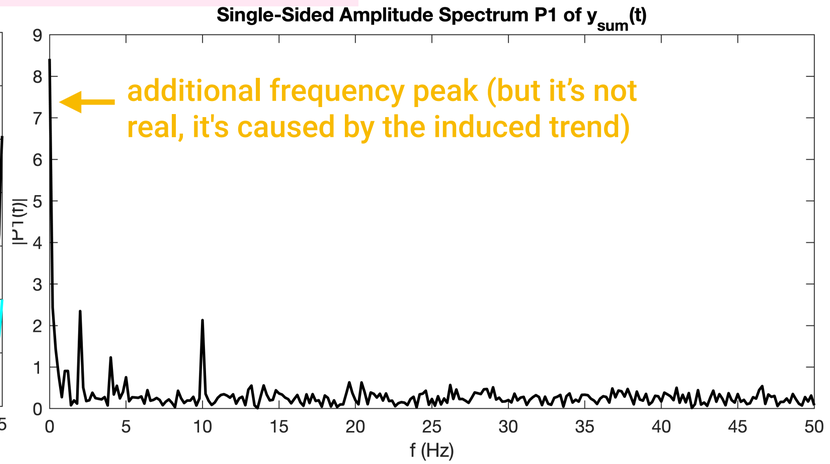
d) Copy your code to a new script and extend the code so that it calculates oscillations also with different frequencies (e.g., `f_array = [1 2 4 10]`).
e) Add some noise to your summed oscillations array (`ySum`) and plot it.

f) Calculate the spectrum of your summed signal *ySum*.
g) Vary the noise factor (3.0) and amplitudes. What happens, e.g., if you set the first amplitude to 0.5 and the noise factor to 4? What happens, if you set the noise factor to 0?
h) Add a constant factor to ySum and re-run your script. What happens to the spectrum?
i) Add a linear increasing trend (*m·t+b*) to your data.

```
noise           = 3.0*randn(1, numel(ySum));
ySum_withNoise = ySum+noise;
```

```
m=1.5;
b=5;
yTrend = m*t+b;
ySum_withNoise = ySum+noise+yTrend;
```





→ before you apply the spectral analysis of your data, first remove any linear trend.

Basic Steps of Time Series Analysis: Detrending

i) Detrend your data and re-run your script.

```
ySum_withNoise = ySum_withNoise-mean(ySum_withNoise);
```

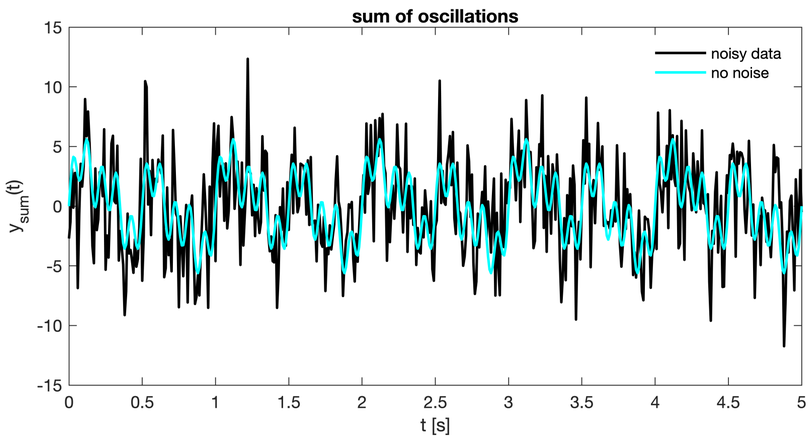
or

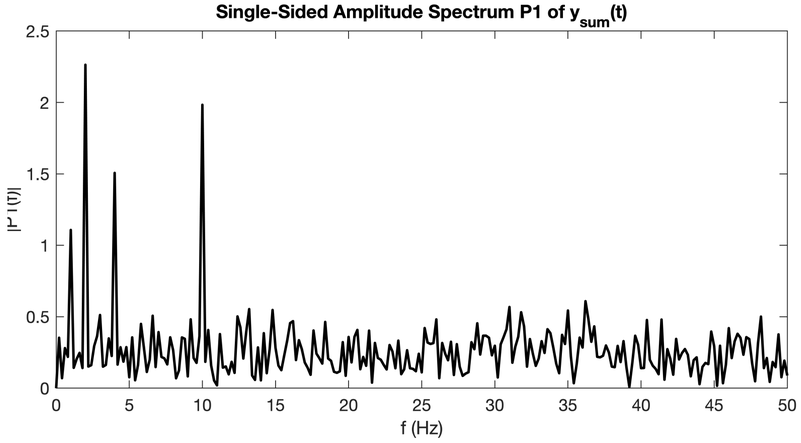
```
ySum_withNoise = detrend(ySum_withNoise);
```

or

```
yFit = fit(t',ySum_withNoise','poly1');
ySum_withNoise = ySum_withNoise-feval(yFit, t)';
```

You can find more about the `fit` function in the MATLAB Help (also check "*List of Library Models for Curve and Surface Fitting*")





Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);
```

```
% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );
```

```
% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );
```

```
% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );
```

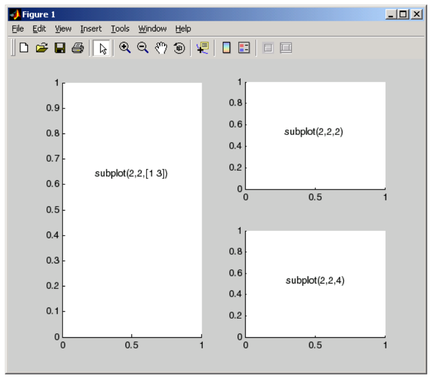
```
% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

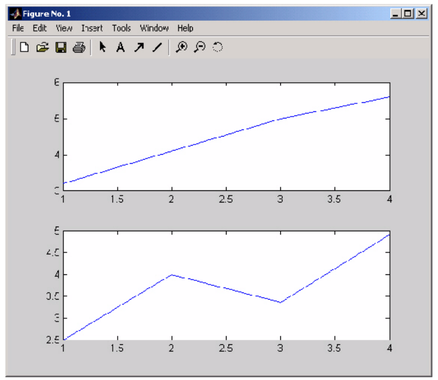
Requiers *Signal Processing Toolbox*

Useful hint: You can plot subfigures into one figure window:

```
figure(1);clf
  subplot(2,1,1)
     plot(…)

  subplot(2,1,2)
     plot(…)
```

`subplot(m, n, No)` plot *m*×*n* subplots within one figure window. With `No` you can address where you want to plot.



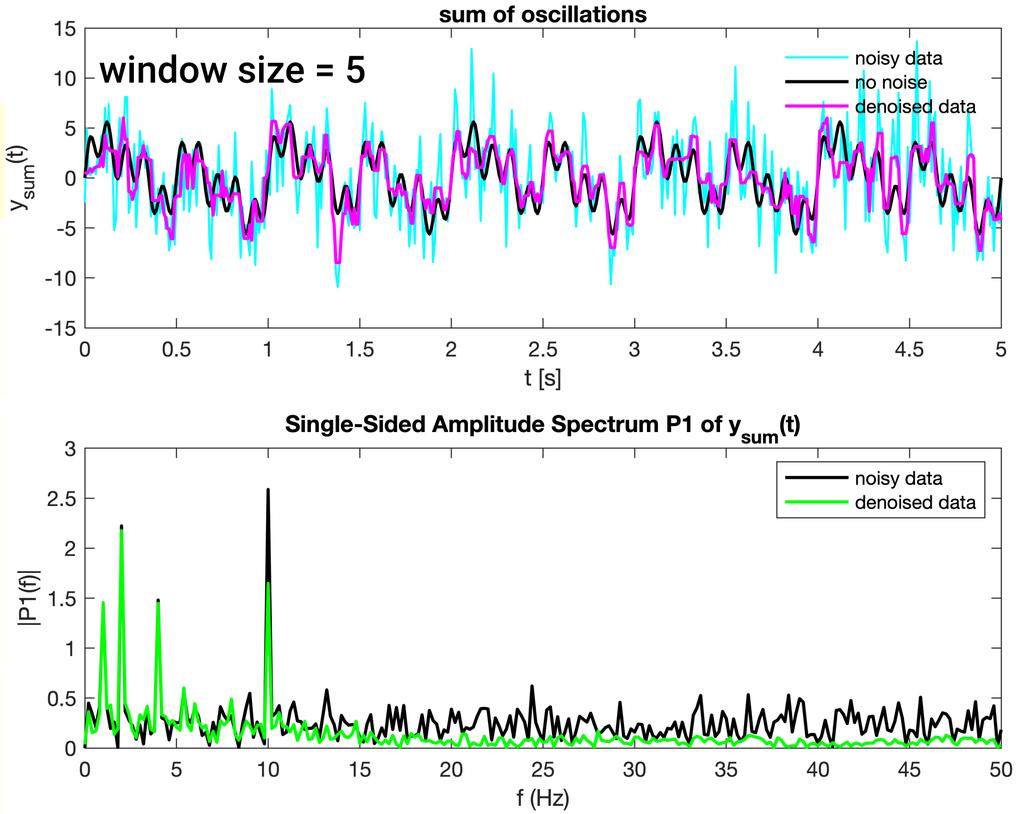


Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

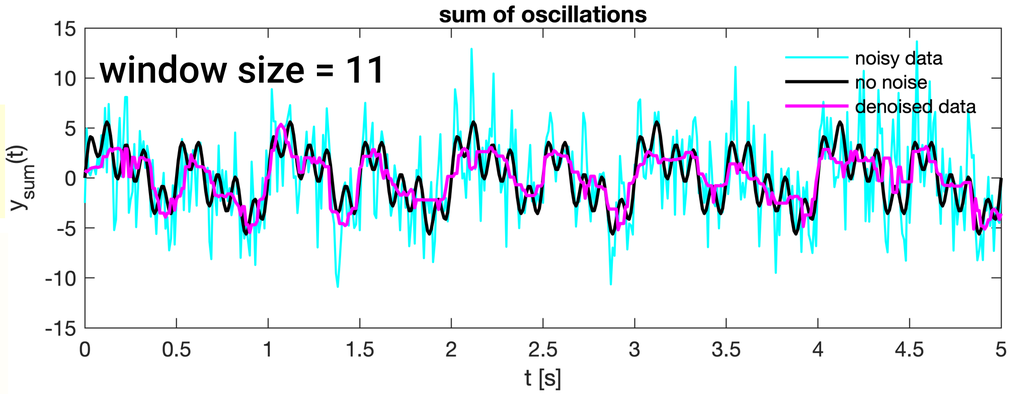
Requiers *Signal Processing Toolbox*

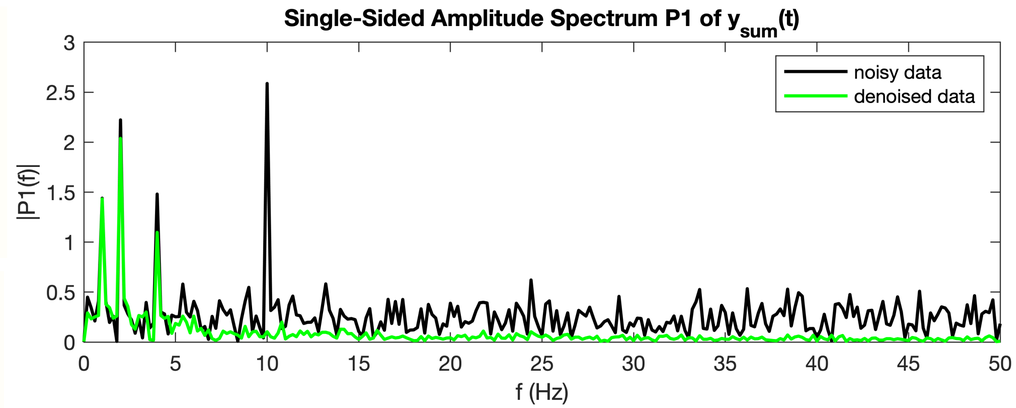


Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```





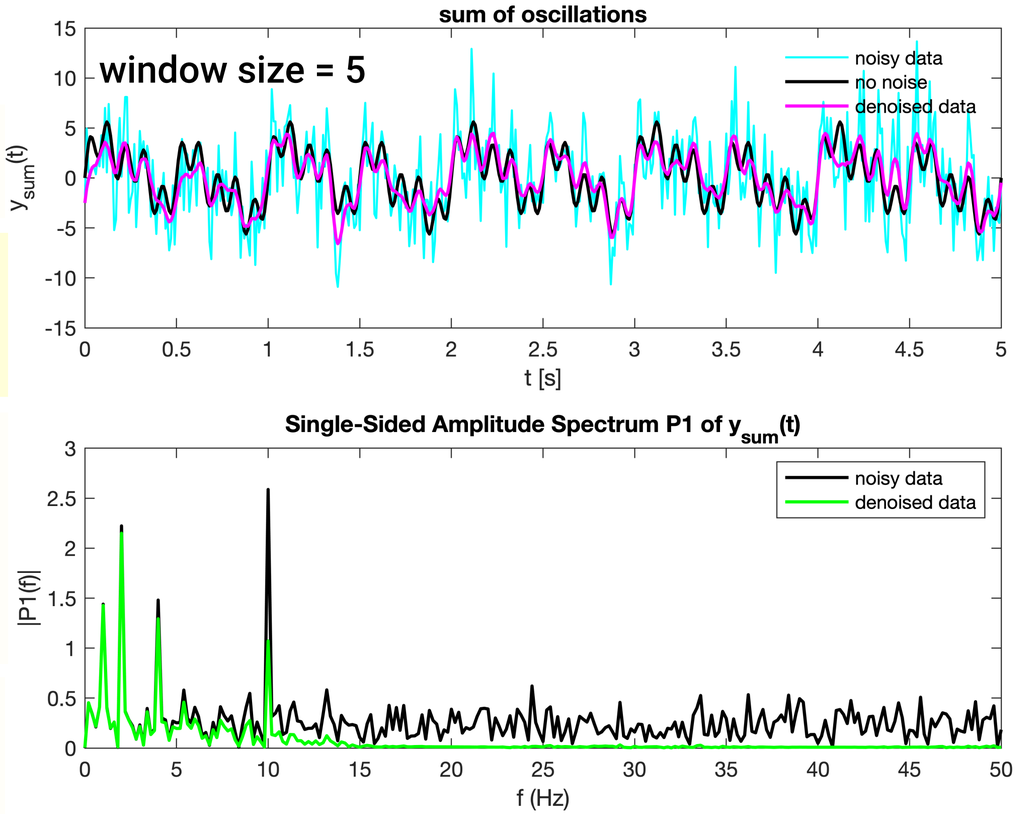
Requiers *Signal Processing Toolbox*

Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

Requiers *Signal Processing Toolbox*

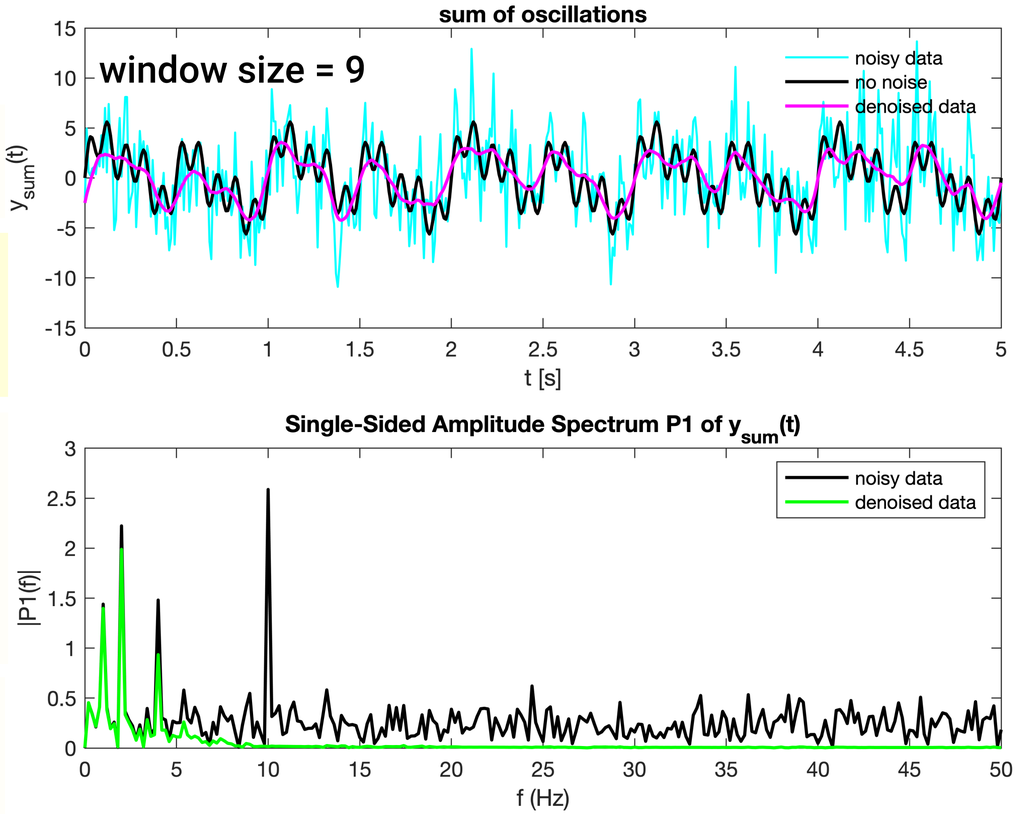


Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

Requiers *Signal Processing Toolbox*

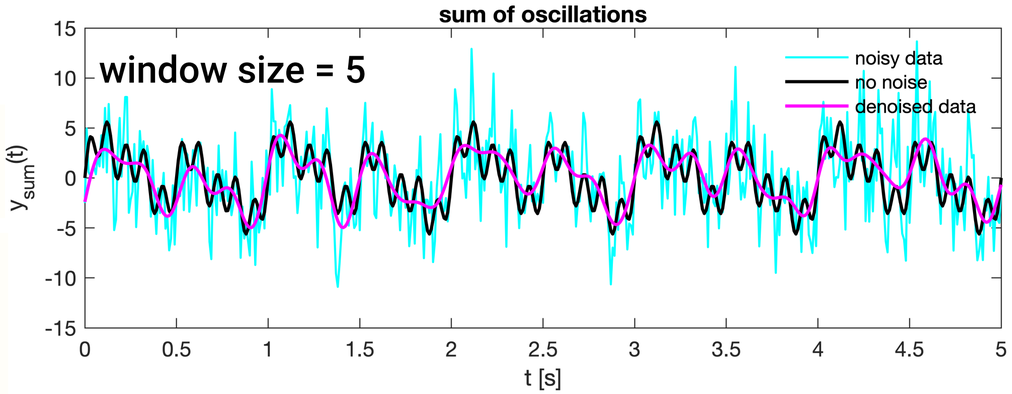


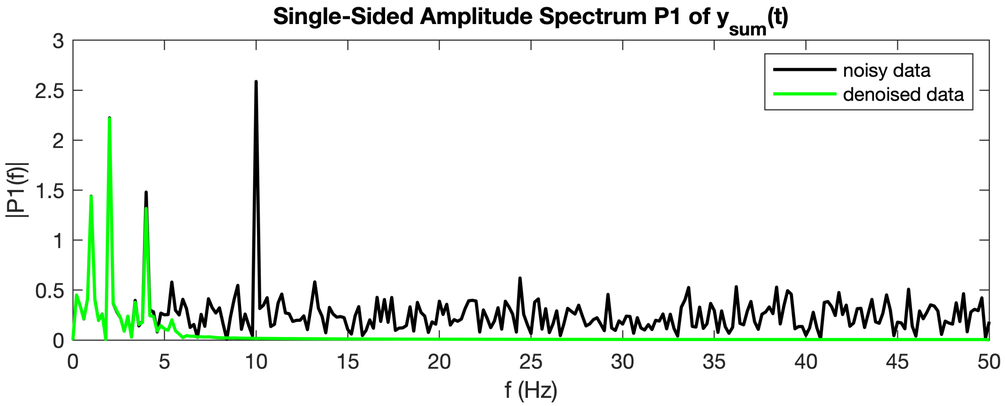
Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

Requiers *Signal Processing Toolbox*



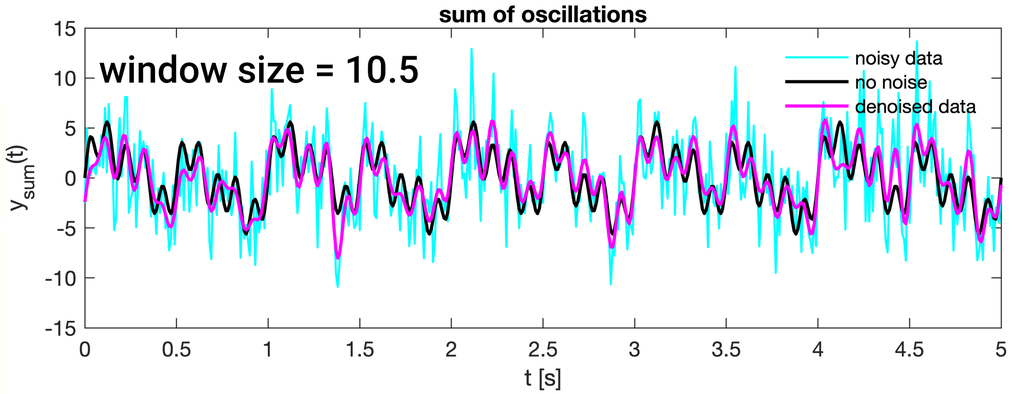


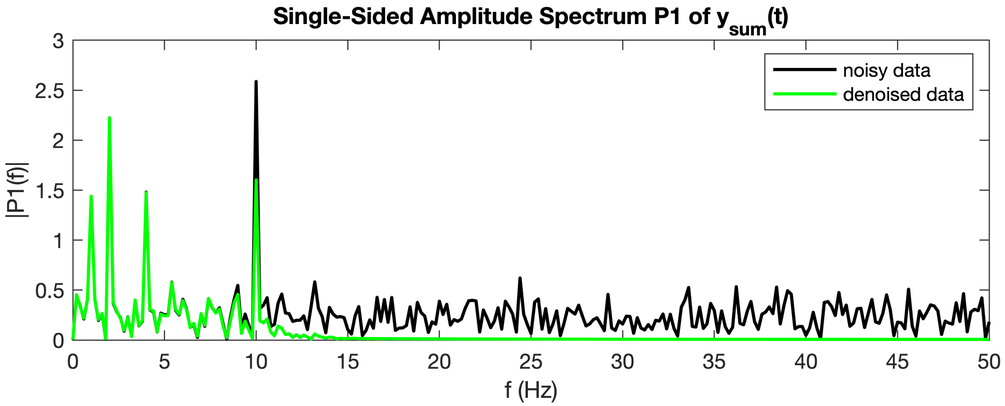
Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

Requiers *Signal Processing Toolbox*

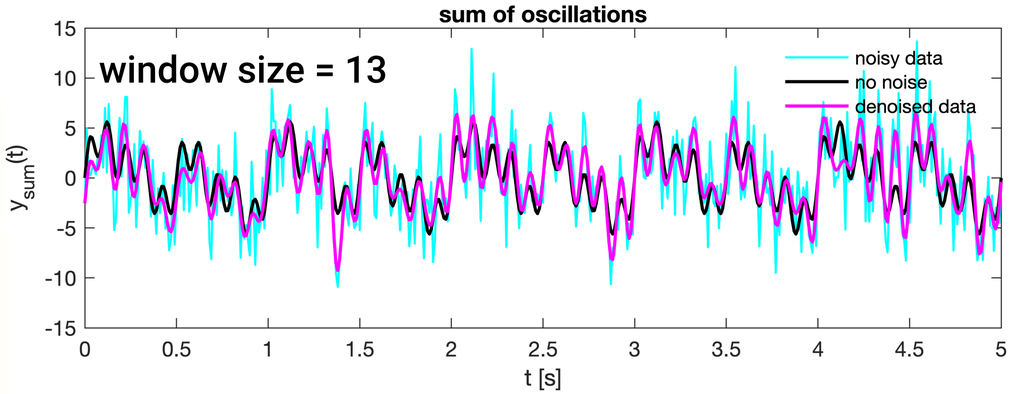


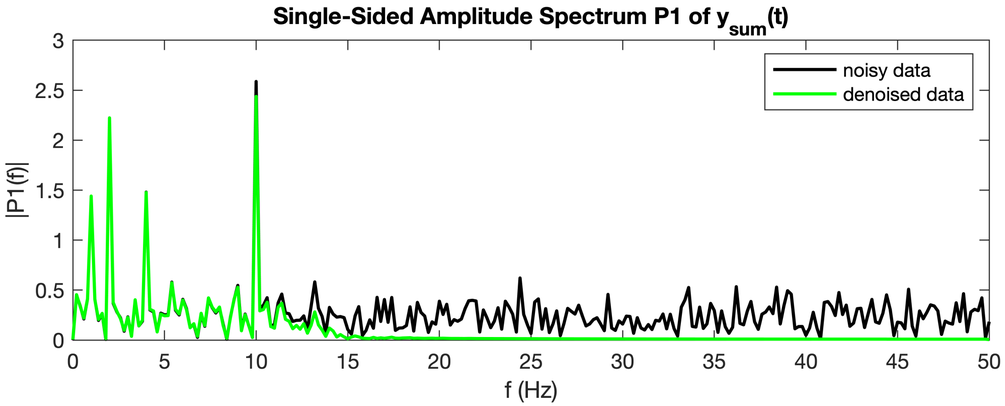


Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );

% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```





Requiers *Signal Processing Toolbox*

Basic Steps of Time Series Analysis: Denoising

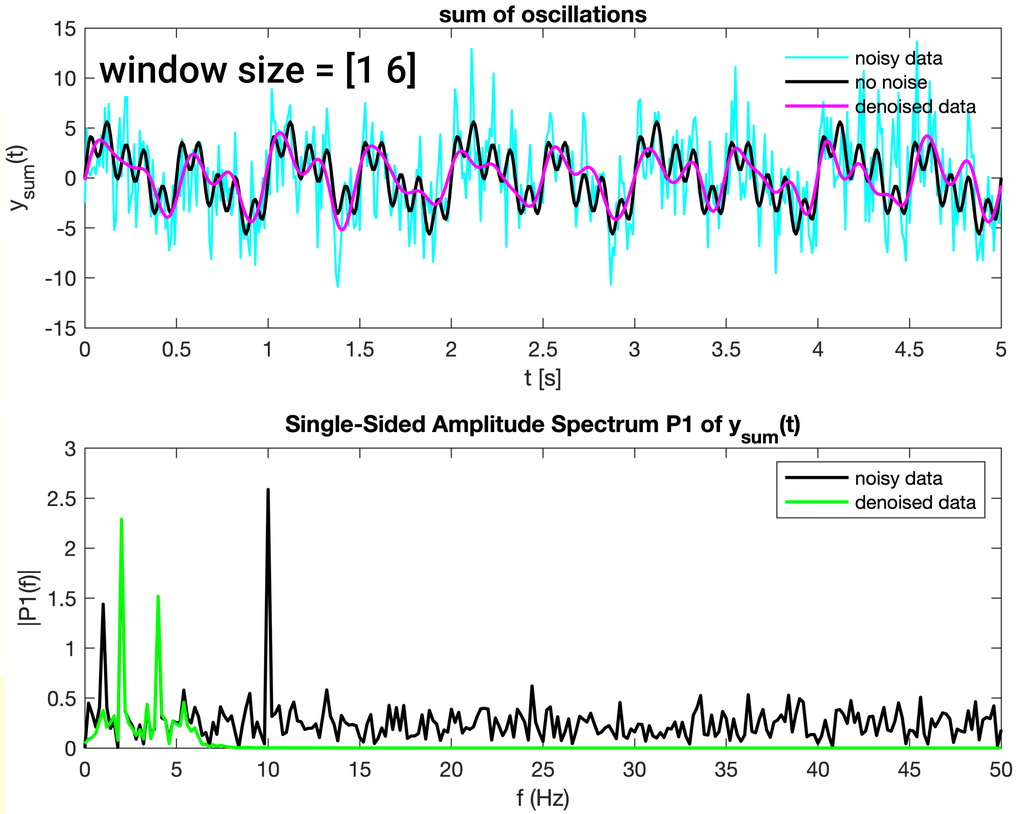
j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );
```

```
% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );
```

```
% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

Requiers *Signal Processing Toolbox*



Basic Steps of Time Series Analysis: Denoising

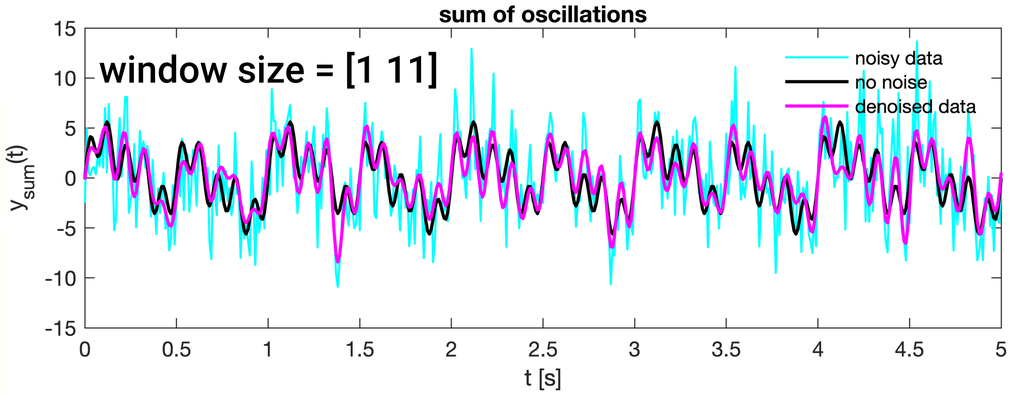
j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:

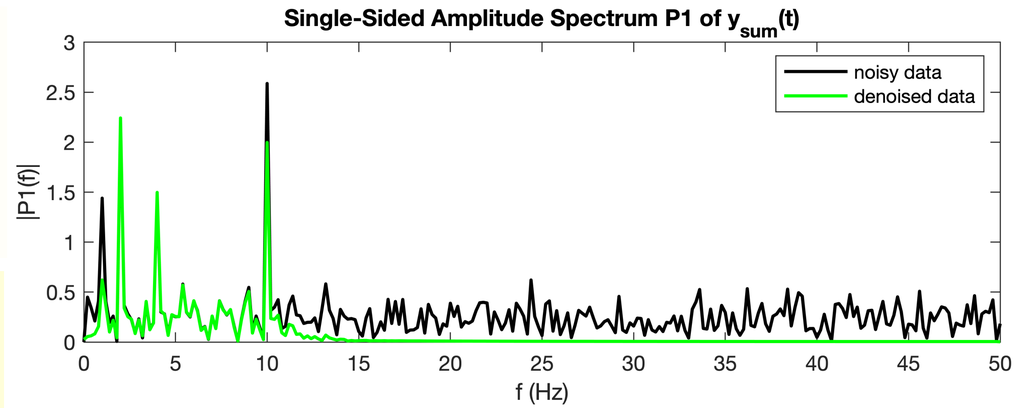
```
% Median-Filter via smoothdata + movmedian:
  ySum_Cleaned = smoothdata(ySum_withNoise,'movmedian',5);

% Moving average filter:
  mAvrgWindowSize = 9;
  MovingAvrgCoeff = ones(1, mAvrgWindowSize)/mAvrgWindowSize;
  ySum_Cleaned = filtfilt(MovingAvrgCoeff, 1, ySum_withNoise );

% Butterworth Filter:
  ButterworthCutoffFreq = 5;
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, ,low');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );
```

```
% Butterworth Filter:
  ButterworthCutoffFreq = [1 6];
  ButterworthOrder      = 5;
  fNorm = ButterworthCutoffFreq / (Fs/2);
  [B,A] = butter(ButterworthOrder, fNorm, 'bandpass');
  ySum_Cleaned = filtfilt(B, A, ySum_withNoise );
```

```
% Low-Pass Filter:
  LowPassCutoffFreq = 10;
  ySum_Cleaned = lowpass(ySum_withNoise, LowPassCutoffFreq, Fs);
```

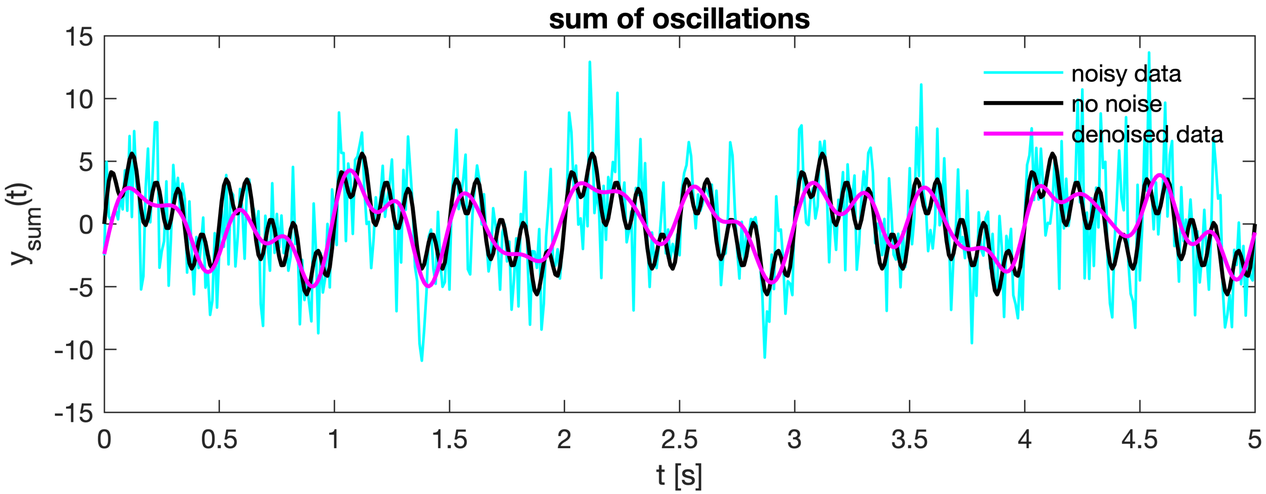
Requiers *Signal Processing Toolbox*

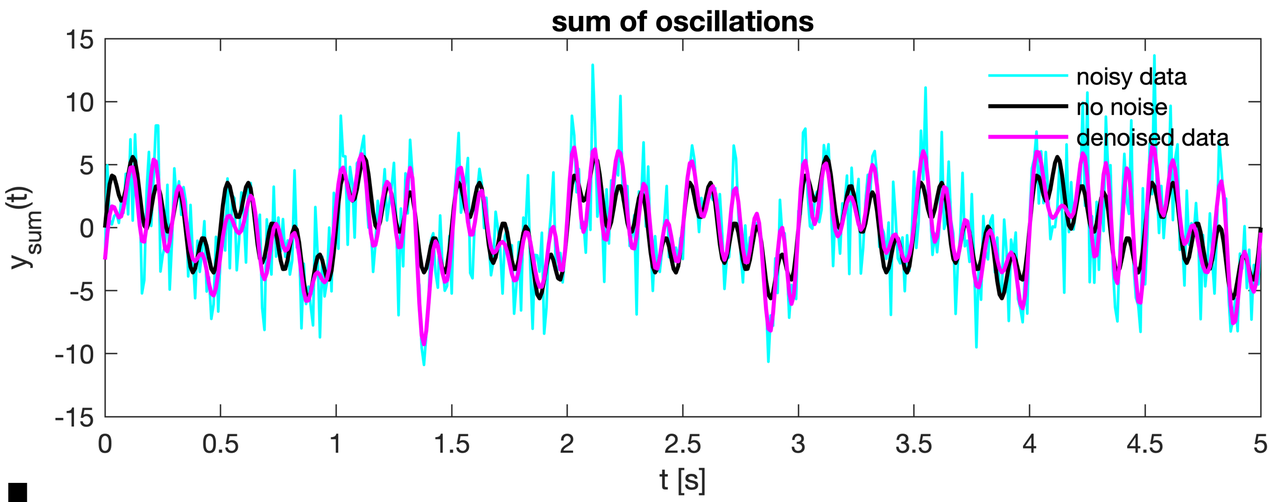




Basic Steps of Time Series Analysis: Denoising

j) Denoise your data and re-run your script. Use one (ore more) of the following tools and vary the filter window sizes:





Depending on what you are interested in, you can remove noise or unwanted frequencies (e.g., 50 Hz power supply voltage) from your data and smooth to your data.

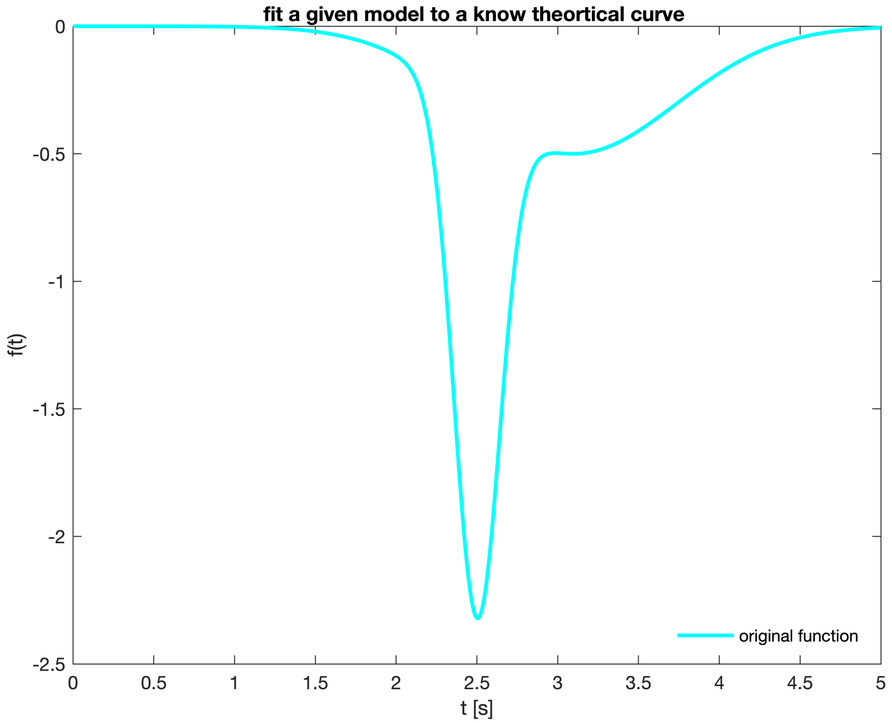
But be careful: If you remove too much, you loose important information content of your data.

Basic Steps of Time Series Analysis: Fitting

Let's define a theoretical curve: $f(t) = a_1 e^{-\left(\frac{t-b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{t-b_2}{c_2}\right)^2}$

```
% Define time-vector:
  dt=0.01;
  t=0:dt:5;        %[s]
% Define coefficients:
  a1=-0.5;
  a2=-2;
  b1=3.1;
  c1=0.9;
  b2=2.5;
  c2=0.2;
% Define function:
  yGauss = a1*exp(-( (t-b1)/(c1) ).^2) +  a2*exp(-( (t-b2)/(c2) ).^2);
```

```
% Fit the generated curve:
  % define the fittype:
    ft = fittype( 'a1*exp(-((t-b1)/(c1)).^2) +  a2*exp(-((t-b2)/
                  (c2)).^2);', 'independent', 't', 'dependent', 'y' );
  % set fitoptions:
    opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
    opts.Display = ,Off';
                      %  a1 a2  b1  b2  c1  c2      initial-guess-
    opts.StartPoint = [-0.5 -2 0.4 0.9 0.3 0.45]; % values

  % Fit model to data:
    [yFit, gof] = fit( t', yGauss', ft, opts );
```



Basic Steps of Time Series Analysis: Fitting

Let's define a theoretical curve: $f(t) = a_1 e^{-\left(\frac{t-b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{t-b_2}{c_2}\right)^2}$

Here we define the fit-function. In our case, we know the exact model-function. But this is often unknown in real data. There are several other built-in fit-functions (e.g., Gaussians, Polynomials, Sinus-curves, …).

```
% Fit the generated curve:
  % define the fittype:
    ft = fittype( 'a1*exp(-((t-b1)/(c1)).^2) +  a2*exp(-((t-b2)/
                  (c2)).^2);', 'independent', 't', 'dependent', 'y' );
  % set fitoptions:
    opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
    opts.Display = ,Off';
                       %  a1 a2  b1  b2  c1  c2      initial-guess-
    opts.StartPoint = [-0.5 -2 0.4 0.9 0.3 0.45]; % values

  % Fit model to data:
    [yFit, gof] = fit( t', yGauss', ft, opts );
```

fit-algorithm starts with these values and performs several iterations until no further significant *changes* can be achieved (convergence) or a maximum iteration number is reached. Can be left empty, if unknown, but it's often good to set them (they constrain and speed up the fit).

fit object (our results). Contains the fitted coefficients, which can be evaluated by `feval(yFit, t)`.

Goodness of the fit, contains among other the residuals, i.e., data-fit-model = *changes*.

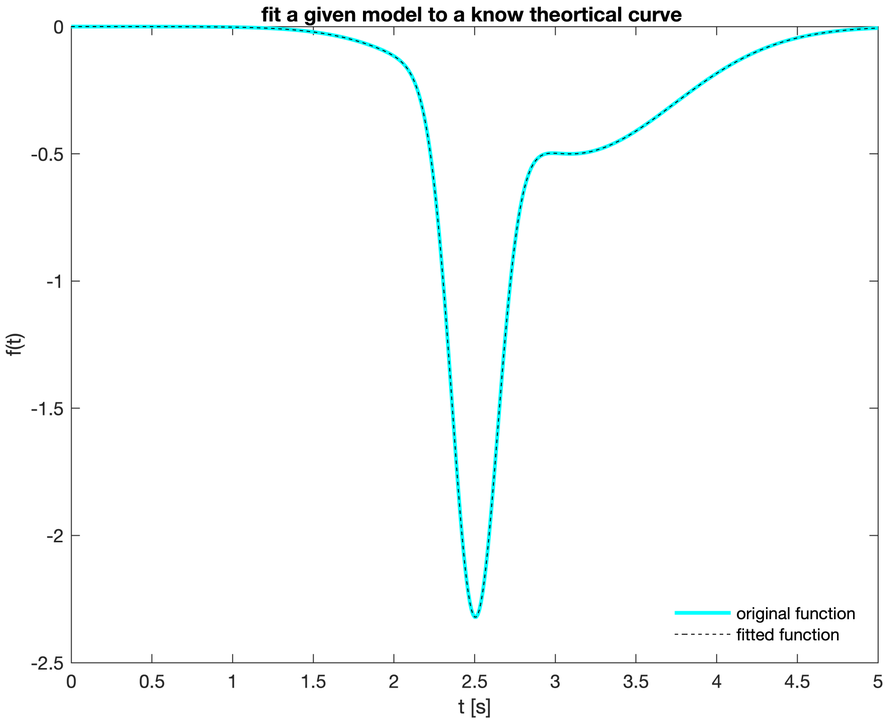
```
General model:
yFit(t) = a1*exp(-( (t-b1)/(c1) ).^2) +  a2*exp(-( (t-b2)/(c2) ).^2);
Coefficients (with 95% confidence bounds):
  a1 =     -0.4845  (-0.4954, -0.4736)
  a2 =      -1.943  (-1.966, -1.92)
  b1 =       3.098  (3.076, 3.119)
  b2 =       2.503  (2.501, 2.505)
  c1 =      0.9341  (0.9116, 0.9566)
  c2 =      0.2041  (0.2011, 0.2071)
```

Basic Steps of Time Series Analysis: Fitting

Let's define a theoretical curve: $f(t) = a_1 e^{-\left(\frac{t-b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{t-b_2}{c_2}\right)^2}$

```
% Define time-vector:
  dt=0.01;
  t=0:dt:5;        %[s]
% Define coefficients:
  a1=-0.5;
  a2=-2;
  b1=3.1;
  c1=0.9;
  b2=2.5;
  c2=0.2;
% Define function:
  yGauss = a1*exp(-( (t-b1)/(c1) ).^2) +  a2*exp(-( (t-b2)/(c2) ).^2);
```

```
% Fit the generated curve:
  % define the fittype:
    ft = fittype( 'a1*exp(-((t-b1)/(c1)).^2) +  a2*exp(-((t-b2)/
                  (c2)).^2);', 'independent', 't', 'dependent', 'y' );
  % set fitoptions:
    opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
    opts.Display = ,Off';
                      %  a1 a2  b1  b2  c1  c2       initial-guess-
    opts.StartPoint = [-0.5 -2 0.4 0.9 0.3 0.45]; % values

  % Fit model to data:
    [yFit, gof] = fit( t', yGauss', ft, opts );
```



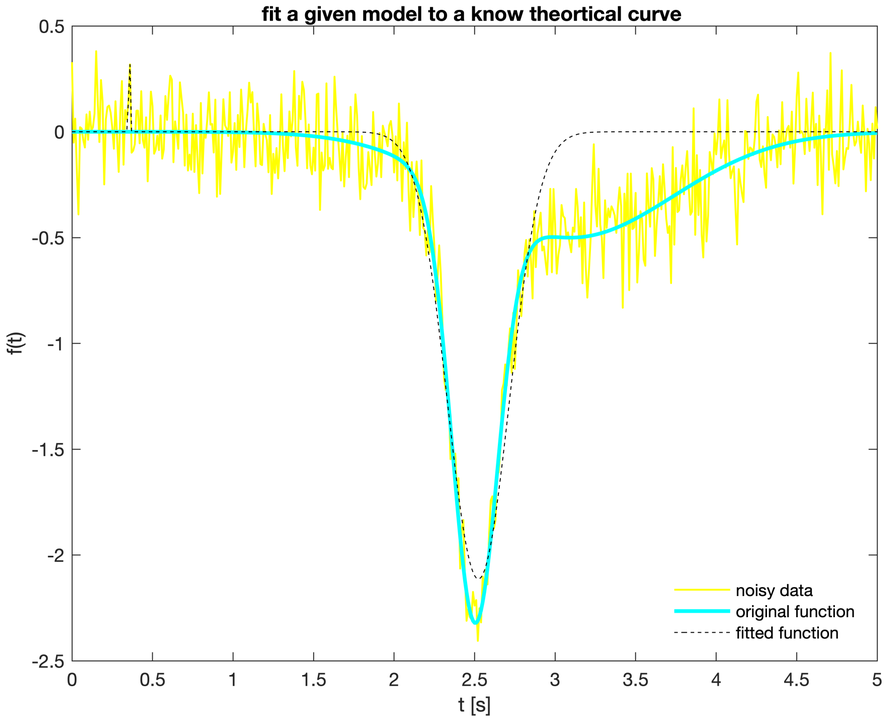
perfectly fitted!

Basic Steps of Time Series Analysis: Fitting

Let's define a theoretical curve: $f(t) = a_1 e^{-\left(\frac{t-b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{t-b_2}{c_2}\right)^2}$

```
% Define time-vector:
  dt=0.01;
  t=0:dt:5;        %[s]
% Define coefficients:
  a1=-0.5;
  a2=-2;
  b1=3.1;
  c1=0.9;
  b2=2.5;
  c2=0.2;
% Define function:
  yGauss = a1*exp(-( (t-b1)/(c1) ).^2) +  a2*exp(-( (t-b2)/(c2) ).^2);
```

```
% Fit the generated curve:
  % define the fittype:
    ft = fittype( 'a1*exp(-((t-b1)/(c1)).^2) +  a2*exp(-((t-b2)/
                  (c2)).^2);', 'independent', 't', 'dependent', 'y' );
  % set fitoptions:
    opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
    opts.Display = ,Off';
                      %  a1 a2  b1  b2  c1  c2       initial-guess-
    opts.StartPoint = [-0.5 -2 0.4 0.9 0.3 0.45]; % values

  % Fit model to data:
    [yFit, gof] = fit( t', yGauss', ft, opts );
```



~~perfectly fitted!~~

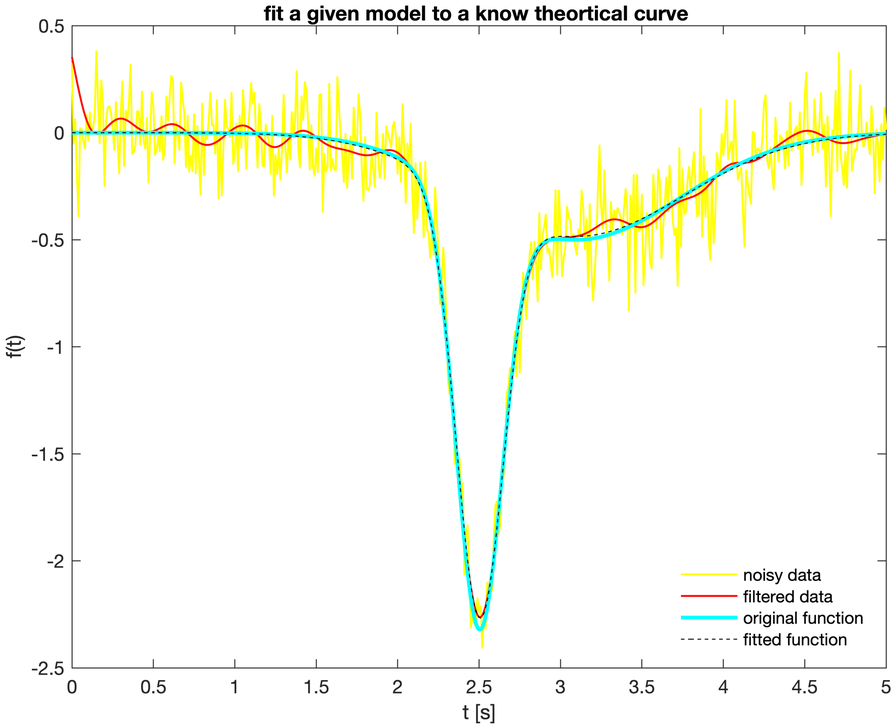
add some noise… fit doesn't perform well.

Basic Steps of Time Series Analysis: Fitting

Let's define a theoretical curve: $f(t) = a_1 e^{-\left(\frac{t-b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{t-b_2}{c_2}\right)^2}$

```
% Define time-vector:
  dt=0.01;
  t=0:dt:5;        %[s]
% Define coefficients:
  a1=-0.5;
  a2=-2;
  b1=3.1;
  c1=0.9;
  b2=2.5;
  c2=0.2;
% Define function:
  yGauss = a1*exp(-( (t-b1)/(c1) ).^2) +  a2*exp(-( (t-b2)/(c2) ).^2);
```

```
% Fit the generated curve:
  % define the fittype:
    ft = fittype( 'a1*exp(-((t-b1)/(c1)).^2) +  a2*exp(-((t-b2)/
                  (c2)).^2);', 'independent', 't', 'dependent', 'y' );
  % set fitoptions:
    opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
    opts.Display = ,Off';
                      %  a1 a2  b1  b2  c1  c2       initial-guess-
    opts.StartPoint = [-0.5 -2 0.4 0.9 0.3 0.45]; % values

  % Fit model to data:
    [yFit, gof] = fit( t', yGauss', ft, opts );
```



~~perfectly fitted!~~

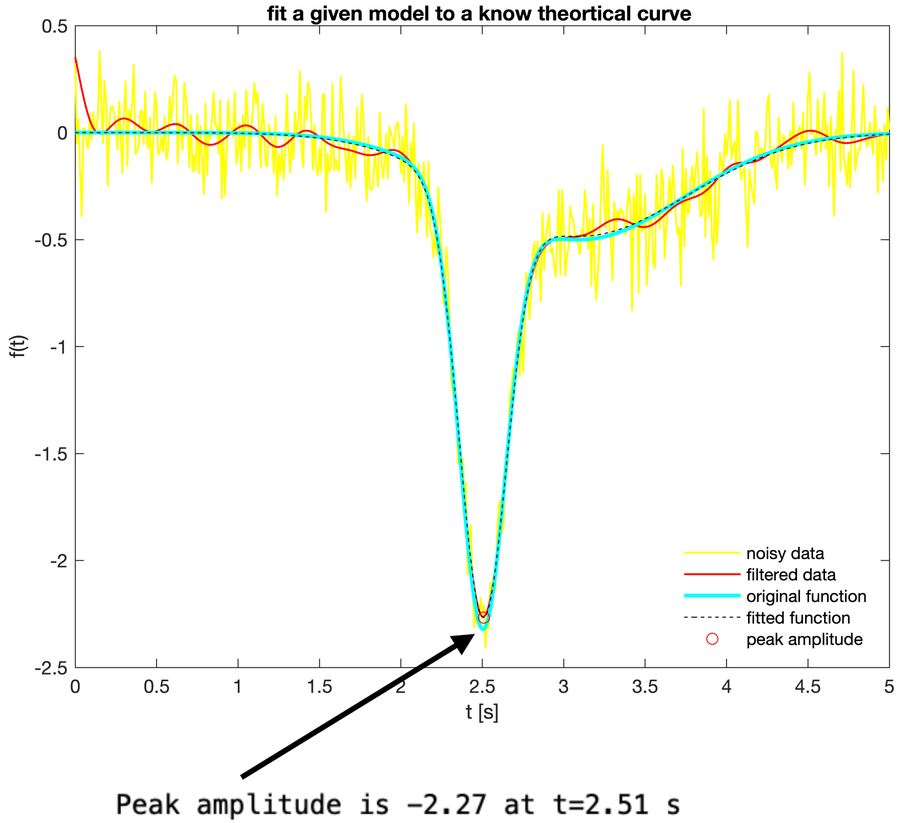
add some noise… fit doesn't perform well.

denoise (e.g., butterworth filter)…

fit is okay.

Basic Steps of Time Series Analysis: Fitting

What is a fit good for?



Proof of principle: Test, how good your theory (model-function) explains the measurements.

$$f(t) = a_1 e^{-\left(\frac{t-b_1}{c_1}\right)^2} + a_2 e^{-\left(\frac{t-b_2}{c_2}\right)^2}$$

Assessment and feature extraction of your data: Noise, data gaps and random disturbances often cause large deviations from the ideal model. Sometimes, the ideal model is even unknown. In these cases you fit an arbitrary fit-function (start simple!) in order to assess your data, e.g. to estimate the peak amplitude of a signal.

```
yFitEval = feval(yFit, t);
yPeakIdx = find(yFitEval==min(yFitEval)); % Find peak-index
yPeak    = yFitEval(yPeakIdx);
tPeak    = t(yPeakIdx);
fprintf(1, 'Peak amplitude is %2.2f at t=%2.2f s\n', yPeak, tPeak)
```

Let’s work with real data: Data I/O (import data)

There are several ways, to import external data files into MATLAB:

```
IN = importdata(filename);
load(filename);
IN = textscan(fileID,formatSpec);
IN = readtable(filename);
IN = xlsread(filename);
```

We will use this function, makes reading Excel files and any .txt, .dat or .csv file with delimited text very easy.

	A	B
1	time	I
2	0,5	3,97E-12
3	0,50001	3,91E-12
4	0,50002	3,78E-12

`IN=readtable;` imports the data into a so-called *structure variable* (actually, a very powerful tool!) and names the structure fields according to the table header, e.g., `IN.time` and `IN.I`.

Generally supported file formats for import and export:

- **MATLAB formatted data** (mat)
- **Text** (txt, csv, dat)
- **Spreadsheet** (xls, xlsx, ods)
- **Extensible Markup Language** (xml)
- **Scientific data** (cdf, fits, hdf, h5, nc)
- **Image** (png, tif, gif, jpg, bmp, …)
- **Audio** (au, aiff, flac, wav, mp4, ...)
- **Video** (mpg, mov, avi…)

Some words on the filename and path:

absolute and relative file path

```
IN = readtable('PatchClampData.xlsx');
```

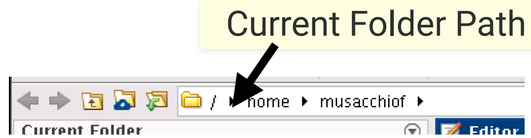
Looks for the specified file in the current folder.

```
IN = readtable('Data/PatchClampData.xlsx');
```

Looks for the specified file in the folder "Data" which lies inside the current folder.

```
IN = readtable('/Users/user-adm/Science/Matlab/PatchClampData.xlsx');
```

Looks for the specified file in the folder "Matlab" preceded by the fullly specified path starting from the root folder of your computer (/…).



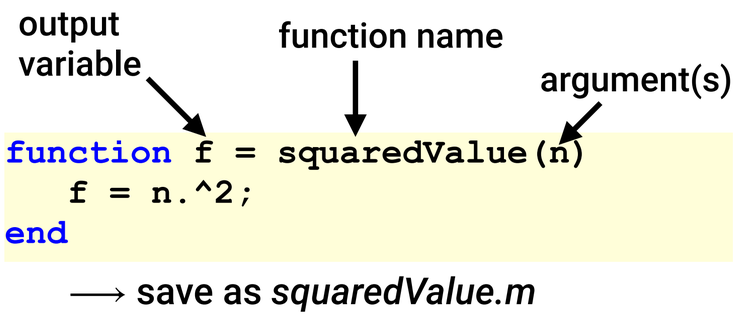
MATLAB functions

Before we import our first data file, we need to learn the concept of a special type of the script: the *function*.

***functions* are saved similar to scripts as m-files. But: the function's name and its file name must be identical!**

***functions* a like encapsulated or sandboxed scripts: Only parameters and constants defined within the function or externally given by input-arguments are known by the function. The function has no access to variables of the current workspace.**
***functions* allow to re-use program steps, which are repeated or applied several times, e.g., inside a for-loop. They can serve as default definitions which can be re-used also in other programs.**

***functions* can also be defined inside a script (but it's not recommended as it quickly becomes confusing and makes debugging difficult).**



Put the denoising procedure using the butterworth filter into a function `fClean= denoise(x,y,CutoffFreq)`. Name you function "*denoise.m*".

```
Fs =numel(x)/x(end);
T = 1/Fs;
L = numel(x);
ButterworthCutoffFreq = CutoffFreq;
ButterworthOrder      = 5;
fNorm = ButterworthCutoffFreq / (Fs/2);
[B,A] = butter(ButterworthOrder, fNorm,
'low'); % 'low'
b_Cleaned = filtfilt(B, A, y);
```

Let's work with real data: Data I/O

a) Download the data file *PatchClampData.xlsx*.
b) Write a new script and import the file. Plot `IN.time` against `IN.I`.
c) Apply your function `denoise(x,y,CutoffFreq)`. Use a `CutoffFreq` of 10 Hz. Plot the result in the same figure.
d) The current baseline should be around 0 pA. Do a rough detrending by pushing the entire denoised signal below 0 pA (`currentClean = currentClean-max(currentClean);`).
e) Fit the denoised data to
   - a sum of Gaussian functions (vary between *gauss2* and *gauss8*)
   - a sum of Fourier series (vary between *fourier2* and *fourier8*)
   - a smoothingspline interpolation (with a smoothing parameter of 0.3, 0.5 and 0.7).

   and plot the result. Calculate the computing time. What do you notice?

```
current = IN.I;      % PatchClamp current [pA]
time    = IN.time;   % time [ms]
```

```
figure(10);clf
   hold on
   plot(time, current, '-c' )
   plot(time, currentClean, '-k', 'LineWidth',2 )
   plot(time, feval(yFit, time), '--m' , 'LineWidth',2 )
```

```
ticTimeElapseStart = tic;
```

```
 % Definitions for Gaussian-fit:
   ft = fittype( 'gauss8' );
   opts = fitoptions('Method', 'NonlinearLeastSquares' );
   opts.Display = 'Off';
 % Fit model to data:
   [yFit, gof] = fit( time, currentClean, ft, opts );
```

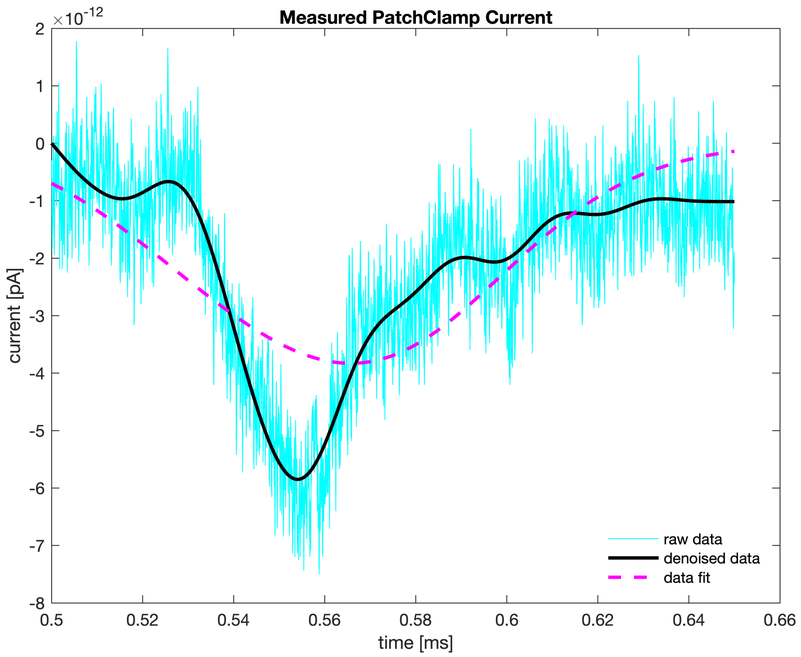
```
% Definitions for Fouerier series-fit:
   ft = fittype( 'fourier4' );
   opts = fitoptions('Method', 'NonlinearLeastSquares' );
   opts.Display = 'Off';
 % Fit model to data:
   [yFit, gof] = fit( time, currentClean, ft, opts );
```

```
% Definitions for Smoothspline-fit:
   ft = fittype( 'smoothingspline' );
   opts = fitoptions('Method', 'SmoothingSpline' );
   opts.Normalize = 'on';
   opts.SmoothingParam = 0.7;
% Fit model to data:
   [yFit, gof] = fit( time, currentClean, ft, opts );
```

```
ticTimeElapsed = toc(ticTimeElapseStart);
fprintf(1,'CPU time %8.2f s)\n', ticTimeElapsed);
```

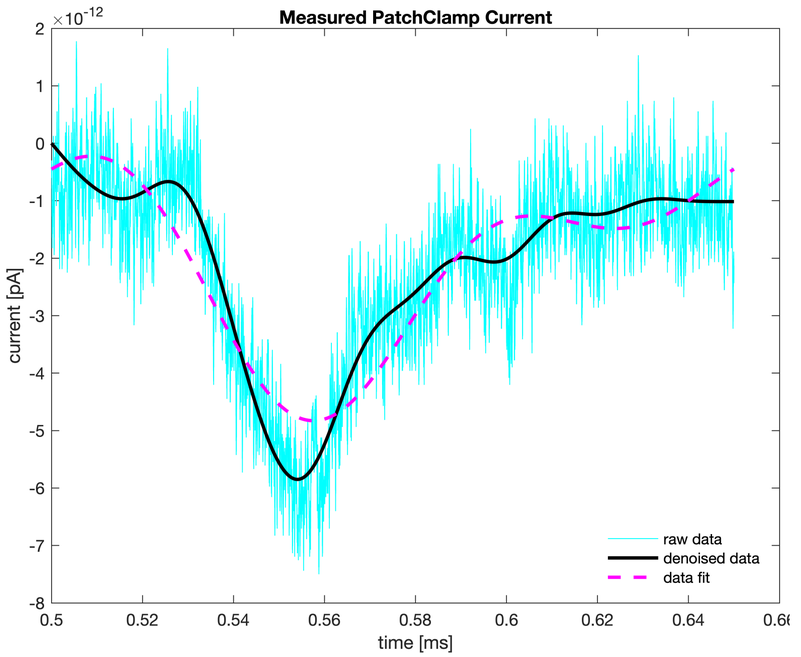
Let’s work with real data: Data I/O

gauss3



rsquare: 0.636032820797316

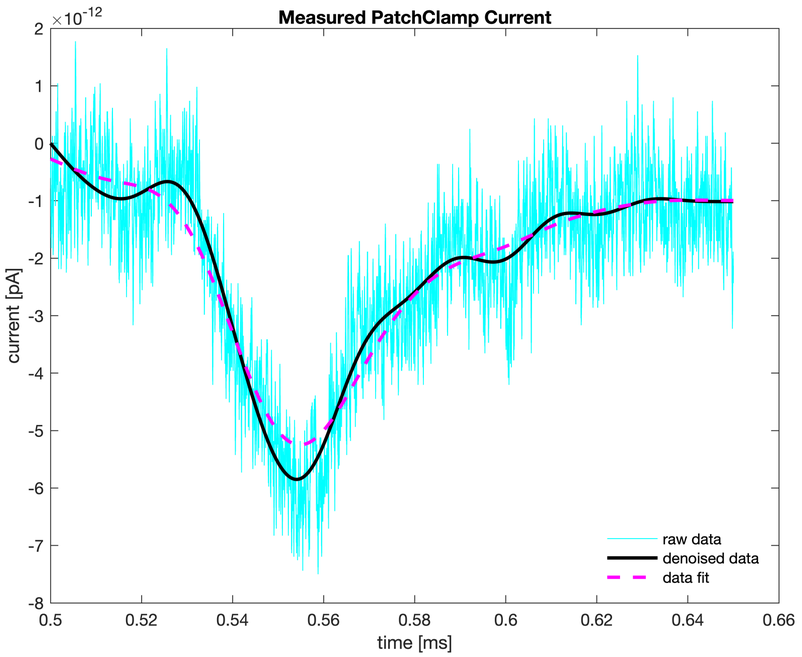
fourier2



rsquare: 0.880926713820189

smoothspline

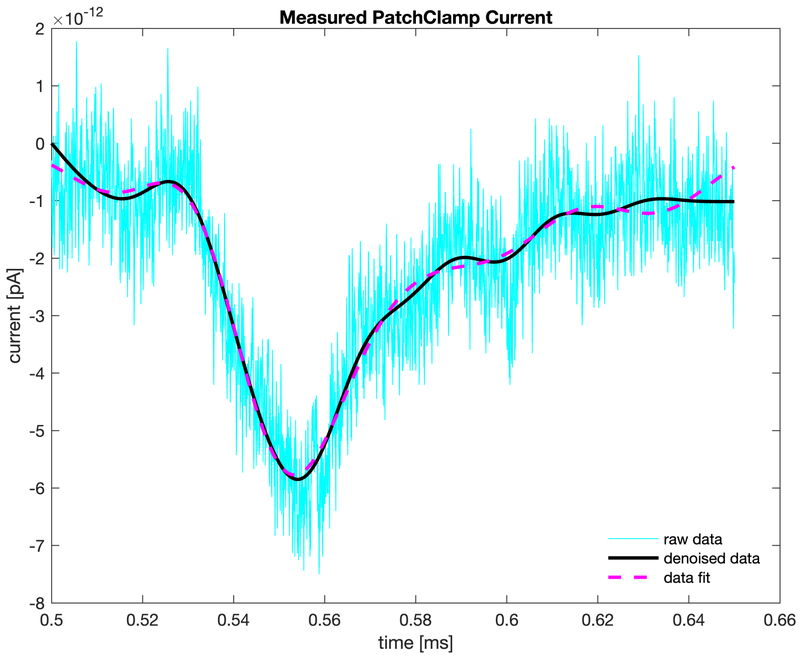
smoothing parameter of 0.3



rsquare: 0.973538103413017

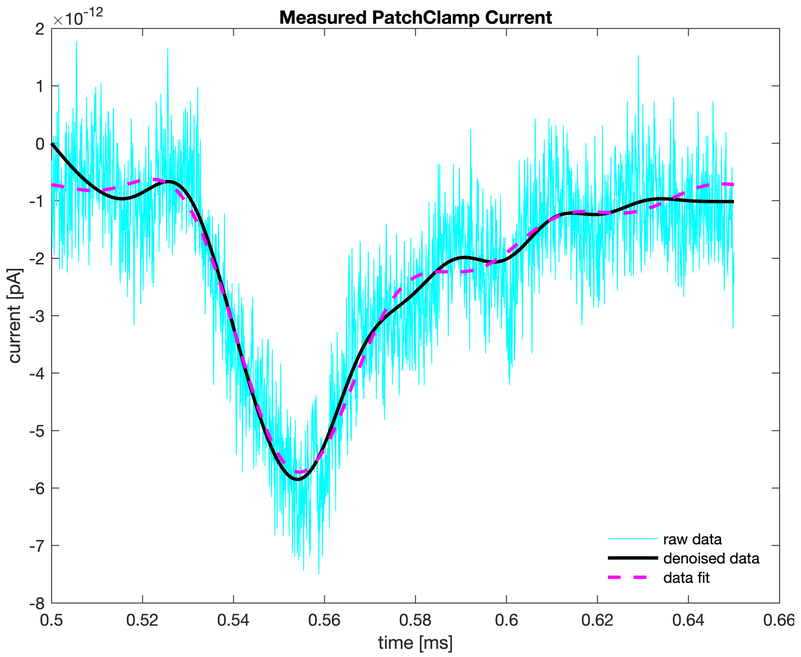
Let’s work with real data: Data I/O

gauss8



rsquare: 0.996197554495393

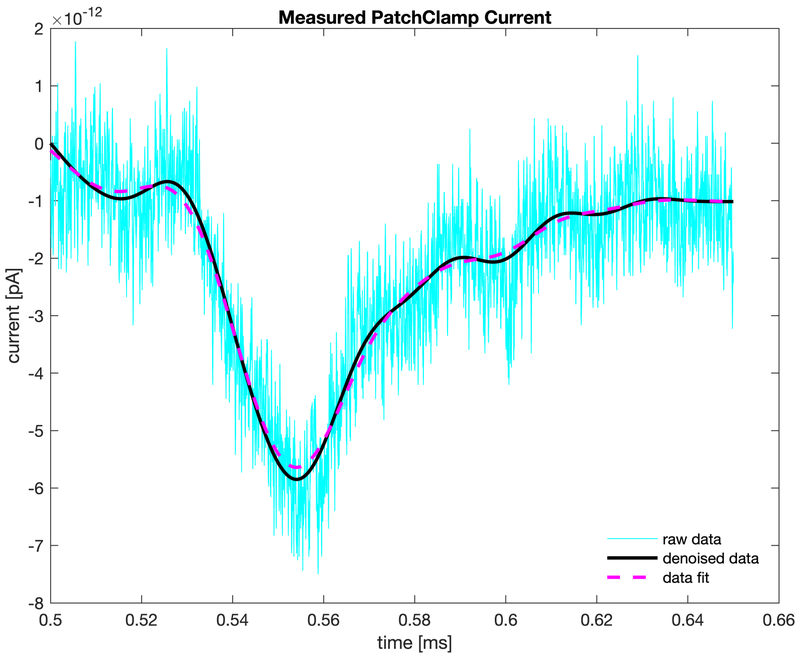
fourier4



rsquare: 0.991729471450458

smoothspline

smoothing parameter of 0.7



rsquare: 0.986308858001857

faster → | ← ***slower***

Let’s work with real data: Data I/O (export data)

There are several ways to save your results out of MATLAB.

Here are two exporting procedures:

```
% Create and open txt-File:
  filename = 'resultsLog.txt';
  fidLog = fopen(filename, 'w', 'n', 'UTF-8');
  if fidLog>0
      fprintf(1, 'Created txt file (%s)\n', filename)
  end
% Write data into file
  fprintf(fidLog, 'Hello World.\n My results:\n');
  for i=1:numel(time)
      fprintf(fidLog, '%3.10f\t %e\n', time(i), ...
              currentClean(i));
  end
% Close file (important!):
  fclose(fidLog);
```

we discuss “if“ very soon

linebreak for commands

```
% Write xlsx-File using table + writetable:
  filename = 'resultsLog.xlsx';
  TableHeader=[{'time'} {'current'}];
  TableArray =table(time, currentClean, ...
                'VariableNames',TableHeader);
  writetable(TableArray,filename);
```

f) Save the denoised data into a new file *PatchClampData_processed.xlsx*. Use the `table` and `writetable` functions.

The *if*-Statement, logical and relation operators

With an *if*-statement you can check, if a condition (or more) are fulfilled or not.
If so, do something. If not, do something else.

```
a=1;
b=2;
if a==1
   fprintf(1, 'Match! a=%d\n', a)
else
   fprintf(1, 'No match :( a=%d\n', a)
end
```

```
if a==1 && b==2
   fprintf(1, 'Match! a=%d and b=%D\n', a, b)
elseif a==1
   fprintf(1, 'Partially matched: a=%d\n', a)
elseif b==1
   fprintf(1, 'Partially matched: a=%d\n', a)
else
   fprintf(1, 'No match :('')
end
```

```
if a==1 || b==2
   fprintf(1, 'Match for a or b!'')
end
```

Relation operators:

>	greater than
<	lower than
>=	greater than or equal
<=	lower than or equal
==	equal
~=	not equal

Logical operators:

&&, &	AND
‖, |	OR
~	NOT

Works also with `find()`:

```
v = 1:10;
index1 = find(v==3);
index2 = find(v>=3);
index3 = find(v>=3 && v<7);
```

and:

```
v(v>5)
v(v>=9) = pi;
```

Let’s do more data!

In our daily work, we are usually confronted with more than just one measurement. Let’s see how to handle a recording with several traces:

	A	B	C	D	E	F
1	time	trace01	trace02	trace03	trace04	trace05
2	0	1,021305918	1,002996344	1,004357628	0,960394892	0,9318
3	0	0,916154434	0,950719315	1,030025877	0,960514786	0,9875
4	0	1,04343296	1,043892715	0,916365592	1,081664231	0,9808
5	0	0,95630638	1,056176559	0,921257353	1,063875568	1,123
6	0	1,067446101	0,984098756	1,02038097	1,068340909	1,0261
7	0,1	1,072434286	1,079493505	1,101252549	1,00582834	1,0002
8	0,1	1,060863821	0,996332715	1,050414451	0,950326924	0,9853

The file *PeakTestData.xlsx* contains *N*=20 recorded traces with *M*=1001 data points. The first column is the time array. Therefore, the table has in total 21 columns.

When we import this file with `IN=readtable('PeakTestData.xlsx')`, MATLAB will load all columns into the structure array `IN.time`, `IN.trace1`, `IN.trace2`, `IN.trace3`, … . The time-column can easily be stored into a new array (e.g., `time=IN.time`). To store the remaining columns, i.e., the traces, into a new array (here: 1001×20), we can use `table2array`:

```
traces= table2array(IN(:,2:end));
```

Why?? ⟶ We want to access the traces inside a *for*-loop and therefore we need the single traces stored and accessible in an appropriate manner:

```
tracesN = size(traces,2);
for i=1:tracesN
   plot(time, traces(:,i) , '-c' )
end
```

Let’s do more data!

a) Open a new script and import *PeakTestData.xlsx* with `readtable`.
b) Prepare the data in the manner as described on the previous slide.
c) Plot all traces into one figure. What do you notice?
d) Denoise the data with your denoise-function.
e) Fit each trace with a Gaussian function (*gauss2*) and plot your result.
f) Determine the highest peak (amplitude and peak time) of each trace and mark its position.
g) Use an *if*-statement in order to distinguish between early (<6 s) and late peaks (≥6s). Re-plot traces with early peaks and late peaks in two different colors.
h) Store the peak time and amplitude into the variables `Tearly`, `Tlate`, `Aearly` and `Alate`.

```
ft = fittype( 'gauss2' );
opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
opts.Display = 'Off';
[yFit, gof] = fit( time, tracesCleaned(:,i), ft, opts );
plot(time, feval(yFit, time), '-m' , 'LineWidth',1.25 )
```

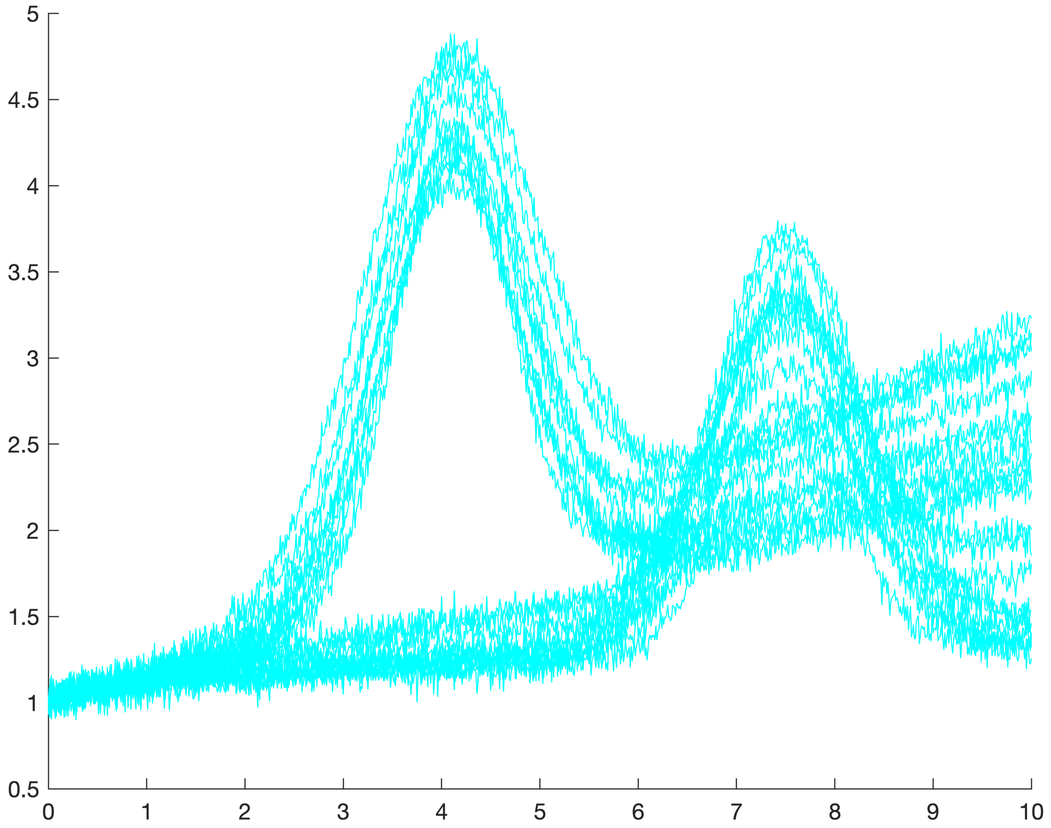
```
tracesCleaned(:,i) = denoise(time, traces(:,i), 5);
```

```
yFitEval = feval(yFit, time);
idxMax = find(yFitEval==max(yFitEval));
```

```
Tearly = [];
Tlate  = [];
Aearly = [];
Alate  = [];

for i=1:tracesN
  ...
    if ...
       Tlate = [Tlate time(idxMax)];
       ...
    end
end
```

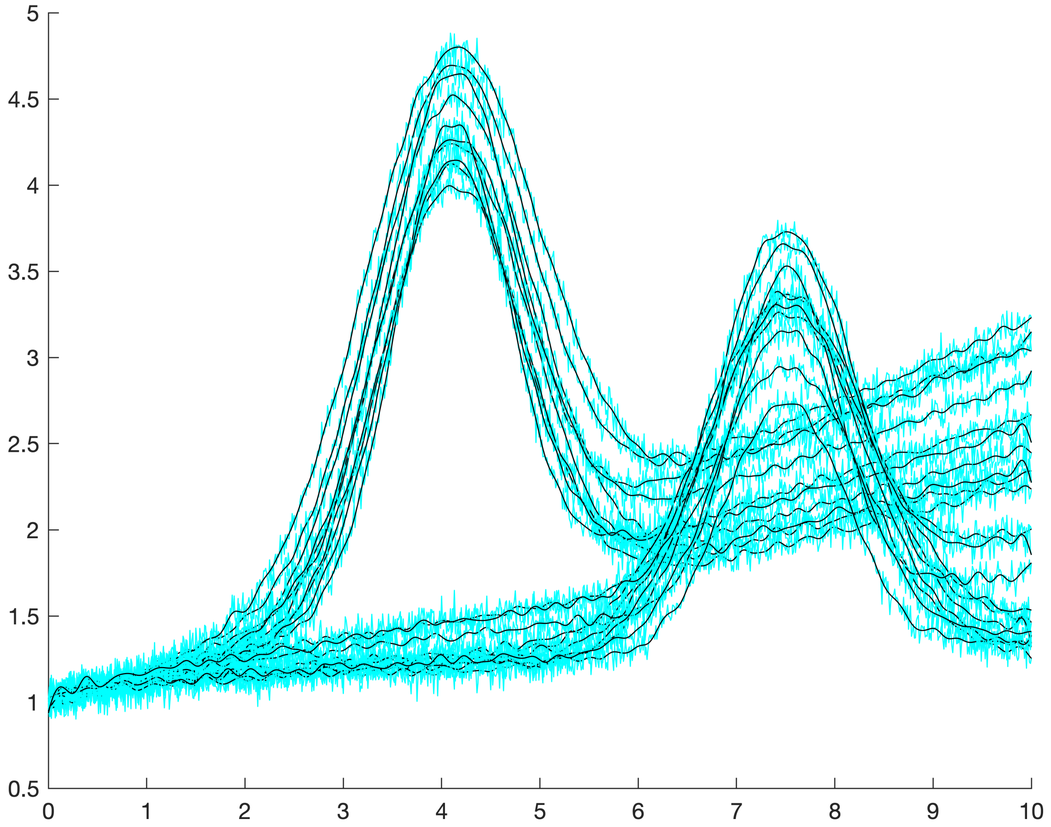
Let’s do more data!

a) Open a new script and import PeakTestData.xlsx with `readtable`.
b) Prepare the data in the manner as described on the previous slide.
c) Plot all traces into one figure. What do you notice?
d) Denoise the data with your denois-function.
e) Fit each trace with a Gaussian function (*gauss2*) and plot your result.
f) Determine the highest peak (amplitude and peak time) of each trace and mark its position.
g) Use an *if*-statement in order to distinguish between early (<6 s) and late peaks (≥6s). Re-plot traces with early peaks and late peaks in two different colors.
h) Store the peak time and amplitude into the variables `Tearly`, `Tlate`, `Aearly` and `Alate`.



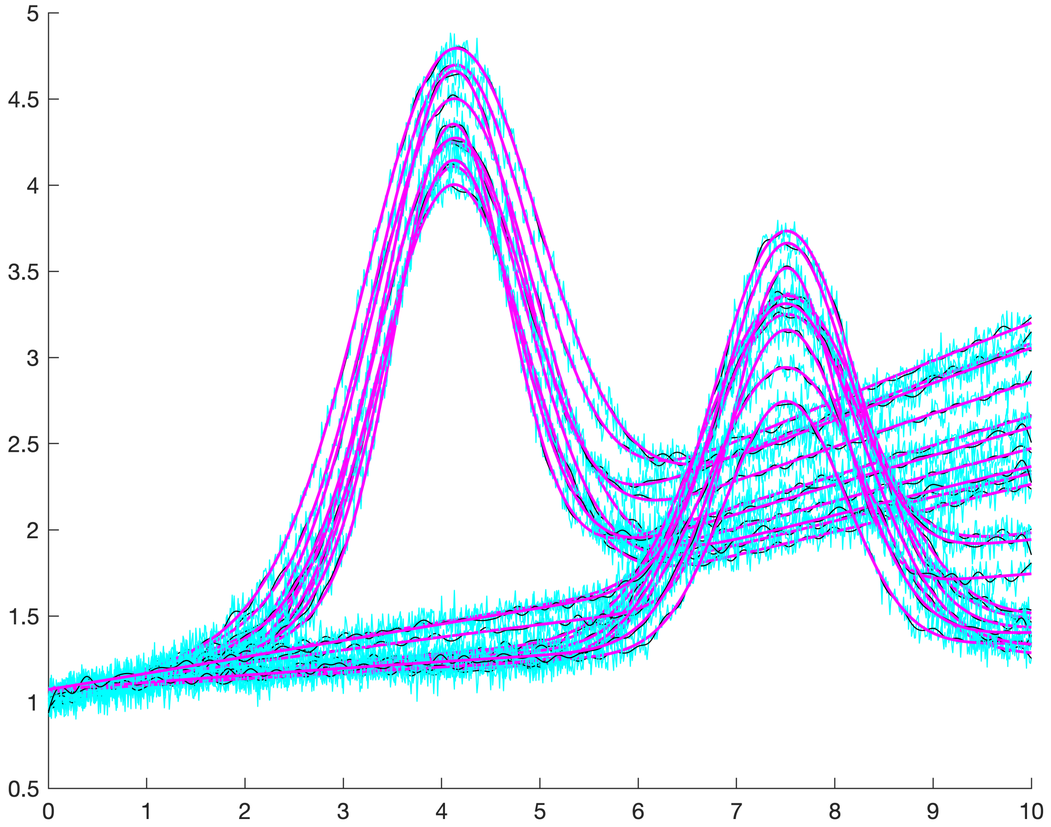
Let’s do more data!

a) Open a new script and import PeakTestData.xlsx with `readtable`.
b) Prepare the data in the manner as described on the previous slide.
c) Plot all traces into one figure. What do you notice?
d) Denoise the data with your denois-function.
e) Fit each trace with a Gaussian function (*gauss2*) and plot your result.
f) Determine the highest peak (amplitude and peak time) of each trace and mark its position.
g) Use an *if*-statement in order to distinguish between early (<6 s) and late peaks (≥6s). Re-plot traces with early peaks and late peaks in two different colors.
h) Store the peak time and amplitude into the variables `Tearly`, `Tlate`, `Aearly` and `Alate`.



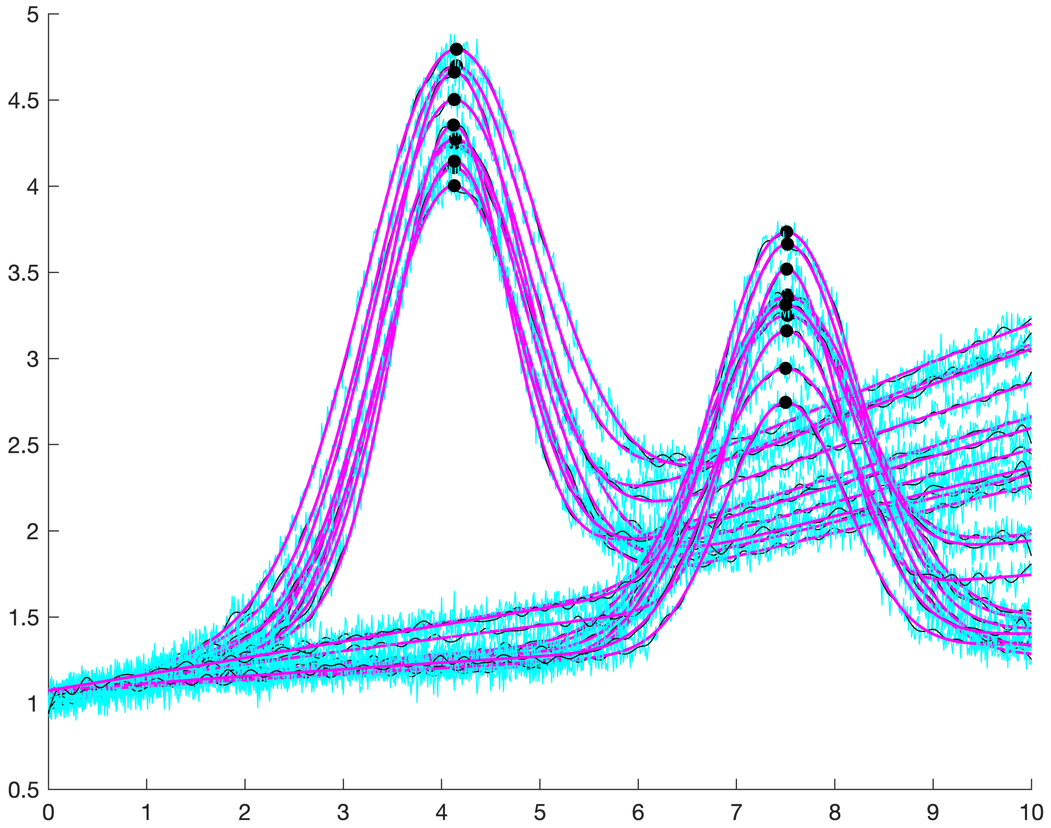
Let’s do more data!

a) Open a new script and import PeakTestData.xlsx with `readtable`.
b) Prepare the data in the manner as described on the previous slide.
c) Plot all traces into one figure. What do you notice?
d) Denoise the data with your denois-function.
e) Fit each trace with a Gaussian function (*gauss2*) and plot your result.
f) Determine the highest peak (amplitude and peak time) of each trace and mark its position.
g) Use an *if*-statement in order to distinguish between early (<6 s) and late peaks (≥6s). Re-plot traces with early peaks and late peaks in two different colors.
h) Store the peak time and amplitude into the variables `Tearly`, `Tlate`, `Aearly` and `Alate`.



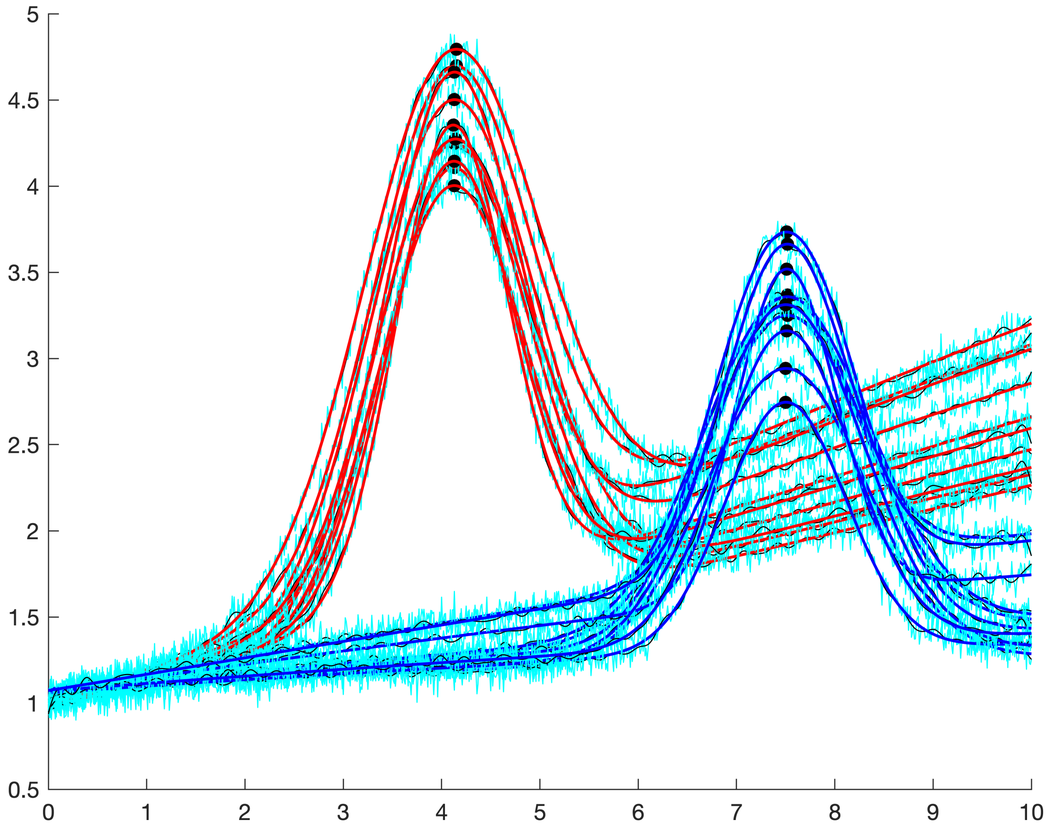
Let’s do more data!

a) Open a new script and import PeakTestData.xlsx with `readtable`.
b) Prepare the data in the manner as described on the previous slide.
c) Plot all traces into one figure. What do you notice?
d) Denoise the data with your denois-function.
e) Fit each trace with a Gaussian function (*gauss2*) and plot your result.
f) Determine the highest peak (amplitude and peak time) of each trace and mark its position.
g) Use an *if*-statement in order to distinguish between early (<6 s) and late peaks (≥6s). Re-plot traces with early peaks and late peaks in two different colors.
h) Store the peak time and amplitude into the variables `Tearly`, `Tlate`, `Aearly` and `Alate`.



Let’s do more data!

a) Open a new script and import PeakTestData.xlsx with `readtable`.
b) Prepare the data in the manner as described on the previous slide.
c) Plot all traces into one figure. What do you notice?
d) Denoise the data with your denois-function.
e) Fit each trace with a Gaussian function (*gauss2*) and plot your result.
f) Determine the highest peak (amplitude and peak time) of each trace and mark its position.
g) Use an *if*-statement in order to distinguish between early (<6 s) and late peaks (≥6s). Re-plot traces with early peaks and late peaks in two different colors.
h) Store the peak time and amplitude into the variables `Tearly`, `Tlate`, `Aearly` and `Alate`.



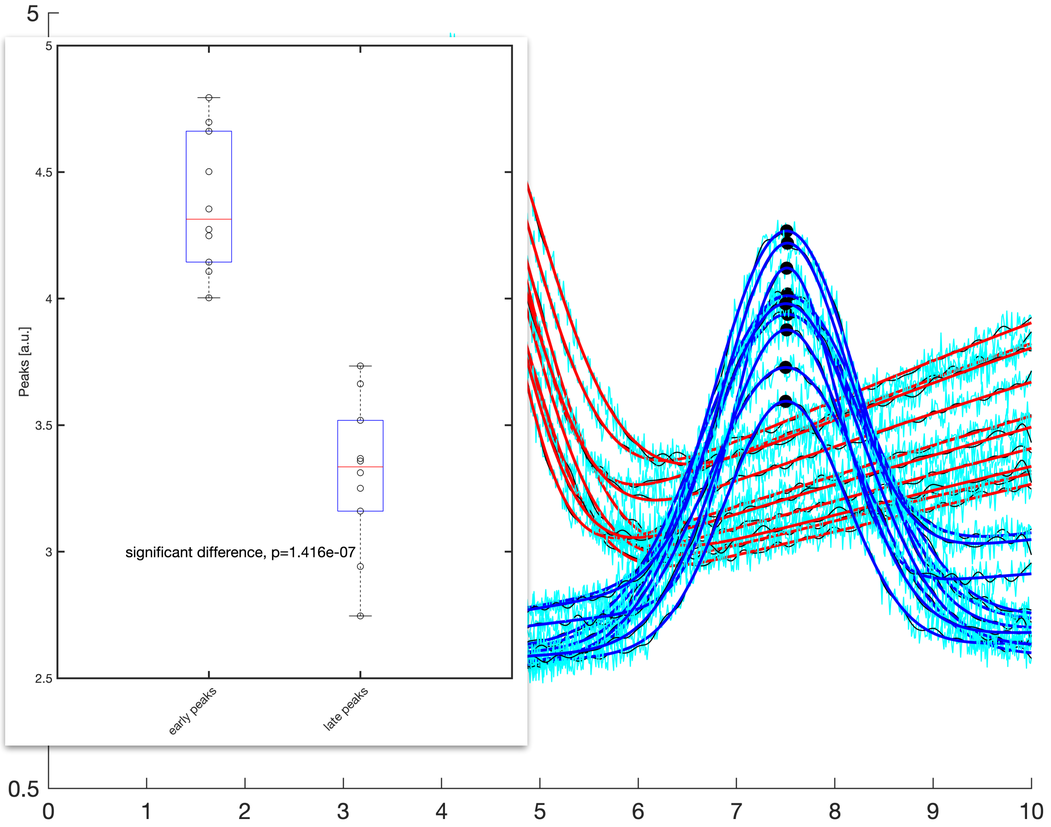
Let’s do more data!

Now we want to clarify, if there is a significant difference between our clustered data sets.

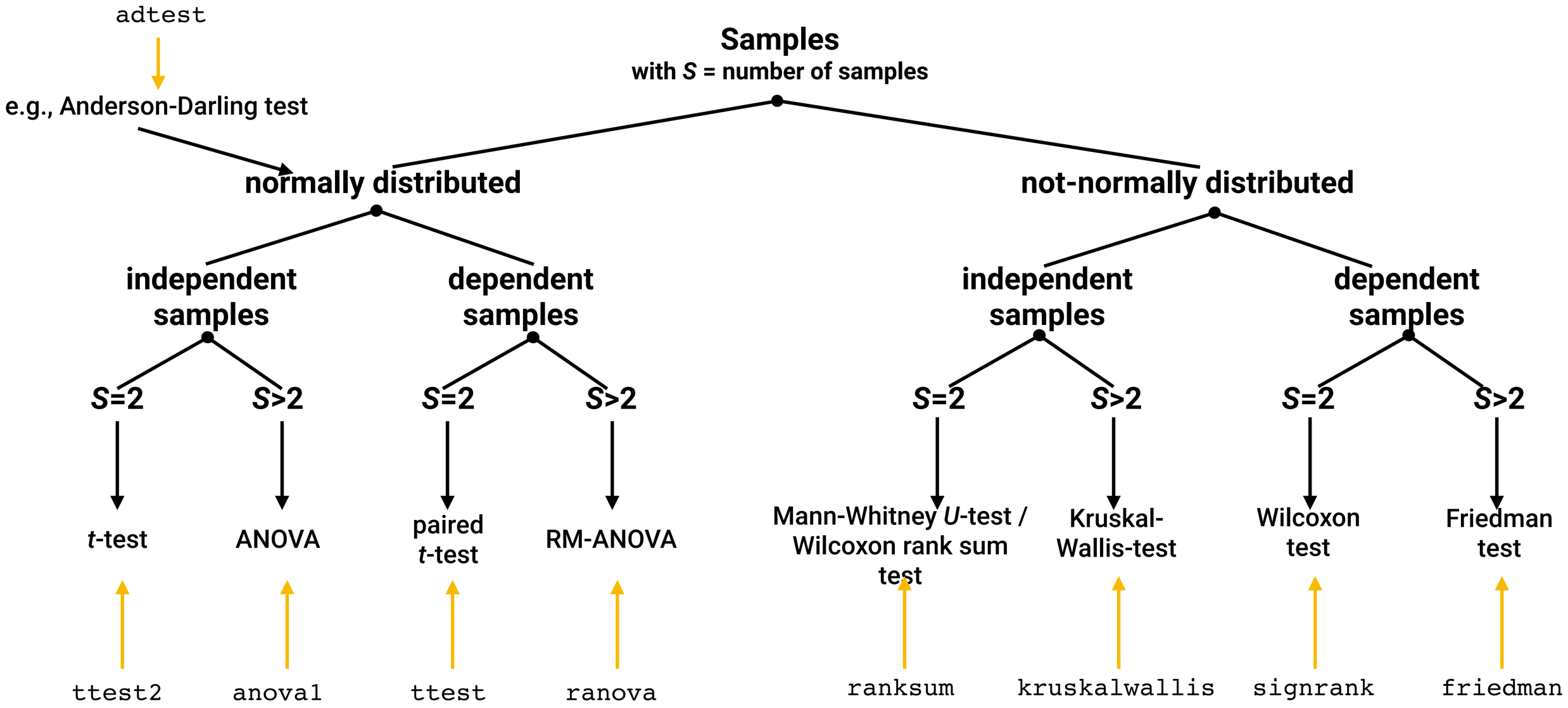
Let’s plot our clustered amplitudes into a boxplot diagram:

```
figure(14);clf
  hold on
  boxplot([Aearly' Alate'])
  plot(ones(numel(Aearly),1), Aearly, ...
              'ok', 'MarkerSize', 5)
  plot(ones(numel(Alate),1).*2, Alate, ...
              'ok', 'MarkerSize', 5)

  xlim([0 3])
  ylim([2.5 5.0])
  ylabel('Peaks [a.u.]')

  set(gca,'XTick', 1:2);
  set(gca,'XTickLabel', [{'early peaks'} {'late peaks'}]);
  xtickangle(45)
```

```
[h, p] = ttest2(Aearly, Alate);
```



Short overview of statistical tests in MATLAB



Let’s do more data!

Let's tune our plot a little bit:

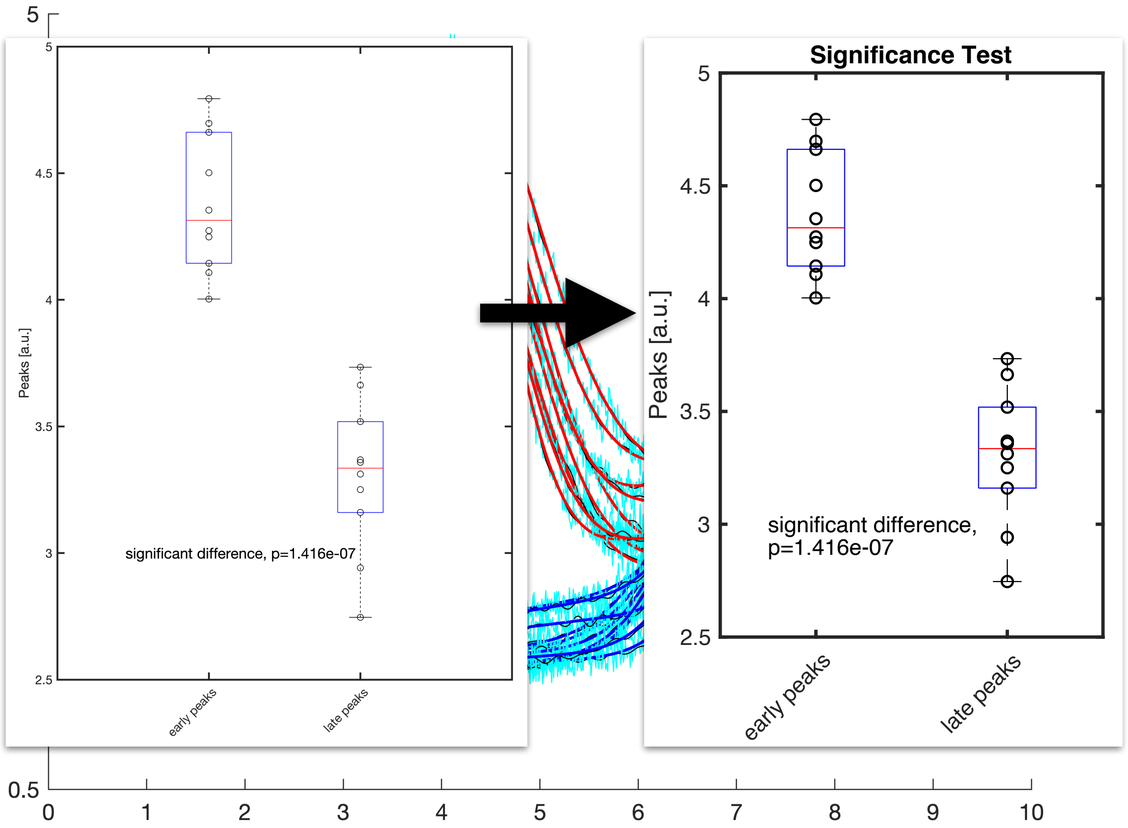
The figure window is a first approach, to visualize our data and to assess our processing steps.

We also know now, that we can save the content of the figure window into, e.g., a PNG file.

Unfortunately, the saved PNG file is a) dependent on the current aspect ratio of the figure window and b) has weak options to adjust its look.

Saving the figure as a PDF is more versatile.

But *versatile* means, we can (have!) to adjust more settings to achieve our desired output design.



Let’s do more data!

Let's tune our plot a little bit:

Saving the figure as a PDF is more versatile.

But *versatile* means, we can (have!) to adjust more settings to achieve our desired output design.

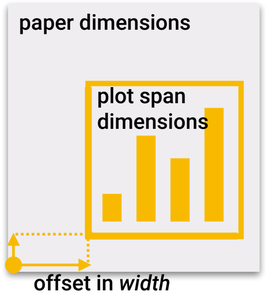
paper dimensions, *width* and *height*.
Here: 8x11.5 cm
DIN A4: 21x29 cm

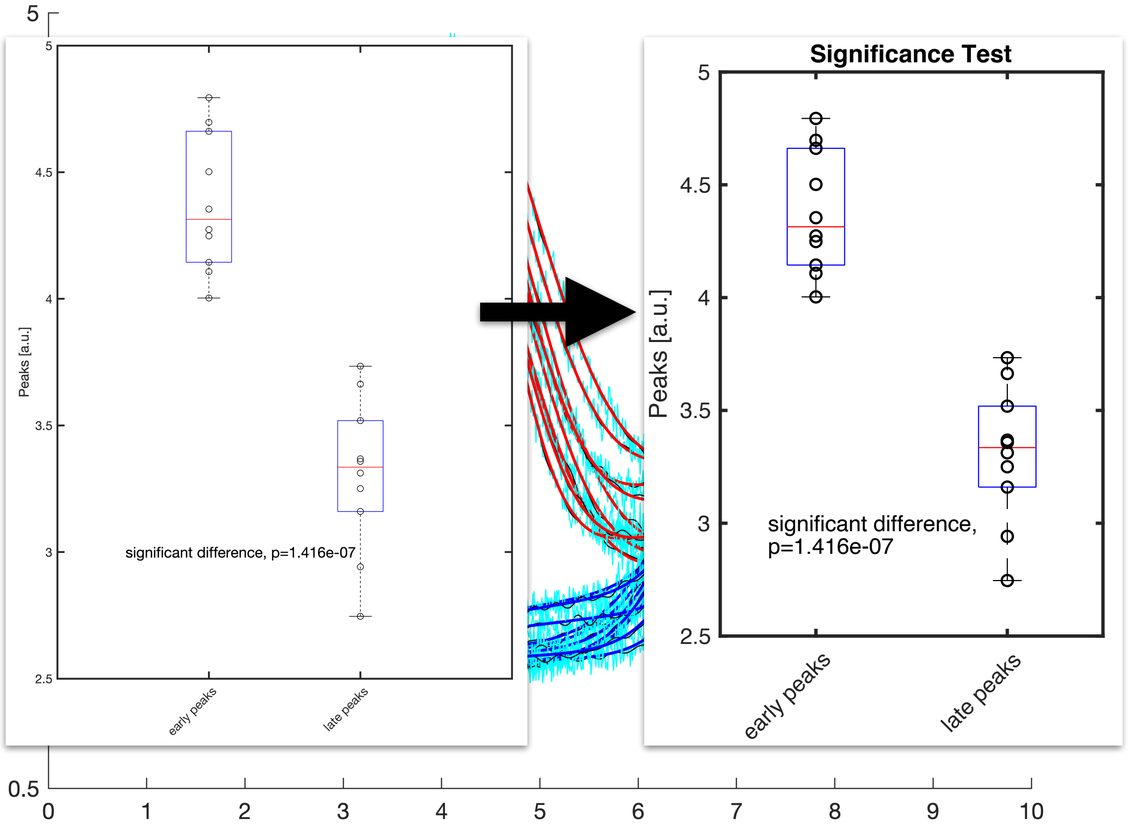
```
fig=figure('Visible','on');clf
  %set(fig,'Visible','off')
  set(fig,'PaperUnits','centimeters',...
          'PaperType','A4','PaperSize',[ 8 11.5 ],...
          'PaperOrientation', 'Portrait')
  set(fig,'PaperPosition',[0.0 0.0 [8 11.5] ]);
```

offset in *width*

offset in *height*

plot span *width* and *height* in cm
It’s a little bit weird, but sometimes these dimensions have to be slightly larger than the paper dimensions





Some words on clustering data

We clustered our data by differentiating the peak time into two classes: peak times below a certain threshold (6 s) were sorted into the “early" class, while higher peak times were sorted into the “late“ class. To achieve this analysis result, we simply made use of an *if*-statement.

In MATLAB, there are already built-in tools, which do the clustering job for us automatically, e.g., when the classification threshold is unknown. This is the case for more complex and larger data sets and/or data with multivariate dependencies.

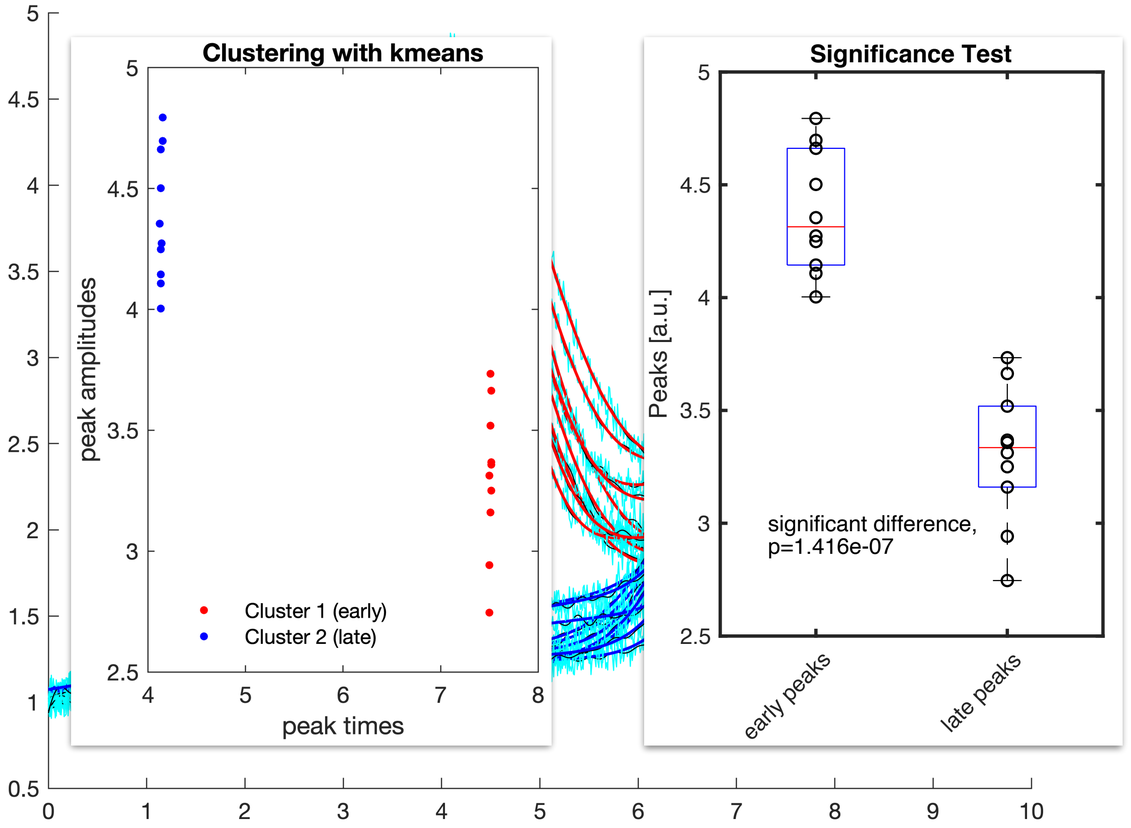
Let’s apply such tool (`kmeans`) for both of our variables, i.e., peak amplitude and peak time:

we have to provide the two variables in a certain form

desired number of clusters

```
Data2Cluster = [ [Tearly Tlate]' [Aearly Alate]'];
k=2;
idx = kmeans(Data2Cluster,k);
```

output: index-array which tells us, which data row in `Data2Cluster` belings to which cluster



Some words on clustering data

We clustered our data by differentiating the peak time into two classes: peak times below a certain threshold (6 s) were sorted into the “early" class, while higher peak times were sorted into the “late“ class. To achieve this analysis result, we simply made use of an *if*-statement.

In MATLAB, there are already built-in tools, which do the clustering job for us automatically, e.g., when the classification threshold is unknown. This is the case for more complex and larger data sets and/or data with multivariate dependencies.

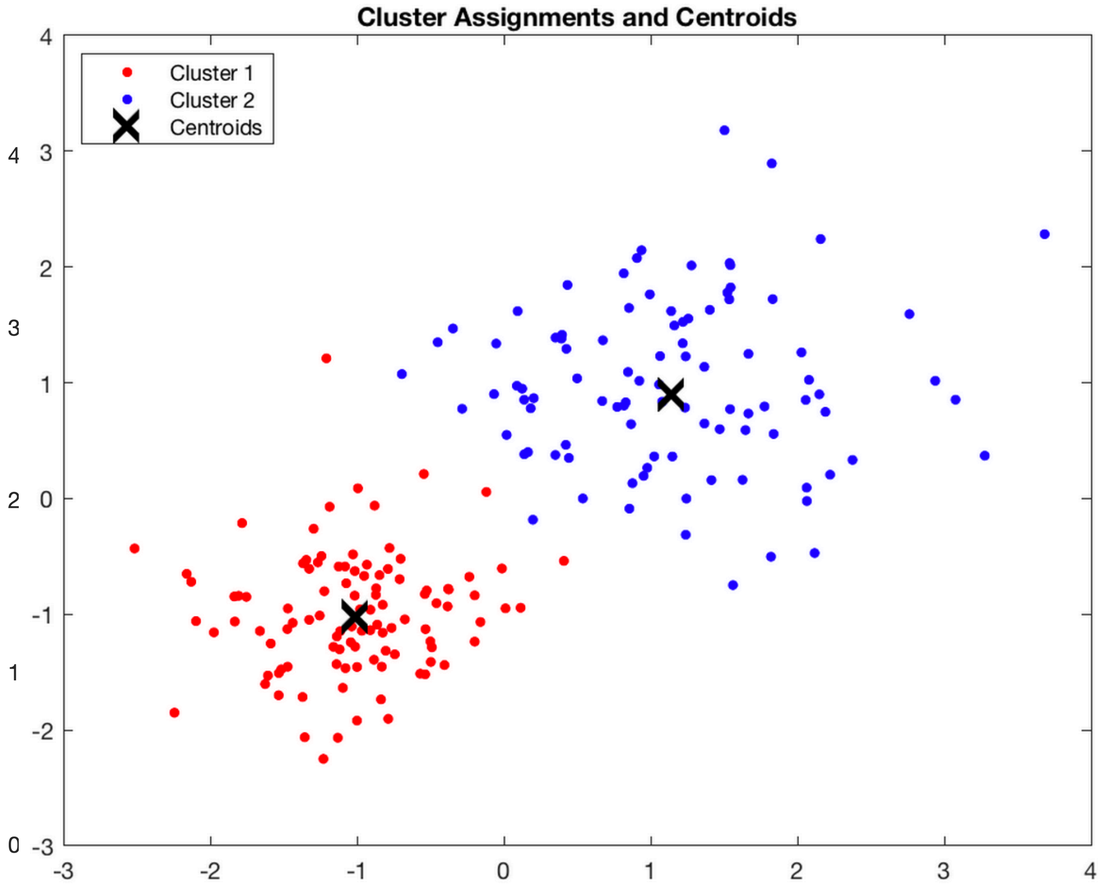
Let’s apply such tool (`kmeans`) for both of our variables, i.e., peak amplitude and peak time:

we have to proved the two variables in a certain form

desired number of clusters

```
Data2Cluster = [ [Tearly Tlate]' [Aearly Alate]'];
k=2;
idx = kmeans(Data2Cluster,k);
```

output: index-array which tells us, which data row in `Data2Cluster` belings to which cluster



Some words on clustering data

We clustered our data by differentiating the peak time into two classes: peak times below a certain threshold (6 s) were sorted into the “early" class, while higher peak times were sorted into the “late“ class. To achieve this analysis result, we simply made use of an *if*-statement.

In MATLAB, there are already built-in tools, which do the clustering job for us automatically, e.g., when the classification threshold is unknown. This is the case for more complex and larger data sets and/or data with multivariate dependencies.

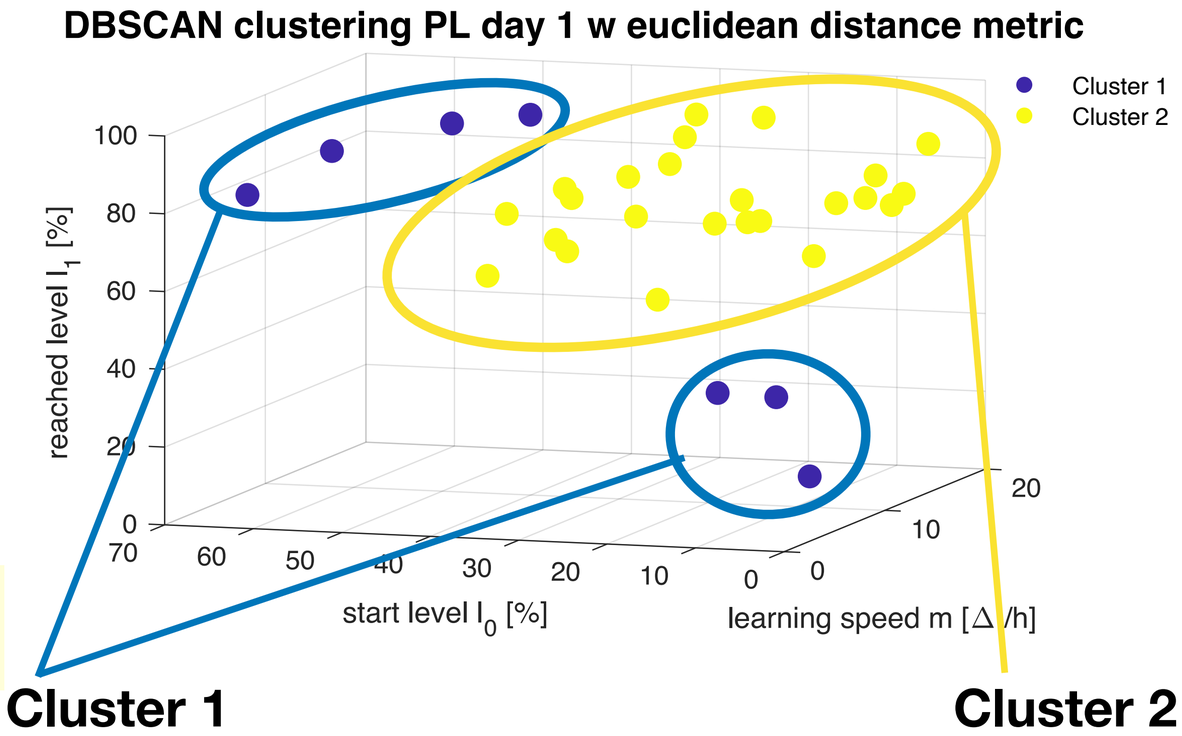
Let’s apply such tool (`kmeans`) for both of our variables, i.e., peak amplitude and peak time:

we have to proved the two variables in a certain form

desired number of clusters

```
Data2Cluster = [ [Tearly Tlate]' [Aearly Alate]'];
k=2;
idx = kmeans(Data2Cluster,k);
```

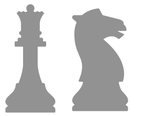
output: index-array which tells us, which data row in `Data2Cluster` belings to which cluster



Strategical Aspects

MATLAB is packed with a bunch of useful toolboxes and functions based on efficient, in-depth mathematical methods and computational algorithms.

We don’t need to know all the theories behind these functions (and may be we even can’t). But before we decide to use them, we should think about *strategy*.



A powerful algorithms might give us at the end the output that we desire. And their names sound cool, too. But may be (often!) there are other, much simpler approaches to get the desired output within a much less computation time, with much less programming effort and much less (sometimes wasted) time to understand and prepare the requirements of the fancy, but bulky algorithm.

Therefore, every new program you write starts on a piece of paper first! Make a rudimentary draft of the logical steps of your program and think about strategy and tools to achieve your analysis results. This draft won’t be your final program, but it helps you to keep track of what you want to analyze.

Don’t tie yourself too much to your initial draft. You will change it frequently (some ideas won’t work or you (or your PI) add new analytical questions to your project). Therefore, while programming, always keep a stack of paper aside from your cup of coffee or tea ;-)

Summary of what we have learned today

How to program

To program, it needs a **script**, a compiler (MATLAB) to run the script, **variables**, ***for***-loops and ***if***-statements. That's it! We have done this today! You are a programmer now! :-D

```
clear
for i=1:participantsN
  if participant==‘satisfied‘
    fprintf(1,’Hurray! :)\n’)
  end
end
```

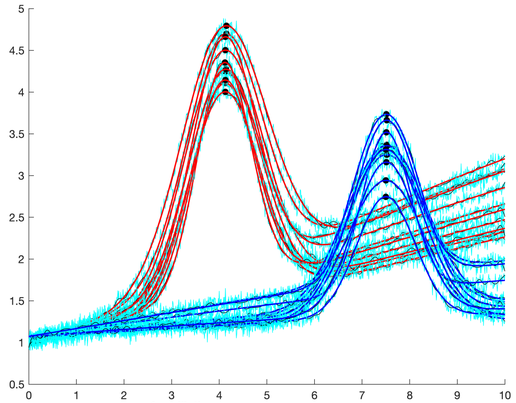
What a time series is

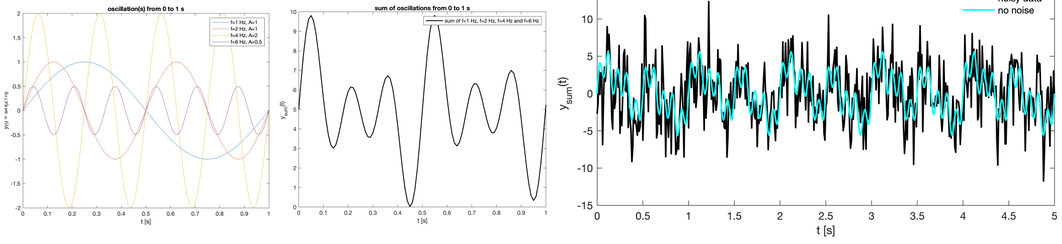
A time series is the most common ‘type’ of data you collect in your experiments. Step by step we deepened into the general generation process of a time series to understand, what is going on. Why? To process our data we need to know where our process steps act on. We (hopefully) gained an understanding for this - actually without really going too deep into maths. The good thing is, the today explained aspects are universal! Every time you look a data curve from now on, you should know what's behind it.

How to import and process data and how to export your result

Even though our artificial data set was kind of abstract and the real data set was definitely very specific, you can use the scripts from today as templates to repeat all analysis steps with your own data.

	A	B
1	time	I
2	0,5	3,97E-12
3	0,50001	3,91E-12
4	0,50002	3,78E-12





Thank you :-)

DZNE
German Center for Neurodegenerative Diseases
within the Helmholtz Association

NII
Neuroimmunology &
Imaging Group