Matlab tutorial - Session 1 - The basics

This document contains the elements presented in a tutorial on MATLAB's basics on 9.4.20 for the CRMN members by Quentin Chappuis

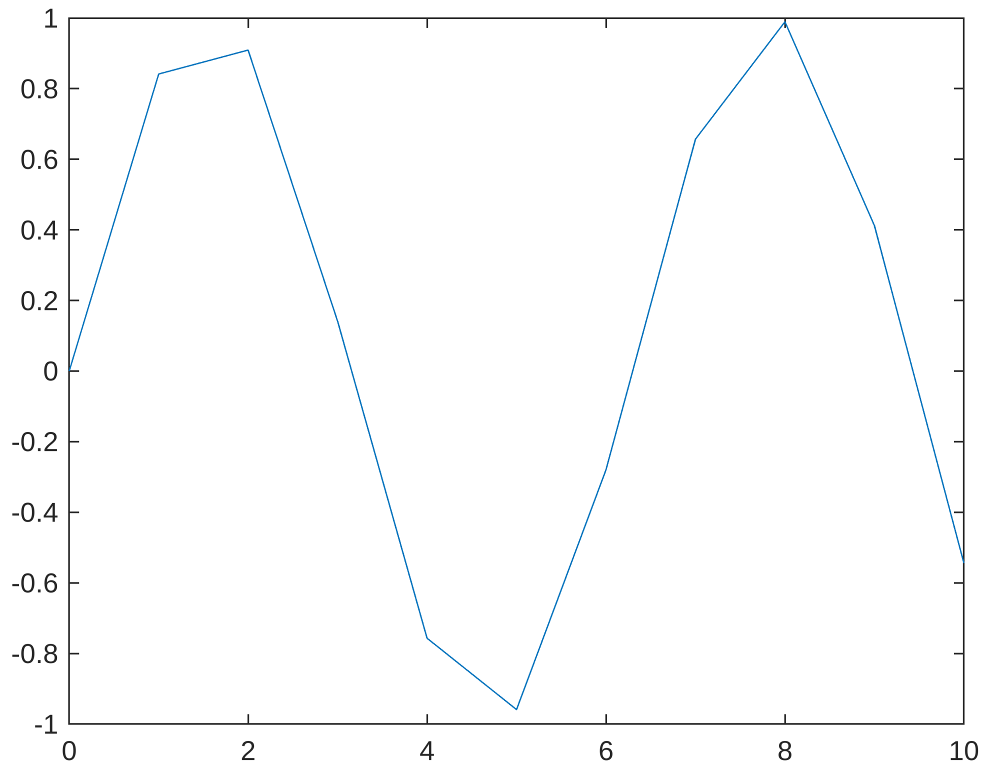
```
% How to run the code
%  - Press F5: runs the whole code
%  - Press CTRL+Enter: runs the section (sections are delimited by %%)
%  - Press F9: pastes the code that is currenlty selected by the mouse into
%    the command window and runs it

% When semilcolon ; are present after a line of code, the output of the
% code is not displayed in the command window. This is important when
% using large arrays such as NMR spectra that often contain thousands
% if not millions of points. You rarely want to display them in the command
% window. Most of the commands below are terminated wih ;. Suppress the ;
% before running in order to follow what the code does.

% Everything that is on the right side of a % is a comment. It is not
% active in the code
```

Define objects

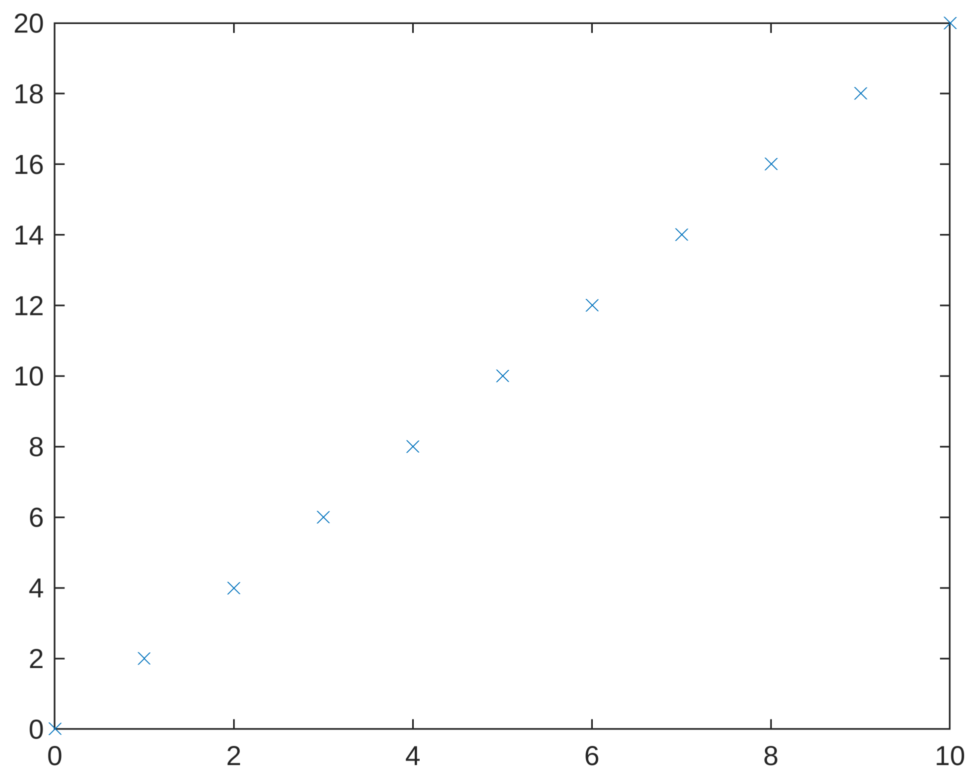
```
% Scalar
A=1;

% Vectors
RV=[1 2 3];     % Row vector
CV=[1; 2; 3];   % Column vector

% Matrices
M=[1 2 3; 4 5 6; 7 8 9];    % This creates a 3*3 matrix
O=[M; 4 5 6];               % This adds a line to the existing matrix
% results in 4*3 matrix

% Access element in the matrix or vector
%
% : accesses all elements along the dimension. If M is a matrix, then
% M(:,:) output the same result as M, Similarly, S(1:end,1:end) outputs
% the same result
% Remove the ; and run the code where you want to see the output
RV(3);
M(1, 3);
M(2,2);
M(:,1);
M(2,:);
M(2:end,3);
M(2:3,1:2);

% Useful functions
% Type "help ones" (or generally 'help FUNCTION_NAME') in the command
% window to see information about a function. Matlab's help is usually very
% instructive.
E=ones(3);      % Creates an array containing 1s (here a 3*3 matrix)
V=ones(3,1);
W=ones(1,3);
D=zeros(2);     % Creates an array containing 0s
J=eye(3);       % Creates an identity matrix (in the sense of linear algebra)
Rnd=rand(3,4);  % Creates an array containing random number between 0 and 1
% with constant distribution, Here the array is 3*4 in size
Rnd2=normrnd(0,0.1,[3 3]);  % Creates an array (3*3 in this case) with
% random numbers normal distributed, centered at 0 with a width 0.1

% Higher dimension-arrays
% We've now seen scalars, vectors and matrices. The same concepts can be
% extended to higher dimensions. The general name is an array (a scalar
% can also be seen as a one-dimensional array)
F=zeros(2,2,3);     % 3-dimensional array 2*2*3
G=ones(2,2,2,3);    % 4-dimensional array 2*2*2*3
G(1,1,2,3);         % returns a 2*3 matrix that corresponds to the first
% element dimension 1 and 2 of G.

size(G);        % Returns the size of G as a row vector, each element
% each element corresponds to the number of elements along
% one dimension
B=size(M);
B(1); % Number of rows of M
B(2); % Numbe of colums of M
```

Operations

```
clear all  % clears all elements in the workspace
t=(0:10);  % creates a row vector with integers values from 0 to 1
y=sin(t);  % computes the sin of t, element by element
plot(t,y)  % plots y as a function of x. This requires that the two vectors
```

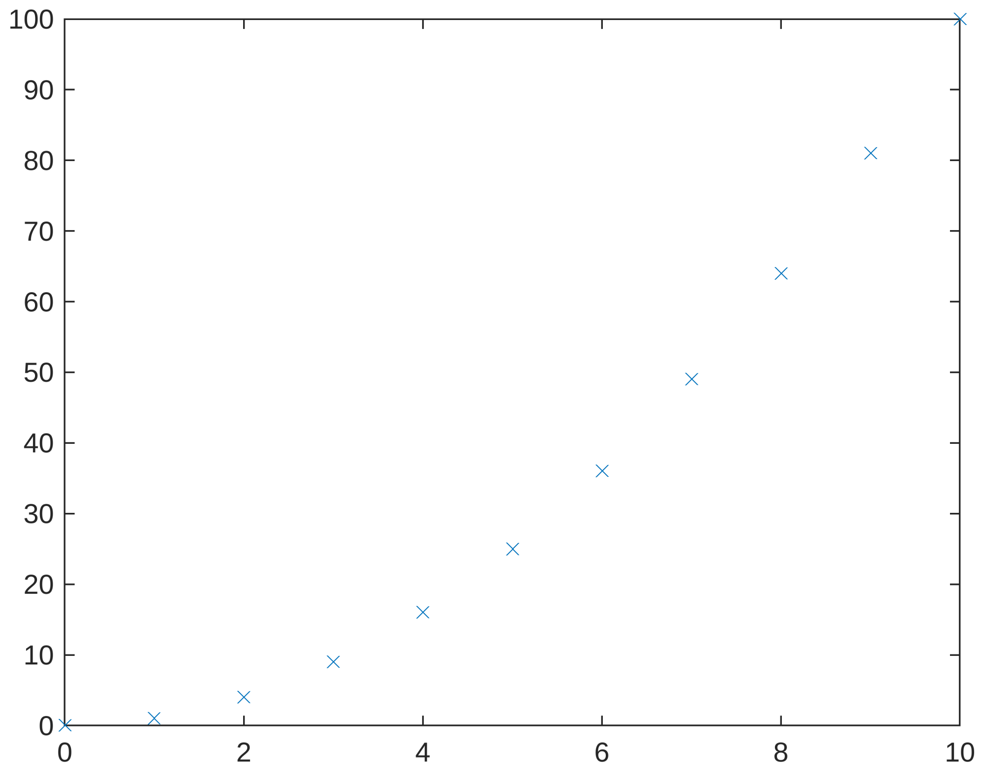


```
% have the same length

y2=2*t;    % Computes the product of the vector t with the scalar 2. In this
% case, the multiplication is non ambiguous (it's a element by
% element product)
plot(t,y2,'x') % plot y2 as a function of t using x symbols as markers
```

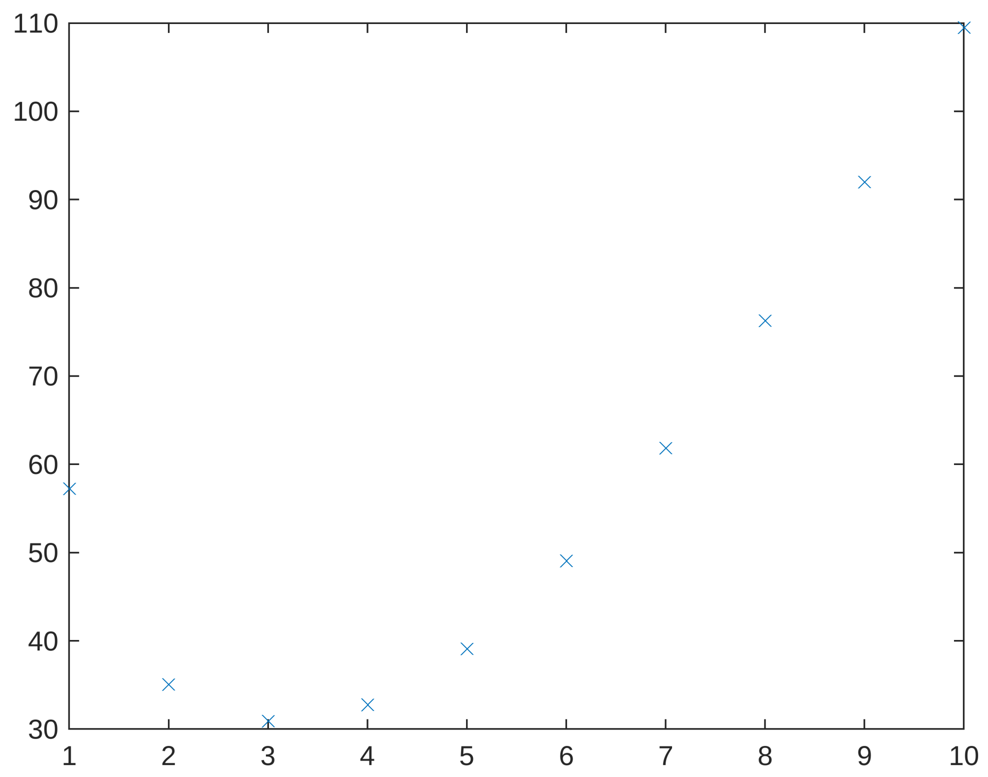


```
% the symbol ' transposes vector from row to column and vice versa. More
% generally it computes the adjoint (complex conjugate of the transpose of
% the matrix). It's not defined for higher dimensional arrays.
t;   % Row vector
t';  % Transpose of a row vector is a column vector
t*t';    % Row*Column returns a scalar
t'*t;    % Column*row returns a square matrix
% If A and B are two matrices and NA and MA is the number of rows and
% columns of A, respectively, and similarly NB and MB, then  A and B can
% be multiplied using * if the following condition is true: MA=NB. The
% results is a NA*MB matrix.
% In other words, to multiply to matrices, the number of columns of the
% first one has to be equal to number of rows of the second. The results
% has the number of rows of the first and columns of the second.
% Applying the this rules implies that the above operations are well
% defined (t*t' and t'*t). t*t would not work and Matlab would throw an
% error message.

% However, another operation is defined in the context of Matlab: the
% element by element product: it works for pairs of array with equal number
% of elements. To use it, a dot . has to be added before the multiplication
% sign *, as in the example below. This operation is loosley related to the
% dot product.
```

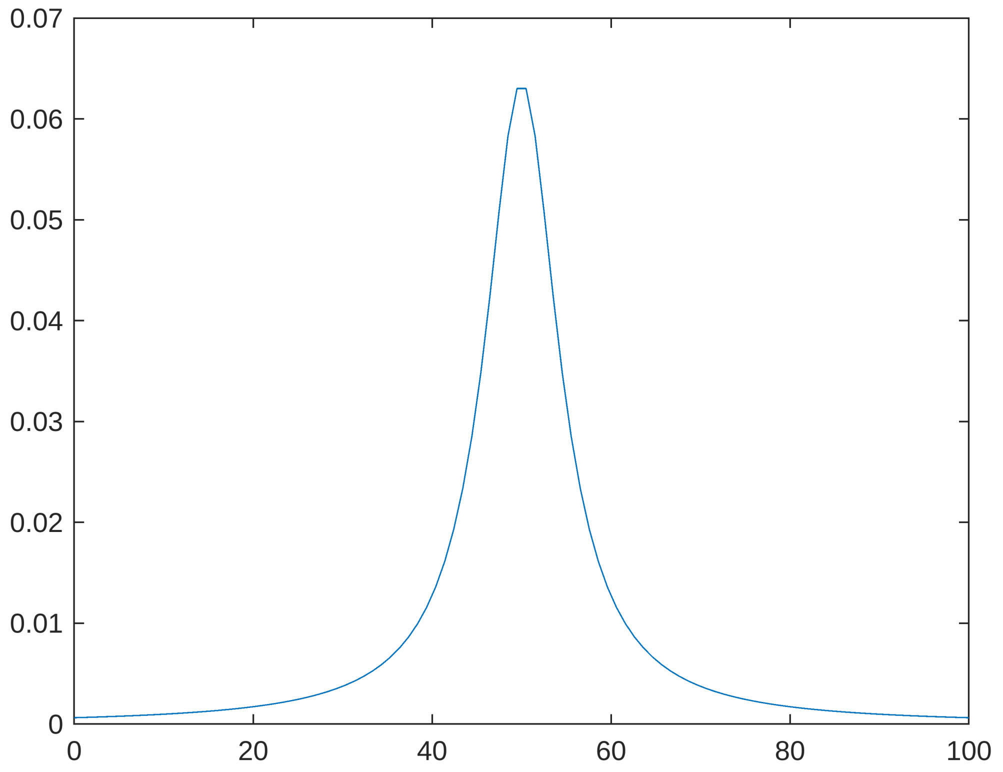
```
y3=t.^2;    % y3 is  avector which is the multiplicaiton, element by
% element, of the elements of t
plot(t,y3,'x')
```



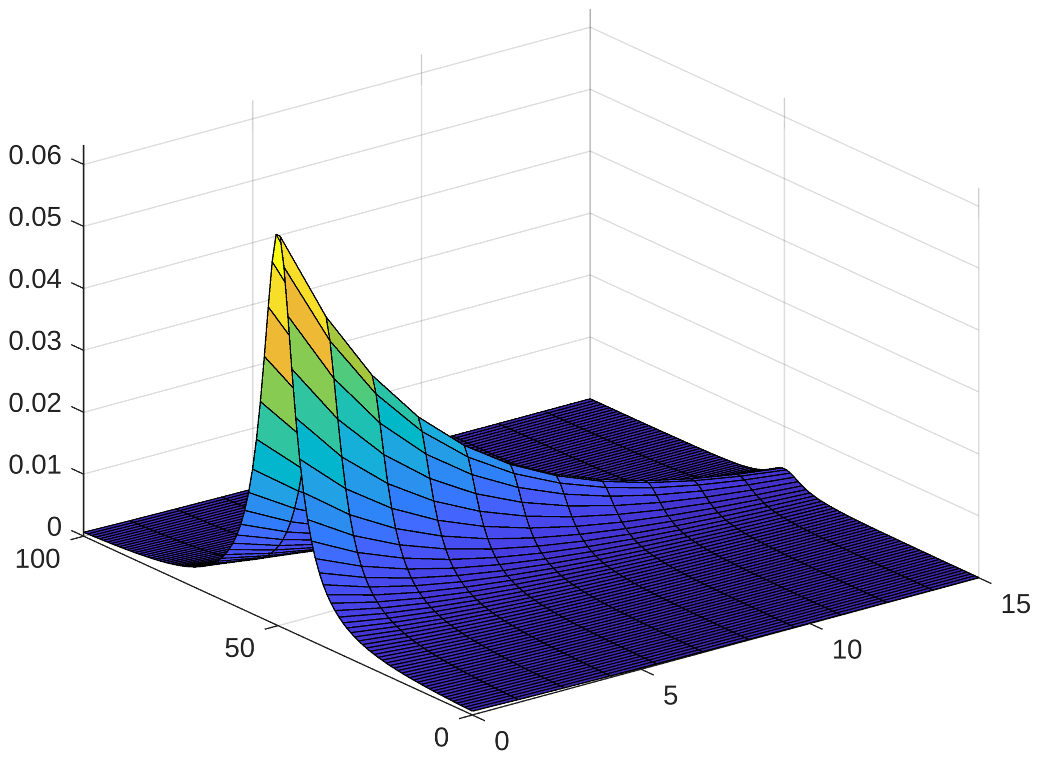
```

% In the example below, the dot . has been written whereever it was
% necessary for the calculation to be valid. Note that elementary functions
% exp and sin don't require the dot since there is not ambiguity on the
% operation. It can only be element by element.
y4=5+t.^2+50./t+sin(t)+exp(-t);
plot(t,y4,'x')
```



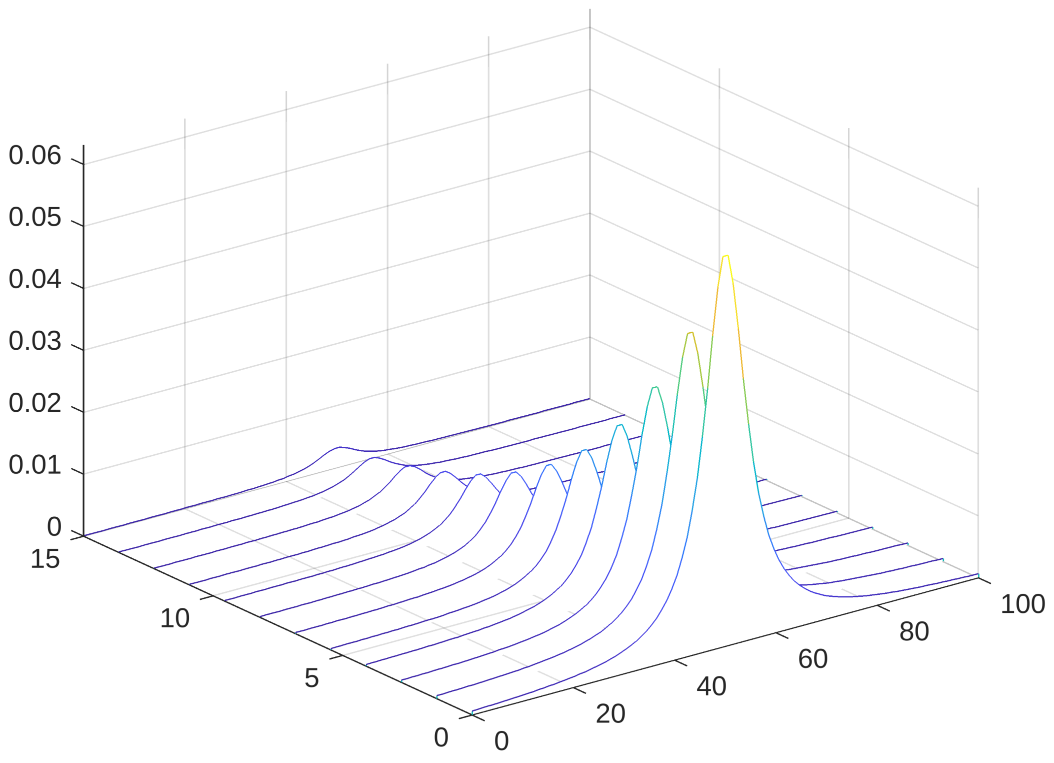
```
% Creates a Lorentzian function with N points
N=100;
w=linspace(0, 100, N);  % linspace(a,b,N) creates a row vector with N
% elements equality spread over a and b
w0=50;                  % Centre of the Lorentzian
FWHM=10;                % Full width at half maximum of the Lorentzian
% Computes the normalized Lorentzian as a row vector using the vector w
% and the scalar w and FWHM defined above.
% Pay attention to where the dot . has be added for the calculation to
% work.
S=1/pi*(FWHM/2)./((FWHM/2)^2+(w-w0).^2);
plot(w, S)
```



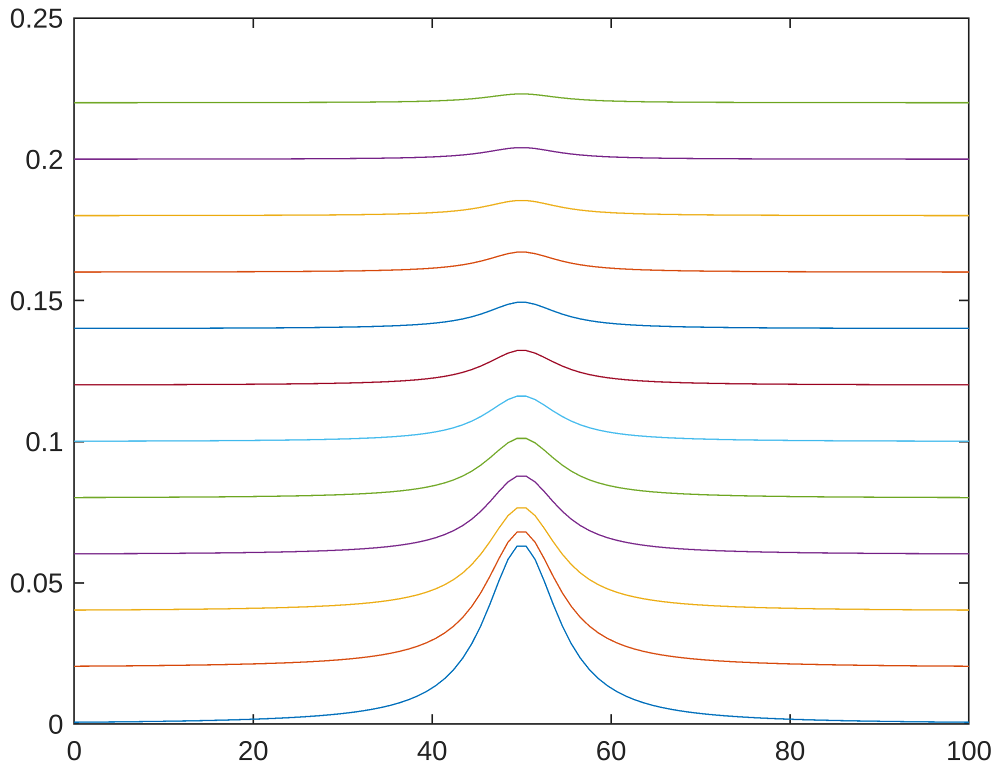
```
% Computes a monoexponential decay with time constant T1.
T1=5;                   % decay time constant
t=linspace(0,15,12);    % row vector corresponsing to time
% exp(-t/T1) creates a monoexponential decay as a row vector. The transpose
% of the signal S yields a column vector. The multiplication of the two in
% the appropriate order returns a matrix corresponding to a simulated
% 2D experiment of decaying signal
S2D=S'*exp(-t/T1);

figure, surf(t,w,S2D)        % Plots the result as a surface
```



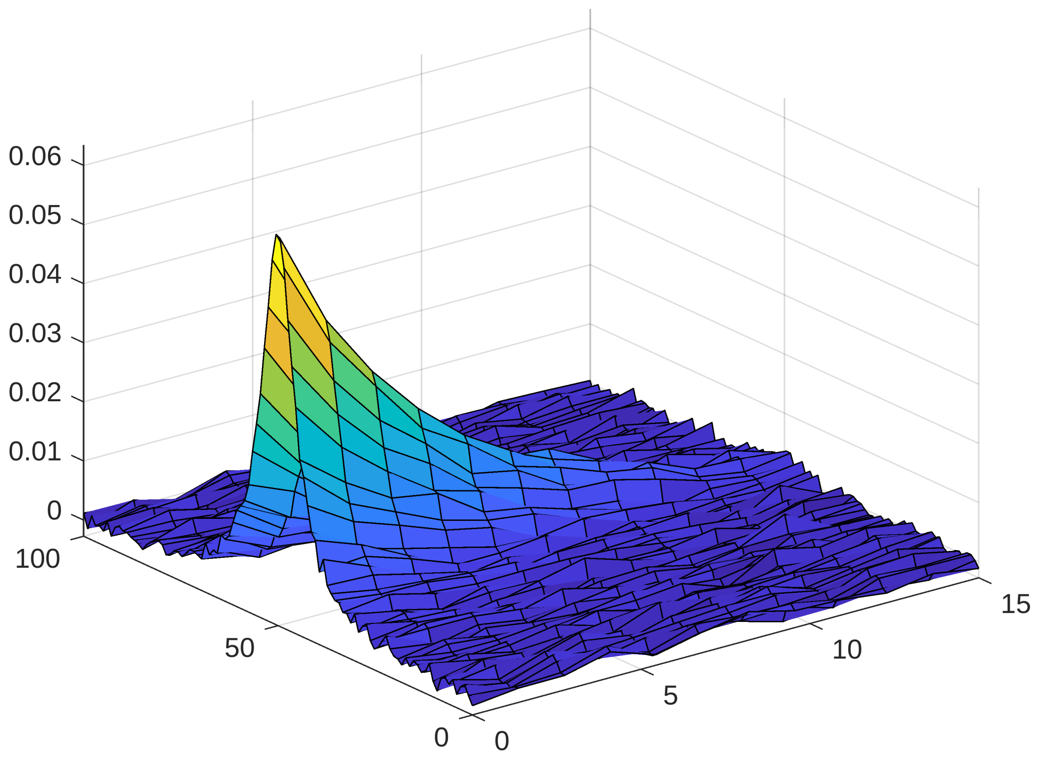
```
figure, waterfall(w,t,S2D')  % Plots the results a series of 1D graph
```



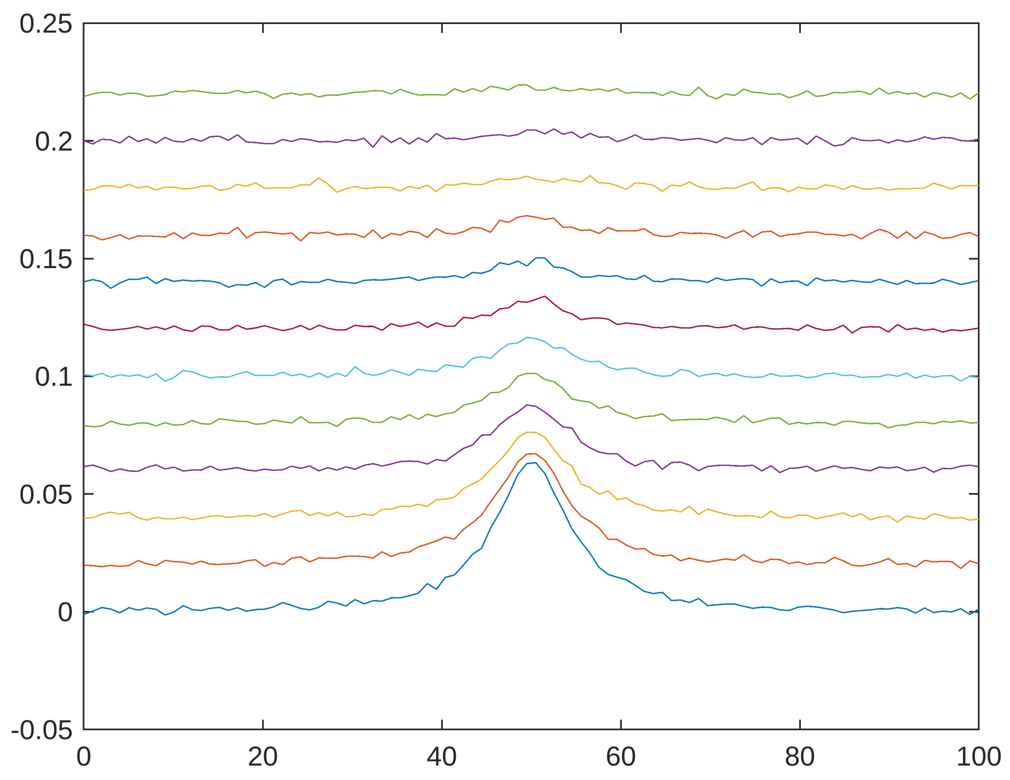
```
figure, plot(w,S2D+ones(N,1)*(0:11)/50) % Plots the result in a Topspin looking way
```



```
% This codes simulates a 2D spectrum with gaussian noise.
S2DwithNoise=S2D+normrnd(0,0.001,size(S2D));
figure, surf(t,w,S2DwithNoise)
```



```
figure, plot(w,S2DwithNoise+ones(N,1)*(0:11)/50)
```



```
% Kronecker product or tensor product
% Rapid introduction to the fact that other operations are defined in
% Matlab.
I2=eye(2);
Ix=1/2*[0 1;1 0];
kron(eye(2), Ix)
```

```
ans = 4×4
         0    0.5000         0         0
    0.5000         0         0         0
         0         0         0    0.5000
         0         0    0.5000         0
```

if Statements

If statement allows to apply parts of code only if certain conditions are fulfilled. Running the codes below should make it clear.

```
A=3;
if A==2
    disp('First condition fulfilled')
elseif A==3
    disp('Second condition fulfilled')
else
    disp('No condition fulfilled')
```

```
end
```

```
Second condition fulfilled
```

```
disp(' ')
```

```
% Conditions can be summed using %% Then all the conditions have to be
% fulfilled
% ~= means "is not equal"
B=5;
C=10;
if A==3 && B~=4 && C>5
    disp('yeah')
else
    disp('nope')
end
```

```
yeah
```

```
if A>0
    disp('cond 1')
elseif A==3
    disp('cond 2')
end
```

```
cond 1
```

for loop

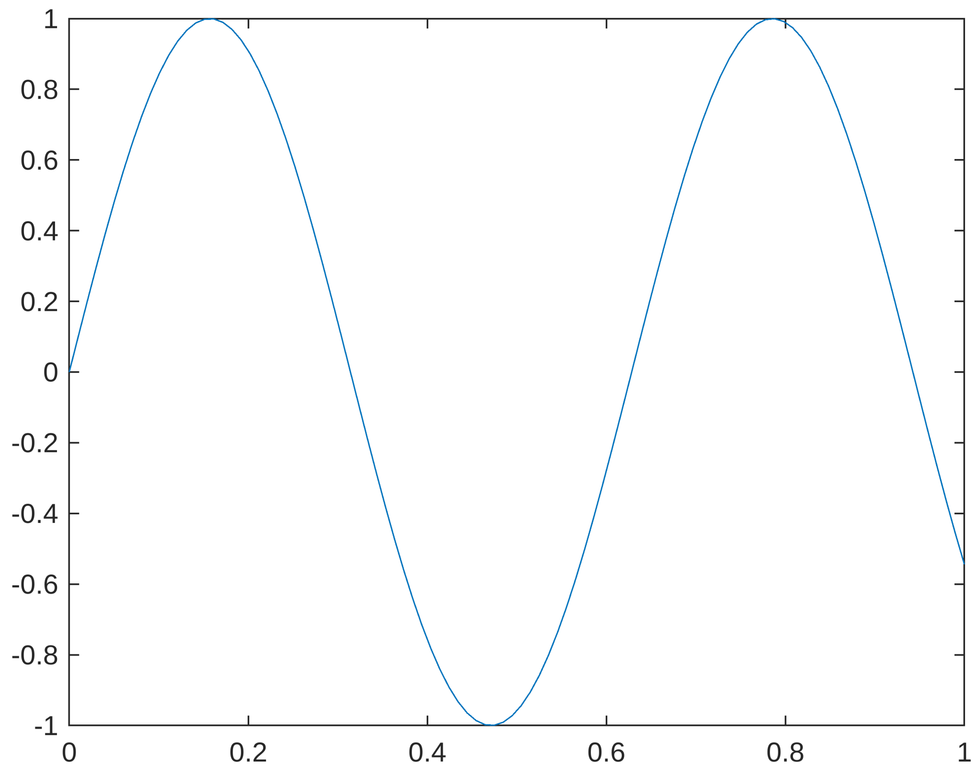
for loops allow to run a code iteratively. The piece of code is repeated incrementing a value of an integer variable at each iteration.

```
% In this example the code disp(i) will be repeated with i taking values
% from 1 to 3 (that is, 1, 2 and then 3, one after the other)
for i=1:3
    disp(i)
end
```

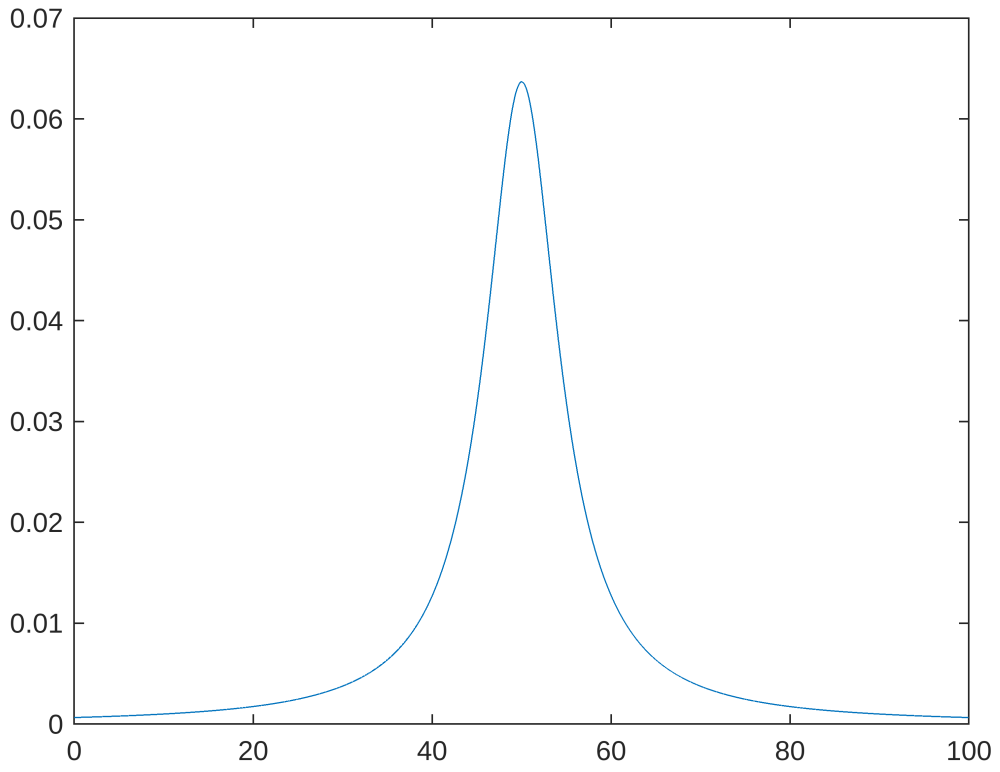
```
     1

     2

     3
```

```
% This code calculates the sin of t by explicitly calling the elements of t
% and placing them at the appropriate location in the vector g using the
% for loop.
N=100;
```

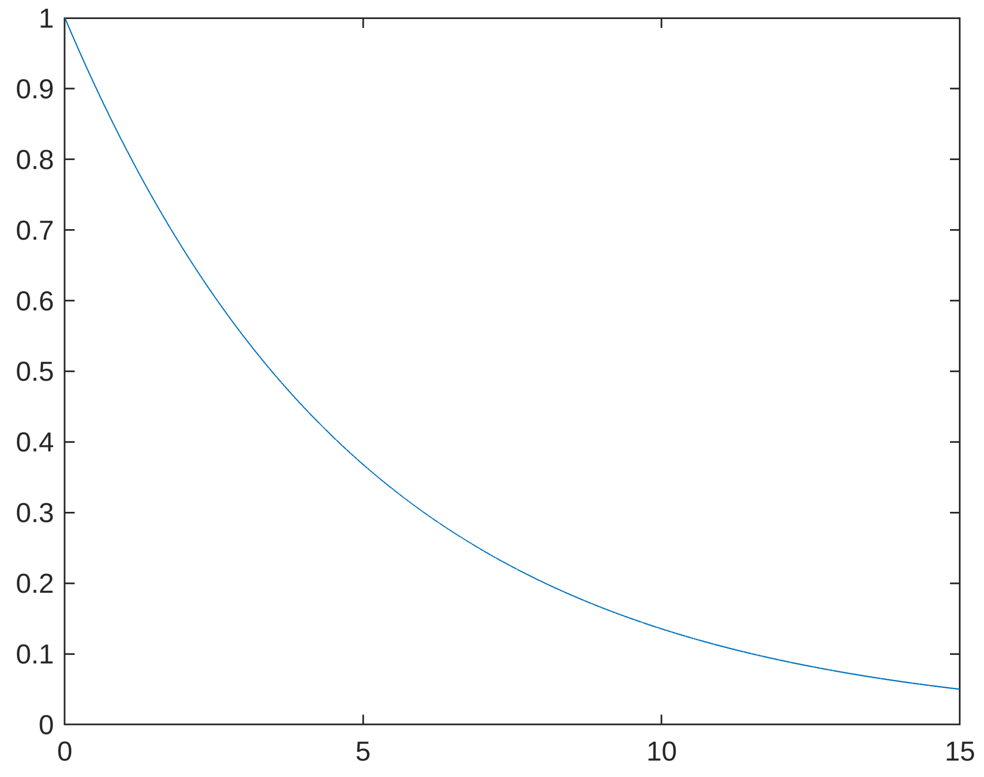
```
t=linspace(0,1,N);
g=zeros(N,1); % g is defined an empty vector (a vector of zeros) prior to
% using it in the code. This is the general good practice.
% If you don't declare it in prior to the loop, then Matlab
% has to reallocate space for g at each iteration of the
% loop. This massively slows down the process
% Try to comment the line g=zeros(N,1) (or suppress it),
% you'll see that Maltab then advices you to modify your code
% in the loop below.

for i=1:N
    g(i)=sin(t(i)*10);
end
plot(t, g)
```



```
% This performs the same calculation as earlier but for much larger
% vectors.
N=10000;
w=linspace(0, 100, N);
w0=50;
FWHM=10;
S=1/pi*(FWHM/2)./((FWHM/2)^2+(w-w0).^2);
plot(w, S)
```



```
M=20000;
T1=5;
t=linspace(0,15,M);
D=exp(-t/T1);
plot(t,D)
```



```
% tic and toc allows to show the time ellapsed between tic and toc commands
% of the code.

% Compare case 1 and 2. Case 2 is the appropriate way to perform a
% calculation in Matlab. For loops should not be used for this sort of
% calculation because Matlab is well optimized for matrix calculation. If
% you use a loop, you don't use all those nice optimzation. See how case 1
% is slower than case 2.

% Case 1
tic
S2D=zeros(N,M);
for i=1:N
    for j=1:M
        S2D(i,j)=S(i)*D(j);
    end
end
t1=toc;

% Case 2
tic
S2D=S'*exp(-t/T1);
t2=toc;
```

```
% This shows the ratio between the time it took for Matlab to perform the
% caculation in case 1 and 2. This ratio is around 7 on my computer
disp(t1/t2)
```

```
    29.0251
```

while loop

while loops perform an operation as long as a condition is not fulfilled.

```
N=3;
while N<=10
    disp(N)
    N=N+1;
end
```

```
     3

     4

     5

     6

     7

     8

     9

    10
```

```

% Careful. The example below would run for an infite time if you were to
% run it (to observe it, remove % comment marks):

%     while N~=0
%         disp(N)
%     end

% If you realized that you have made such a mistake and that Matlab has
% entered an endless loop, go in the command window and press CTRL+C. This
% kills the process.
```

Using structures and cells

Cells can contain elements of different sorts. In the example below, cells 1, 2 and 3 of A contain a scalar, a char and a matrix, respectively. This is an useful feature but the downside is that operations between groups of cells are not defined.

```
clear all
A{1}=1;
A{2}='d';
A{3}=ones(3);
```

```
A{2,2}='e';

% This syntax creates elements H and G that are part of the structure F. F
% is then nohting more than a sort of location containing H and G. As for
% cells, the elements listed in the structure can be of different nature.
F.H=1;
F.G=zeros(3);
F.G(2);
```

importing NMR data

Thanks to Nils Nyberg, we have the a function that imports topspin data in a very complete way. It is available here: https://de.mathworks.com/matlabcentral/fileexchange/40332-rbnmr

```
% You can also get it from my list of Matlab function
% https://mycore.core-cloud.net/index.php/s/X6abY5o4JQTrCfD

% In the example below I load data from a dDNP experiment. You can download
% this data from the mycore link above. To run the code of the sections
% below, download the folder named 150 and unzip it. Then modify the path
% below so that rbnmr can access it in your computer.
Path='C:\Bruker\TopSpin4.0.7\examdata\20200107_Dissolution\150';

% rbnmr creates a structure of structures containing all the parameters
% of Topspin and of course the NMR data themselves. It takes the path of
% your data as input.
NMRdata=rbnmr(Path);
```

```
Unrecognized function or variable 'rbnmr'.
```

```

% If there are several proc in your exp (I use Topspin wording here), rbnmr
% returns a group of cells (as a column), each cell correpsonds to a proc.
% I usually only use the first proc. For convenience, I erase the rest
% using the line below:
NMRdata=NMRdata{1};
```

Playing with NMR data

```
w=NMRdata.XAxis;    % Takes the frequency axis created by rbnmr and
% stores in a vector called w
S2D=NMRdata.Data';  % Saves the 2D signal saved in Data
% Here I take the transpose of Data. This is because rbnmr is coded in such
% a way that spectra a stored as row vectors. I always find it more
% convenient to work with column vectors (matter of taste). By transposing
% the matrix, I turn the row vector-spectra into colum-vector spectra.
figure, plot(w, S2D(:,1))   % Creates a (very heavy) figure of the 2D data
```

Region selection and integration

Here I define two regions (in point number) where I want to integrate my signal.

```
SB=[3100 3500];     % Signal boundaries
NB=[13000 15000];   % Noise boundaries

% subplot(N,M,O) seperates the plot area in a grid with N rows and M
% columns. Once you type subplot(N,M,O), what you now write acts on the
% Oth plot cell of the grid. In the example below, I create a grid with 3
% cells.

figure
subplot(1, 3, 1)
% The first plot contains the full spectrum (on which I can zoom to
% find the point numbers where I want to integrate).
plot(S2D(:,1))
title('Whole spectrum')
xlabel('Freq/ppm')
ylabel('Relative intensity/a.u.')
% The syntax plot(S2D(:,1)) gives only one input (a vector) for plot.
% That's why it displays the vector on the y-axis and the x-axis is the
% point number (or index number). It is equivalent to
% plot((1:size(S2D,1),S2D(:,1))

subplot(1, 3, 2)
% The second contains the first spectrum that was choped to show only the
% selected signal of interest. The x-axis is frequency units
plot(w(SB(1):SB(2)), S2D(SB(1):SB(2),1))
title('Signal')
xlabel('freq/ppm')

subplot(1, 3, 3)
% The third contains the first spectrum that was choped to show only a region
% of noise that can be used for SNR calculation. The x-axis is frequency
% units
plot(w(NB(1):NB(2)), S2D(NB(1):NB(2),1))
title('Noise')
xlabel('freq/ppm')
```

Remove empty spectrum

If the 2D experiment (or pseudo 2D in this case) contains empty spectra (vectors of zeros), it is convenient to use the code below to suppress them.

```
N=size(S2D,2);  % The number of elements in the second dimension of S2D
% corresponds to the number of spectra of the 2D experiment.
% It is stored as variable N.

% This while loop will perform the code within the loop as long as the
% condition sum(S2D(:,N))==0 is not true. This is condition is equivalent
% to saying "the integral of spectrum N is 0" or in other words "spectrum N
% is empty".
while sum(S2D(:,N))==0
    % At each iteration of the loop, the value of N is decreased by 1.
    N=N-1;
end
% The loop stops when the condition is fulfilled: The N th specturm of
% S2D(:,N) is not empty. N then corresponds to the last non empty spectrum.

% This chops out the empty spectra.
S2D=S2D(:,1:N);
```

Integration

Now that the empty spectra spectra have been removed, S2D only contains the relevant information. We can now integrate the spetra for example in order to fit the T1.

```
% This piece of code contains information that the user (me) knows from how
% the experiment was performed i.e. it was performed as pseudo2D
% experiment separating each aquisition by a constant delay D1.
% This code is not completely general.

% Gets the D1 (repetition time between the acquisitions). Be careful: in
% Topspin, the first delay of the vector D starts at 0. In Matlab, indices
% start a 1. So the delay that was called D1 in Topspin shifts to D(2) in
% Matlab.
D1=NMRdata.Acqus.D(2);

% Creates a time vector
t=linspace(0,D1*(size(S2D,2)-1),size(S2D,2));

% Sums the signal over the range defined earlier
I=sum(S2D(SB(1):SB(2),:),1);

% Normalizes the signal
I=I/max(I);

% Creates a figure with the signal integral
figure
plot(t, I)
grid on
xlim([min(t) max(t)])
ylim([-0.05 1.05])
xlabel 'time/s'
ylabel 'Relative signal integral/a.u.'
```