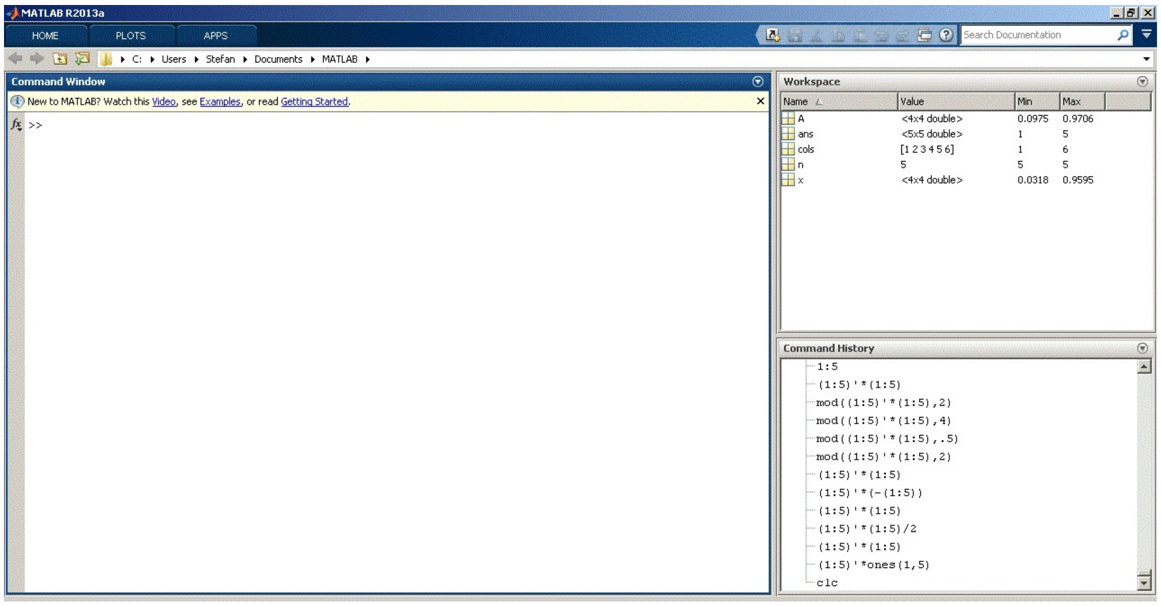
Ex. 1, Vectors and Functions

Stefan Karlsson

January 15, 2014

1 Introduction


MATLAB R2013a
HOME
PLOTS
APPS
Search Documentation
C: ▸ Users ▸ Stefan ▸ Documents ▸ MATLAB ▸
Command Window
New to MATLAB? Watch this Video, see Examples, or read Getting Started.
Workspace
Command History

```
>>
```

Name	Value	Min	Max
A	<4x4 double>	0.0975	0.9706
ans	<5x5 double>	1	5
cols	[1 2 3 4 5 6]	1	6
n	5	5	5
x	<4x4 double>	0.0318	0.9595

```
1:5
(1:5)'*(1:5)
mod((1:5)'*(1:5),2)
mod((1:5)'*(1:5),4)
mod((1:5)'*(1:5),.5)
mod((1:5)'*(1:5),2)
(1:5)'*(1:5)
(1:5)'*(-(1:5))
(1:5)'*(1:5)
(1:5)'*(1:5)/2
(1:5)'*(1:5)
(1:5)'*ones(1,5)
clc
```

Figure 1: The appearance of Matlab desktop.

Start Matlab and get familiar with the environment. There are usually several subwindows or panes in the main matlab window. In fig. 1, we see 3 subwindows[1], command window, workspace and command history. In the command window you issue all commands. Type in the command window(and hit enter):

```
a = 1
```

[1]the number of subwindows vary depending on your settings, but there is always a command window

You just made your first variable, and assigned it value 1. If you have the window workspace available, you will see the variable listed there. Now type

```
b = 2;
a + b
```

to generate a new variable `b`, and add the values together. Lets make a script of commands, that we can save in a text file. First thing to note is that Matlab is always working with an active folder, and will put all the files we generate there by default. The active folder is visible on the top of the main matlab window. Make sure you are in a folder where you want to gather the results from this exercise. To generate a script[2] in the current folder, type...

```
edit myFirstScript.m
```

and let Matlab create the file for you. You will automatically enter the Matlab editor, and see an empty file content. Type in the file the following:

```
%Any text following a "%" is treated as remarks
a = 1;                  %first variable,
b = 2;                  %second...
c = 3;                  % ... and third
d = a+b+c               %finally, sum them all up
```

Save it, go back to the command window and type:

```
myFirstScript
```

The output should be as you expect, and your workspace window now lists 2 additional variables `c` and `d`.

Matlab stands for MATrix LABoratory. It's a language that was originally made for manipulating vectors and matrices, and that is still where its strongest uses are. There are many ways of generating Matrices and vectors[3], and we will explore vectors in this exercise.

2 Vectors

To generate a vector with some values(including π and $\sqrt{2}$):

```
a = [8, -10, 15, 1.1, pi, sqrt(2)]
```

To generate a vector of increasing integers from 1 to 10:

```
a = 1:10
```

For a vector of integers that increment with 2:

```
a = 1:2:10
```

[2]you can also press Ctrl+N

[3]a vector is nothing more than a list of numbers, and a matrix is a table of numbers

To access individual elements in the vectors, we use:

```
b = a(1)
c = a(4)
```

There are many functions for visualizing data in Matlab. Lets try one that is called `stem`:

```
stem(a)
```

Lets generate a bit larger vector[4]:

```
a = 1:1000;
```

using `stem` for this kind of data is not so reasonable (try it). A better function for this case is `plot`:

```
plot(a)
```

Another method to generate vectors of increasing numbers is the function `linspace`. This function does in general not give integers, but gives additional control of the size and range of the vector. To generate a vector of 1000 values, ranging from the value $-\pi$ to $+\pi$, we type

```
t = linspace(-pi,+pi,1000);
plot(t);
```

To get the length, and maximum/minimum of a vector we use

```
max(t)
min(t)
length(t)
```

You can see this information in the workspace window as well.

2.1 Indexation

Using regular integers for indexation we can get isolated values such as `t(1)` or `t(27)`. A useful matlab property is the ability to index vectors in various other ways. Lets say we have a small vector of integers, we can use it to index other vectors directly:

```
indices = [1, 600, 2];
t(indices)
```

The output of this indexation is itself a vector, of the same size as the indices vector. It is often most convenient to define the `indices` vector on the same line as it is used. Here are a few examples of indexation of that type(file: 'indexation1.m'):

[4]dont forget to put the semi-colon at the end, or Matlab will clutter your command window

```
t1 = t([1, 600, 2]);                   %same as the previous example
t2 = t(1:  10);                        %the first 10 elements of t
t3 = t(1:  1000);                      %the same vector, unchanged
t4 = t(1:2:1000);                      %every second element of t
t5 = t(1000:-1:1);                     %flipped order

plot(t);
hold on;                               %allows many plots in the same figure
plot(t4,'r');                          %plot in red   ('r')
plot(t5,'g');                          %plot in green ('g')
legend('original t', ...
       'every second element', ...
       'flipped order');
```

One of the many interesting things we can do with indexation of this kind, is to assign values to subsets of vectors. Consider the following ('indexation2.m').

```
a = 1000:-1:1;                             %vector of 1000 elements
plot(a);                                   %plot before we modify it
hold on;

a(1:200)   = 800;                          %change the first 200 elements
a(801:end) = 200;                          %change the last  200 elements

plot(a,'--r')                              %plot in dashed red after modification
legend('original vector','modification')
```

In the above example we assign a single value to occupy many elements in the vector. We can also assign a range of values in the following way('indexation3.m'):

```
a = 1000:-1:1;                              %vector of 1000 elements
plot(a);                                    %plot before we modify it
hold on;

a(1:200)   = 601:800;                       %change the first 200 elements
a(801:end) = 201:400;                       %change the last  200 elements

plot(a,'--r')                               %plot in dashed red after modification
legend('original vector','modification')
```

In this second approach, we have a vector being assigned as a subvector of another vector. This can only be done if the vector we wish to assign fits exatly into the subvector we indicate. The following lines generates errors.

```
a(1:200)   = 600:800;                %error, right side has 1 more element then left side
a(1:201)   = 601:800;                %error, left side has 1 more element then right side
```

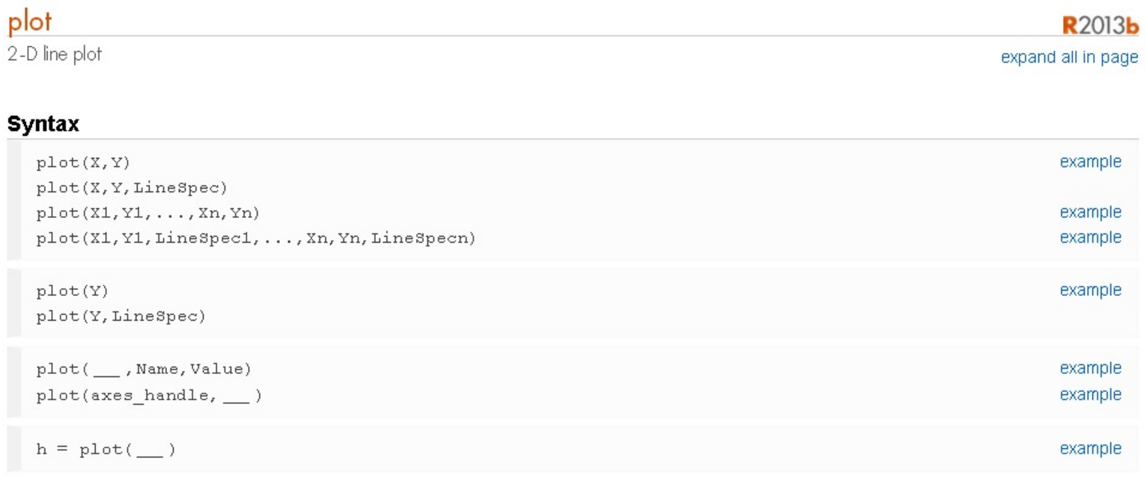
2.2 Elementwise operations

Most matlab functions are defined for matrices and vectors. Standard math functions work elementwise, so that it is extremely easy to apply the same operations on many values. If a Matlab function operates elementwise, then the output is of the same size as the input. A good example is the `sin` function. Lets investigate this now:

```
s1 = sin(t);
plot(s1);
```

This should display one period of a sin-wave. While the values of the function (y-axis) seems correct (ranging in -1 to +1) it seems the x-axis values are incorrect. On the x-axis the range is from 0 to 1000 and not $-\pi$ to $+\pi$. This is the sampling index(the index number of the vector element, and we have 1000 elements in `t`).

2.3 More on Plotting


plot
2-D line plot

Syntax

```
plot(X,Y)
plot(X,Y,LineSpec)
plot(X1,Y1,...,Xn,Yn)
plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)

plot(Y)
plot(Y,LineSpec)

plot(___,Name,Value)
plot(axes_handle,___)

h = plot(___)
```

Figure 2: Documentation on plot

At this point, we would like to check out how the `plot` function works, and see if there is an easy way to get the range of the x-axis as we would like it. Check built-in documentation of Matlab by typing:

```
doc plot;
```

giving the contents of figure 2.

The function is quite versatile, but we won't get into all its uses here. We see usage no. 5 in fig. 2 being the easiest. This is simply supplying Y values in a vector and it is how we have used it so far. We can also see usage no. 1, where we can supply X values in addition to Y. If you scroll down in the help section, you can get all the details on how its used. Lets take option 1 for now, and supply the X values as follows:

```
s1 = sin(t);
plot(t, s1);
```

This put the range of the x-axis correct. To put a title on the plot:

```
title('A plot of sinus');
```

2.4 Plotting geometric shapes

Because we have full control of the coordinates of the plotted lines in `plot` we can draw full geometric shapes(not only 1D functions as we have done so far). To draw a square with one corner in the origin, we just define coordinates of 4 points. We should add the starting point of the graph at the end of the list, so the graph is closed:

```
x = [0, 1, 1, 0, 0];                   %x coordinates for square
y = [0, 0, 1, 1, 0];                   %y coordinates for square
plot(x,y);
```

The resulting square is drawn at the very edges of the graph(it looks more like a frame than a graph), and may be difficult to see. This is because Matlab puts the limits on the axis to be as small as possible while still containing the graph. To put the axis limits manually:

```
xlim([-1, 2]);
ylim([-1, 2]);
```

The shape is now more visible closer to the centre of the figure, but it is not a square but rather a rectangle. We can see that matlab has not scaled the axes equally. We can set it to do that by:

```
axis equal
```

and sometimes we want a grid for visibility:

```
grid on
```

To draw a cirle, we simply define the x and y coordinates;

```
%make x and y coordinates of a circle
x = cos(t);
y = sin(t);
plot(x, y);
axis equal;
```

We can draw many circles simultaneously, and let them vary in radius('circles.m'):

```
%several cirlces, different radius
plot(    cos(t),     sin(t), ...
     0.8*cos(t), 0.8*sin(t), ...
     0.6*cos(t), 0.6*sin(t), ...
     0.4*cos(t), 0.4*sin(t));

title('many circles, different radius');
legend('1','.8','.6','.4');
```

We recall that the variable `t` is a vector of 1000 elements, linearly spaced between $-\pi$ to $+\pi$. Thus, even though our plotted circles look smooth, they are actually 1000 line segments. We can reduce the number of line segments used by decreasing the size of the `t` vector(generating a new `t` vector to replace the old one). Another way is to use the `t` vector we already have, but to use only a few of the values we have in the vector. This can be done by using vector indexation:

```
%reduce size of t:
t2 = t(1:100:end);                       %'end' gives the end of allowed indexation
plot(cos(t2), sin(t2));
```

In general: more samples → better shape.

For such small amounts of sample points on the circle, we may want to explicitly mark the positions of the points. For this, and several other graphical options, we can use a `LineSpec` argument to `plot` (recall the many options we have in using `plot` in fig. 2). A `LineSpec` is an additional argument consisting of a string of characters, indicating the appearance of the plot. Lets make use of the string `'r--o'`, indicating a red line ('`r`') that is dashed('`--`'), with a ring marker for the sample points('`o`'):

```
plot(cos(t2), sin(t2),'r--o');
```

You may also try out the lineSpecs: `'-.x'`, `'m-x'`, `'x'`.

2.5 Example application: audio

Sampled sound is a long vector of values and it is common with very large vectors in audio processing. For good quality sound, we need to sample at very high rates, sometimes as high as 50 000 times per second (50 kHz). Sampling 1 second of audio at 50 kHz results in a vector that is 50 000 elements long. Lets generate some audio data into our matlab workspace, but first we should clean up the workspace. Type

```
clear                                %kills all your variables in workspace
```

Now, generate some audio synthetically as a single sinus wave:

```
Fs = 10000;                           %sampling frequency we use
t  = linspace(0,2,Fs*2);              %2 seconds long vector
y  = sin(t*4000);                     %generate audio with frequency 4k

sound(y ,Fs);                         %play audio
```

We generated this data ourselves, so we know exactly what the samples look like:

```
plot(t(1:200), y(1:200));             %plot the first 200 samples
ylim([-3,3]);                         %set limits of the y-axis
title('pure frequency audio');
```

We can change the intensity, the tone (frequency) and just about anything we like this way:

```
F = 6000;                                       %base tone

      %Make vectors of different tones and duration
y1 = sin( t(1:4000)*F     );
y2 = sin( t(1:4500)*F*1.12);
```

```
y3 = sin( t(1:5500)*F*0.89);
y4 = sin( t(1:4500)*F*0.45);
y5 = sin( t(1:6500)*F*0.67);

      %play some music:
sound([y1 y2 y3 y4 y5],Fs);
```

Now, to load some audio[5]:

```
clear;
load chirp;                   %loads new audio vector "y", and variable "Fs"
sound(y ,Fs);
```

You can see that your workspace now contains a new vector `y` (1329 elements long), and variable `Fs` $= 8192$. We can plot the audio signal:

```
plot(y); title('bird chirp');
```

lets try another audio file:

```
load gong;                          %replaces audio vector "y", and variable "Fs"
sound(y ,Fs);                       %plays audio
plot(y); title('bell gong');        %plot the audio curve
```

3 Functions

We learned about scripts and the workspace. A script is nothing more than a collection of commands that are executed as you call the script. Writing each line of the script in the Matlab command window would yield the same result as running the script. A script can change the workspace content in any way! Another kind of m-file, is one that contains a function. A function has its own local workspace, which is created when you call[6] it, and disappears when its execution is finished. First clear the workspace

```
clear
```

... then make a new m-file:

```
edit myFirstFunc.m
```

As before, this will generate a blank m file for you. In order to make this into a function file, put the following contents into the newly created file

```
function myFirstFunc
 e = 1;
 f = 2;
 g = 3;
 h = e+f+g
```

[5]In this example we will load some standard Matlab sounds that are available. If you want to load your own audio from file, use the function 'wavread'

[6]When you type its name and hit enter in the command window

You may find this file very similar to the "myFirstScript" file generated in section 1. Like the script, we call the function file by typing its name on the Matlab command line:

```
myFirstFunc
```

Unlike the script file, calling the function file *did not generate any new variables in our workspace.* Not even the final variable `h` was created, although we could see the resulting value of it because we did not end the line with a semi-colon.

Functions are stand-alone, separate modules that calculate a result and/or perform some action when its called, and then leaves no trace of itself. Functions typically work on some input, and delivers some output. Lets change our definition of our newly created function to gather input, and generate output:

```
function h = myFirstFunc(e,f,g)
 disp('Hello World, from myFirstFunc');
 h = e+f+g;
```

This very simple function now does 2 things:

1) takes 3 input values and outputs their sum, and...

2) displays the text: "Hello World...".

To call this function we need to supply it with some input, which we do by:

```
L = myFirstFunc(1,2,3);
```

notice that this creates a new variable `L` in your workspace that will contain the value from your function.

Now, lets make a more reasonable function, that might actually start to look useful. Lets say we want a function to calculate the area AND perimeter of a triangle. As input, we shall take the length of three sides, as illustrated in figure 3. When all sides are given in a triangle, we can use Heron's Formula:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where s is half of the triangles perimeter, given by

$$s = \frac{a+b+c}{2}$$

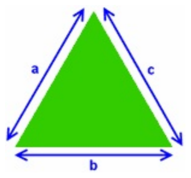


Figure 3: A picture from elementary school math

A function that implements this might look like:

```
function [A, P] = calcTriArea(a,b,c)
 P = a+b+c;                                 %The perimeter

 s = P/2;
 A = sqrt( s*(s-a)*(s-b)*(s-c) );           %Heron's formula
```

A way to test this is to take some simple triangle examples. A right angled triangle with area 0.5 has sides $a = b = 1$, and $c = \sqrt{2}$, which we can use our newly developed function to test:

```
[A,P] = calcTriArea(1,1,sqrt(2))
```

3.1 Functions and their arguments

A Matlab function can be called with any number of input arguments, and any number of output arguments. Sometimes, a function NEEDS a certain number of inputs, or its behavior is not defined. Try for example:

```
[A,P]   = calcTriArea(1,1)                       %error, not enough inputs
[A,P,D] = calcTriArea(1,1,sqrt(2))               %error, too many outputs
[A,P]   = calcTriArea(1,1,sqrt(2),2,3,4)         %error, too many inputs
```

These cases will generate errors, because we have the wrong number of arguments for the function. Note however that these cases work:

```
A = calcTriArea(1,1,sqrt(2))                     %1 output
    calcTriArea(1,1,sqrt(2))                     %0 output
```

Sometimes we wish to extend functions for other number of input and output arguments. For this purpose, we have access to the number of arguments supplied through the Matlab keywords `nargin` and `nargout`[7]. `nargin` and `nargout` are variables that may only be used inside of functions, and contain the number of inputs and output arguments respectively. Lets extend our function `calcTriArea` to handle the case of 2 inputs in the following way:

```
function [A, P] = calcTriArea(a,b,c)
  if nargin == 2                          %if we have  2 inputs ...
      c = 1;                              %put c to default value
  end                                     %end of the "if statement"

   P = (a+b+c);
   s = P/2;
   A = sqrt( s*(s-a)*(s-b)*(s-c));
```

After we have redefined our function, we can use it without error in the following manner:

```
[A,P]   = calcTriArea(1,1)
```

[7]number of arguments in/out

Tasks

Solutions that do not fill the following requirements **EXACTLY, PRECISELY AND TO THE LETTER** will not be considered:

- Send your solutions by email to:

  stefan.karlsson@hh.se

  subject: Matlab, Exercise `X`, `YourNames`

- Send all the files that are requested, no more and no less, in one single zip file per exercise, with NO sub-folders in the zip file. All the files, for all the tasks should be bundled into one zip file.

- Put the Names of the authors, in remarks, at the top of every m-file.

- Send the solutions within 2 weeks of every exercise session. That is, you have a two week deadline to hand it in.

Task 1

In the function `calcTriArea` (as defined in section 3.1) There is no logical reason to have the default value of `c` to be 1. Lets do something more intelligent with `calcTriArea`, by making it handle right sided triangles in the case when 2 arguments are provided. When two 2 inputs are supplied, we assume it's a right angled triangle, and the 2 arguments are the legs (catheti). Make a new function `e1_1.m` which does this.

To try this new function, we can use a well known right-sided triangle with sides 3, 4 and 5:

```
a = 3; b = 4; c = 5;
[A1,P1] = calcTriArea(a,b,c)
[A2,P2] = e1_1(a,b)
```

You should verify that

```
A1 == A2  %statement gives value 1, if true, 0 if false
P1 == P2
```

Hand in only the function file `e1_1.m`.

Hints

Pythagoras

Task 2

Make a new function `e1_2.m` which is called in the following way:

```
[A,P] = e1_2(p1,p2,p3)
```

where `p1`, `p2` and `p3` are 3 points (vectors of length 2). The function should perform 2 tasks:

1. return the perimeter and area of the triangle that the three points define.
2. plot the triangle in a figure.

The function does not need to handle any special cases of inputs, assume three points are always given.

Hand in only the function file `e1_2.m`.

Hints

first calculate the sides of the triangle, then use the function from the previous task.

Task 3

The *script* `e1_3.m` should:

1. Let the user click 3 times in a figure, defining a triangle (`ginput`).
2. The area and perimeter of the triangle clicked should be displayed in a dialog box (`msgbox`);
3. The triangle should be drawn in a figure.

Hand in only the file `e1_3.m`.

Hints

- looking at `doc ginput`, the most appropriate use seems to be `[x,y]=ginput(n)`.
- get a dialog up, displaying the numerical values of `A` and `p`:

```
msgbox(['Area: ' num2str(A) ', perimeter: ' num2str(p)]);
```

Task 4

In section 2.5, a small piece of music was generated. The same piece of music should start playing when a function `e1_4.m` is called as one of the following syntax:

```
e1_4(a);
e1_4;
```

The default value for `a` is 6000:
The music should play 2 times when the function is called.

- First time : with a base tone equal to `a`.
- Second time: with a base tone equal to `(2/3)*a`.

Hand in only the file `e1_4.m`.

Hints

If you implement this correctly, it will sound pleasant. **Be warned**, there is a possibility of generating bugs that can make matlab run out of memory with this task. You may also by mistake set matlab to play sound for 10 minutes, that could annoy you and everyone around you. Restart Matlab if this happens, the m-file is saved automatically every time you run it.