Open your mind. LUT.
Lappeenranta University of Technology
LUT Machine Vision and Pattern Recognition 2018-09-03

BM40A0701 Pattern Recognition

Lasse Lensu

Exercise 1: Introduction to Matlab

Notes: This is a "crash course" about basic Matlab. The practicing can be done during the exercise session, that is, the "tasks" do not have to be done at home.

1. Introduction to Matlab (no points): Matlab is a numerical mathematics program that will be used in the exercises of this course. The following topics will be discussed. Examples are shown after the following list.

   (a) Starting, exiting, use of help
   (b) Inserting matrices and vectors, `zeros`, `ones`, `eye`, `a:b`
   (c) Accessing matrix elements, rows, and columns
   (d) Size of matrices, length of vectors
   (e) `who`, `whos`, `clear`
   (f) Basic arithmetic using scalars
   (g) Arithmetic for matrices (element-by-element and matrix arithmetic)
   (h) Matrix operations (transpose, inverse, eigenvalues and eigenvectors)
   (i) Elementary functions (trigonometric, exponential, manipulation of complex values)
   (j) Solving sets of linear equations
   (k) File operations (load and save, ascii and binary)
   (l) Control structures (if-then-else, while, for)
   (m) Function definitions and scripts
   (n) Debugging scripts
   (o) User input and output
   (p) Data analysis (maximum, minimum, mean, standard deviation, sum)
   (q) Plotting data
   (r) Dynamic Plotting
   (s) `find`

   Examples:

   (a) Starting, exiting, use of help
   In Windows, start Matlab from Start-menu.
   In Linux, start Matlab by typing

   ```
   brian:~> matlab
   ```

   Note: If you wish to use matlab without java interface start matlab using command

```
brian:~> matlab -nojvm
```

After Matlab has started you can begin typing commands at the prompt `>>`. The first command to know is `exit` which exits Matlab.

```
>> exit
```

To get the list of help topics, type

```
>> help
```

To get help on a single topic or function, use `help <topic>`.

```
>> help general
```

Command `lookfor` if useful if you are searching command but do not know exact command name.

```
>> lookfor integration
```

(b) Inserting matrices and vectors, `zeros`, `ones`, `eye`, `a:b`

You are free to use any string starting with a letter as a variable. Variables do not need to be declared. Matlab is case-sensitive, thus `a` and `A` refer to different variables.

To define a scalar variable

```
>> t=1
```

A matrix can be defined as follows

```
>> A=[1 2 3; 3 3 4; 3 4 5]
```

As you see, Matlab normally shows the result of each statement it executes. To hide the result, end the statement with a semicolon ";". Note also that vectors are just matrices with only one row or column

```
>> x=[0 2 4 6 8]
>> y=[1;3;5;7;9];
```

Several statements can be put on the same line, separated with "," or ";". You can also see the value of a variable by just typing its name

```
>> x,A
>> A;t
```

Matrices can be converted to vectors using `A(:)`

```
>> A(:)
```

Matrices with each element zero or one can be created with `zeros(n)`, `zeros(n,m)`, `ones(n)`, and `ones(n,m)`

```
>> zeros(2)
>> ones(3,5)
```

Identity matrix can be created with `eye(n)` or `eye(n,m)`

```
>> eye(4,5)
```

Vector [a, a+1, ..., b] can be created with `a:b`

```
>> 1:10
```

Vector [a, a+d, ..., a+i*d, b] can be created with `a:d:b`

```
>> 0:0.1:1
```

Check also the help of `linspace` and `logspace`.

(c) Accessing matrix elements, rows, and columns

To access matrix element (i, j), use `A(i,j)`. To access row i, use `A(i,:)`. For column j, use `A(:,j)`. These can also be used in setting the value of a variable.

```
>> A(2,2)
>> A(3,:)=[5 6 7]
>> A(:,1)
```

(d) Size of matrices, length of vectors

```
>> size(A)
>> length(x)
```

(e) `who`, `whos`, `clear`

Use `who` to list variables in the current workspace. `whos` gives the variables together with information about their size, bytes, class, etc. `clear` can be used to clear the workspace or a single variable.

```
>> who
>> whos
>> clear t
>> t
```

(f) Basic arithmetic using scalars

Standard addition, subtraction, multiplication, division and power can be used for scalars.

```
>> u=1.23;v=8.76;
>> [u+v u-v u*v u/v u^v]
```

(g) Arithmetic for matrices (element-by-element and matrix arithmetic)

Standard matrix addition, subtraction and multiplication can be used for vectors and matrices.

```
>> B=[8 7 6; 5 4 3; 2 1 0];
>> A+B
>> A-B
>> A*B
```

Matrix and scalar can be added, subtracted and multiplied.

```
>> -A+16
>> 2*A
```

Multiplication, division and power can be used element-by-element for matrices.

```
>> A.*B
>> B./A
>> A.^B
```

(h) Matrix operations (transpose, inverse, eigenvalues and eigenvectors)

`transpose(A)` is (almost) the same as `A'` which is the complex conjugate transpose. `inv(A)` is the inverse matrix. `eig(A)` returns eigenvalues and also eigenvectors if two output arguments are used.

```
>> transpose(A)
>> A'
>> inv(A)
>> eig(A)
>> [eigvec, eigval]=eig(A);
```

(i) Elementary functions (trigonometric, exponential, manipulation of complex values)

Standard trigonometric and exponential functions are available. For matrices, they compute element-by-element. `i` can be used as the imaginary unit. Real and complex parts can be obtained using `real` and `imag`. See also `help elfun`.

```
>> cos(x)
>> atan(A)
>> log(exp(A))
>> imag(sqrt(-1))
```

(j) Solving sets of linear equations

A set of linear equations $Ax = b$ can be solved using `x=A\b`, which is roughly the same as `x=inv(A)*b` except that it is computed using Gaussian elimination. Backslash operator `\` can also be used for computing the least squares approximation for over- or underdetermined system of equations.

```
>> b=[0;2;5]
>> A\b
>> C=[1 2 3 5 7;1 1 2 3 5]'
>> x=C\y
>> C*x-y
```

(k) File operations (load and save, ascii and binary)

Matrices can be read and written with `load <file>` and `save <file> [vars]`. Without vars-argument `save` operates on the entire workspace (all variables). Download the provided data file `balloon.mat` to your current directory before trying the following example.

```
>> load -ascii balloon
>> save x x
>> clear x
>> x
>> load x
>> x
```

Note that sometimes, especially when processing files within scripts with variable names you must give the names as strings and cannot use above method. Some commands like load and save also have function notation you can use so the above load command can be written as

```
>> fname = 'balloon';
>> load('-ascii',fname)
or
>> load('-ascii',[fname '.mat'])
```

(l) Control structures (if-then-else, while, for)

The general form of the `if` statement is

```
if <expression>
<statements>
elseif <expression>
<statements>
else
<statements>
end
```

`while`-loops can be constructed using the form below. Note that `break` can be used to exit the loop any time.

```
while <expression>
<statements>
end
```

`for`-loops are used to repeat statements a specific number of times.

```
for <variable>=<expression>,
<statements>
end

>> for k=1:5,
y(k)=1/k;
end
```

Note that you can also use commands `break` and `continue` within `for` and `while` loops.

(m) Function definitions and scripts

`function` command is used to add new functions. The commands and functions that comprise the new function must be put in a file whose name defines the name of the new function, with a filename extension of '.m'. At the top of the file must be a line that contains the syntax definition for the new function. A function can contain internal subfunctions but these cannot be called outside the file they are defined in. `return` statement can be used to force an early return.

Text editor can be launched with `edit` command. To be able to use your own .m-files, you must add the corresponding directory to Matlab's search path using `addpath <dir>`, e.g., `addpath '.'` adds the current directory to the path.

If you are running Matlab without Java engine (or just want to use another editor) on unix machines it can be invoked directly from matlab command line with !

```
>> !xemacs stat.m&
```

! allows any command line operations to be run from matlab (and & makes it a background process so it doesn't take up the command line).

The following should be put to `stat.m`

```
function [m,stdev] = stat(x)
% STAT   statistics
n = length(x);
m = avg(x,n);
stdev = sqrt(avg((x - m).^2,n));

function m = avg(x,n)
% MEAN subfunction
m = sum(x)/n;
```

A script file is an external file that contains a sequence of statements. By typing the filename, subsequent input is obtained from the file. Script files must also have the filename extension of ".m". Note that within scripts the current workspace is used and the script can use, alter or delete variables in the workspace, while the variables in a function are internal.

(n) Debugging scripts

It is usually a very good practise to write everything in script files even when experimenting so that you don't lose verything if matlab crashes. It is also possible to debug scripts. When script encounters command `keyboard` the execution stops and matlab provides command prompt. This allows normal command operations to be run from keyboard within script.

(So you can change values of variables or plot system status, etc…) Command `dbstep` executes one line of script and `dbcont` resumes the script execution. Test these by adding command `keyboard` on different lines in above stat.m script. You can also use command » `dbstop error` to start debuggin when any error is encountered. Or remove this state by » `dbclear error`.

(o) User input and output

`disp(x)` can be used to display information to user. The argument can be a variable name or a string within single quotes. `input('prompt string')` is used to prompt in the string and wait for user input. The input can be any valid expression which is evaluated using the variables in the current workspace and the result is returned. Note that inside a function the current workspace contains only the internal variables of the function but in a script, the whole user workspace can be used. To get the result as a string, use `input('prompt string','s')`. The following example should be put to a script file.

```
x=input('What should be evaluated?');
n = length(x);
m = mean(x);
stdev = sqrt(mean((x - m).^2));
disp(['The mean is ' num2str(m)])
disp('Standard deviation')
disp(stdev)
```

(p) Data analysis (maximum, minimum, mean, standard deviation, sum)

Row or columnwise maxima can be found in a matrix with `max(A,[],1)` or `max(A,[],2)`. Similarly, minima can be found and sums and means calculated with `min`, `sum`, and `mean`. The maximum (or minimum, sum, or mean) of a matrix can be found with `max(max(A))` or `max(A(:))`. Columnwise standard deviation can be calculated with `std`.

```
>> max(balloon)
>> min(A,[],2)
>> sum(sum(A))
>> mean(A)-std(B)
```

(q) Plotting data

Data vectors can be plotted using `plot`. `hold on` and `hold off` can be used to hold and release the current plot.

```
>> plot(balloon)
```

A increasing trend can be seen in the data. Actually, the data consist of 2001 observations taken from a balloon about 30 kilometres above the surface of the earth. In the section of the flight shown here the balloon increases in height. As radiation increases with height there is a non-decreasing trend in the data. There are outliers caused by the fact that the balloon slowly rotates, causing the ropes from which the measuring instrument is suspended to cut off the direct radiation from the sun. To better see the outliers, lets remove the linear trend from the data.

```
>> n=1:2001;
>> A=[1:2001; ones(1,2001)]';
>> x=A\balloon;
>> balloon2=balloon-A*x;
>> plot(balloon2)
```

Histogram of data can be calculated and visualized using `hist`. The outliers are clearly visible also in the histogram.

```
>> hist(balloon2)
```

(r) `Dynamic plotting`

Sometimes the state of the data alters in process and it may be useful to show this change as the processing progress. Example:

```
>> F = figure;
>> figure(F),for x = 0 : 200, plot(0:x,sin([0:x]/10)),drawnow,end
>> figure(F),for x = 0 : 200, plot(0:x,sin([0:x]/10)),pause(0.01),end
>> hold on
>> plot(-1*sin([0:200]/10),'r');
>> close(F)
```

(s) `find`

`find(X)` returns the indices of the vector X that are non-zero. It is a very useful command that requires some effort to master. For example, `k=find(A>100)`, returns the indices of A where A is greater than 100. Now lets continue our data analysis example. Remember that in Gaussian distribution over 99 % of the data is within two and a half standard deviations from the mean. Since we have removed the linear trend of the data, the mean of vector `balloon2` should be 0. We can use `find` to pick those elements of the vector `balloon` that lie within two and a half standard deviations. Lets also see how many elements are outside that range.

```
>> mean(balloon2)
>> balloon3=balloon(find(abs(balloon2)<2.5*std(balloon2)));
>> plot(balloon3)
>> length(find(abs(balloon2)>=2.5*std(balloon2)))
```

This is the end of the short tour to Matlab. Remember to try `help` always first when you encounter problems. You can also try command `demo` to run Matlab's demos. A comprehensive source of information is the Mathworks WWW-site at `www.mathworks.com`. For example, take a look at
http://www.mathworks.se/help/matlab/index.html

An open-source alternative to Matlab called Octave is also available. It has most of the same functionality, and when installed, it can be started with

```
brian:~> octave
```

Note that to use plotting, you need to have also gnuplot installed.

Additional files: `balloon.mat`.