



RESEARCH METHODS (MSC PROGRAMME), 2018

INTRODUCTION TO MATLAB 1

Programming in MATLAB: matrices, random numbers, conditional statements and loops, scripts and functions

Piotr Z. Jelonek

May 23, 2018

1 Introduction

Basic information

- MATRIX LABORATORY (MATLAB[®]) – commercial product, delivered by Math Works[®]
<http://uk.mathworks.com/products/matlab/>
- High-level script programming language, integrated with a user-friendly interface, allows for publication quality graphics
- Matrix-oriented – matrices are basic data structure, manipulating matrices is easy, many functions are **overloaded** to work with: numbers, vectors and matrices
- Designed for scientific computing, very precise
- Originally developed (mostly) for engineers, but currently very useful for econometrics and finance (has toolboxes for both)

Pros

- Extremely user friendly
- Labour efficient – you can do a lot just with a single line
- Easy access to your data, good graphical features
- Flexible, you can program your own solutions to non-standard problems (unlike programs where you have just a menu of options)
- Extremely popular, has a large community of users
- Much loved by macroeconomic community (‘*Dynare*’ toolbox for DSGE estimation)
- Everything can be done in one script: Econometrics, pricing Financial Instruments Statistics and Machine Learning, Optimization, Neural Networks, Signal Processing Wavelets, PDE’s, Curve fitting, Symbolic Math (and these are just few of the toolboxes you have access to under the Warwick University *TAH Student* license)

Cons

- Price
- Not integrated with database solutions (an alternative, tailored for business analytics is SAS:
http://www.sas.com/en_gb/home.html
(but that one is even more expensive, upside: employability)
- Not designed for very large data sets
- Since it is a high-level language, it is not a good choice if you need a code that is *lightning-fast*. It is optimized to work well on matrices, so operations on matrices and vectors are fast. But some other things – like logical operations – are slow

Alternatives

- Octave: free open-source MATLAB clone with some of its functionality and similar (often identical) syntax

<https://www.gnu.org/software/octave/>

- GAUSS: matrix-oriented, oldschool but still used by some econometricians, sold by Aptech Systems Inc.
- S-PLUS: object-oriented, programming language derived from S, sold by TIBCO Software Inc.
- R: a free open-source programming language, also derived from S, object-oriented environment for statistical computing

<https://www.r-project.org/>

capable of doing all statistics/econometrics, highly recommended (community, R-Journal)

Where to ask for help?

- Google: *how to do X in MATLAB* (usually you need < 30 s. to get your answer)
- Search documentation: *F1* redirects you to *MATLAB Help*, useful when you do not know how to use a toolbox, a MATLAB build-in function or how to generate a fancy 3D chart
- Community: over 225,000 contributors, thousands pieces of code available online, forums where you can ask for help

<http://uk.mathworks.com/matlabcentral/>

2 Interface

What I find useful

- In *Home* tab: *New* → *Script*, *New* → *Function*, *Save* → *Save as*
- In *Editor* tab: *Save* → *Save as* (to save a script or a function), *Run* (save and run, **F5**), *Comment* (**Ctrl+R**, after you have highlighted a part of script or a function) and *Uncomment* (**Ctrl+T**).
- *Command Window*: to input commands
- *Editor*: to edit scripts or functions
- *Workspace*: to look into all the variables, kept in memory, visualize them, plot them etc.
- *Current folder*: to manage your files and to check which of your functions MATLAB is currently able to 'see'
- *Help*: circled question mark in the top right corner (shortcut: **F1**), use it to explore the documentation (for MATLAB and its toolboxes)[drawback: it is massive]

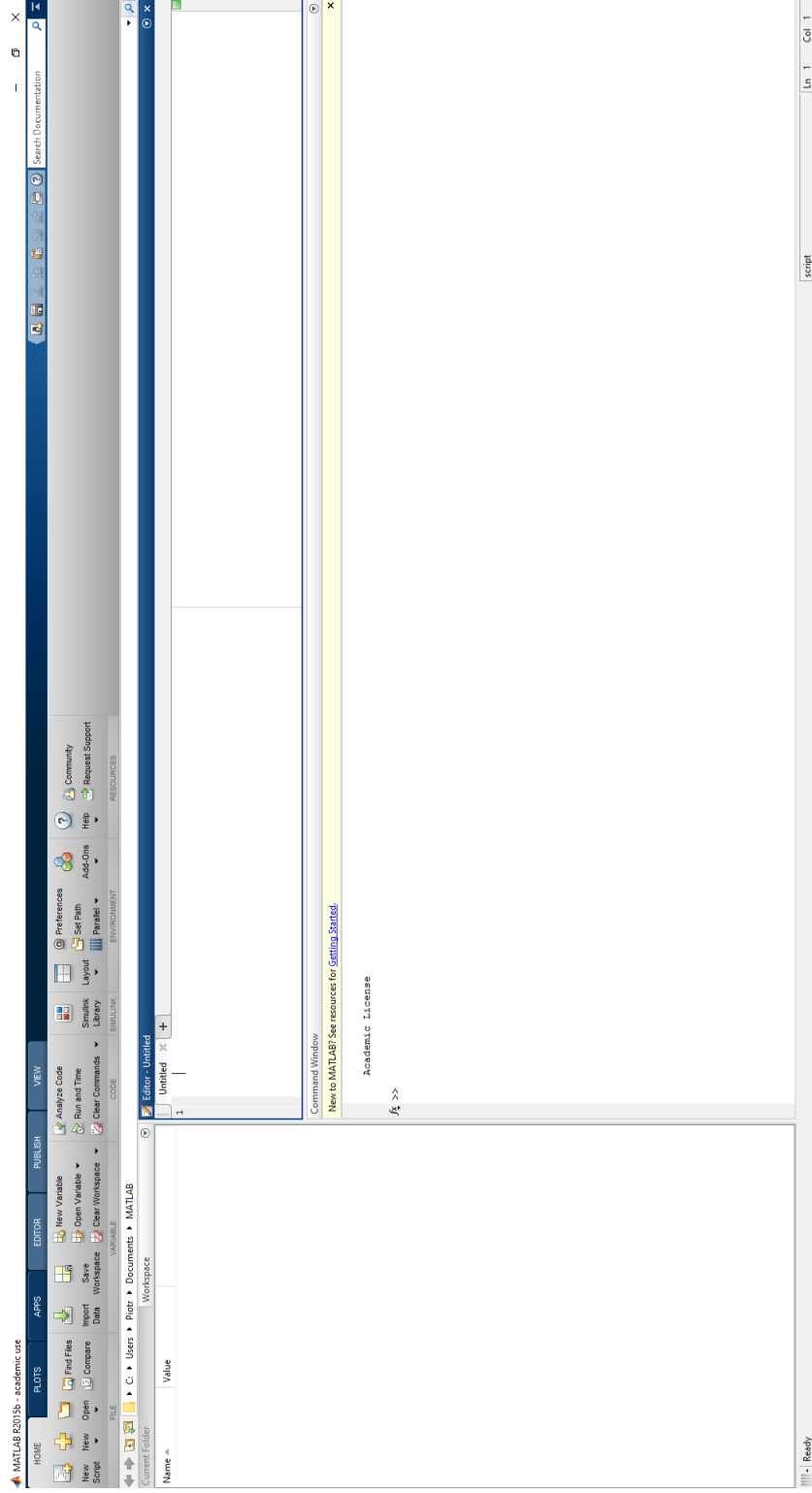


Figure 1: MATLAB interface: *Editor* and *Command Window* on the right, *Current Folder* (greyed out, inactive) and *Workspace* on the left. For the *Editor* tab to appear on the screen select *New Script*.

3 Useful tips

- There are two ways of working with MATLAB: either you input commands directly into the **command window** or you execute line by line a text file called a **script**
- The first option is useful when you want to see what a command does and if it is working as you have intended, the second option is handy when you want to run the entire analysis from start (loading and processing data) to finish (plotting charts, generating tables)
- Today will be using (mainly) the command line
- Percentage symbol ('%') denotes beginning of a comment – anything that follows is ignored by the interpreter and will **not** be executed
- Semicolon (;) at the end of a line suppresses the output, if you want to see the outcome of a given line – simply do not put it there
- On your keyboard **arrow up** brings up the most recent command, press it **again** to iterate through the commands previously executed
- MATLAB .m file is simply a text file - you can create a .txt file, change the extension to .m and it works
- Use **Ctrl+C** to interrupt the computation (if it takes too long or if you have entered an infinite loop)
- When you call a function, MATLAB searches for its definition in: current folder (first), toolboxes (next), among the build-in functions (finally). This search is continued until the first hit.
- Use command **help** *function_name* to learn more about a function, use **which** *function_name* to learn which version of a function is used (examples will follow)
- MATLAB distinguishes capital and small letters, so *A* and *a* are different variables

4 Programming in MATLAB

Copy the following bits of code to the MATLAB command window. Since none of these lines ends with a semicolon (;), the output will be printed on the screen.

4.1 Vector and matrix operations

defining a column vector: use **square** ([]) brackets, semicolon (;) denotes the end of a row:

```
b=[1; 2; 9; 18; 29; 47]
```

defining a row vector: use square brackets:

```
c=[1 4 5 -2 13 27]
```

vector of integers from 2 up to 7:

```
d=2:7
```

vector transposition:

```
b=c'
```

to access elements of a vector, use **standard** ('(')brackets:

```
x=b(4)
```

```
y=x*b(6)
```

take **sum()** of all the elements a vector:

```
s=sum(b)
```

take product (function: **prod()**) of all the elements a vector:

```
p=prod(b)
```

1×7 vector of ones:

```
z=ones(7,1)
```

cumulated sum of z (has the same dimension as z):

```
x=cumsum(z)
```

cumulated product of b (has the same dimension as b):

```
y=cumprod(b)
```

defining a matrix: use square brackets, semicolon (';') denotes the end of a row:

```
A=[1 2 3; 4 5 6; 7 8 9]
```

a 4×2 matrix of zeros:

```
z=zeros(4,2)
```

a 3×4 matrix of ones:

```
x=ones(3,4)
```

identity matrix, 3 ones on diagonal:

```
B=eye(3)
```

we may also have matrices of more than 2 dimensions, they are just harder to print:

```
C=zeros(2,2,2)
```

```
D=zeros(3,3,3)
```

to access elements of a matrix, use **standard** ('(')brackets:

```
x=A(1,1)
```

```
x=A(2,3)
```

matrix transposition:

```
A=A'
```

(matrix) multiplication:

```
C=B*A
```

inverse of matrix C (we need to define another matrix, the previous C is not invertible):

```
C=[1 2 1; 0 1 0; 0 0 2]
```

```
D=inv(C)
```

in matrix A select rows from 1st to 3rd, first column:

```
z=A(1:3,1)
```

in matrix A select rows from 1st to the last row, second column:

```
z=A(1:end,2)
```

in matrix A select rows from 1st to 2nd, columns from 2nd to 3rd:

```
z=A(1:2,2:3)
```

in matrix A select rows from 2nd to 3rd, columns from 1st and 3rd:

```
z=A(2:3,[1 3])
```

in matrix A select rows from 1st to 2nd, **all** the columns:

```
z=A(1:2,:)
```

substitute matrix B into matrix A :

```
B=[2 2; 2 2]
```

```
A(2:3,2:3)=B
```

matrix **concatenation**, C and D stacked horizontally (number of rows needs to match):

```
E=[C D]
```

matrix concatenation, C and D stacked horizontally (number of columns needs to match):

```
E=[C; D]
```

concatenating vector of ones to matrix D (number of rows needs to match):

```
E=[ones(3,1) D]
```

function **size()** for given matrix (e.g. E) outputs a vector:

```
size(E)
```

its first parameter is a number of **rows**:

```
rowsE=size(E,1)
```

its second parameter is a number of **columns**:

```
colsE=size(E,2)
```

matrix of ones, of the same dimensionality as E ($size(E)$ is a vector):

```
F=ones(size(E))
```

in MATLAB **Not a Number** (NaN) is a code for a missing value, it may be treated as a number:

```
A=[1 2 NaN; 4 5 NaN; 7 NaN 9; 10 11 12; 13 14 15]
```

we may take a **sum()** of values in A along the first dimension:

```
s=sum(A,1)
```

which is the same as the sum across columns of A :

```
s=sum(A)
```

sum of values in A along the second dimension:

```
s=sum(A,2)
```

which is the same as:

```
s=sum(A')
```


4.2 Random number generation

matrix of independent draws from a uniform distribution $U \sim U(0, 1)$:

```
U=rand(10,2)
```

matrix of independent draws from a uniform distribution $V \sim U(a, b)$:

```
a=2
b=3
V=(b-a)*rand(10,2)+a*ones(size(U))
```

random independent draws from standard normal distribution $Z \sim N(0, 1)$:

```
Z=randn(7,3)
```

random independent draws from normal distribution $X \sim N(\mu, \sigma^2)$:

```
mu=pi          % <- mu=3.1416...
sigma=2
Z=sigma*rand(4,3)+mu*ones(4,3)
```

random independent draws from log-normal distribution $\ln Y \sim N(\mu, \sigma^2)$:

```
e=exp(1)       % <- e=2.7182...
% dot power - a power taken element-by-element
Y=e.^Z
```

for independent draws from t-distribution with ν d.f. use *'random'* procedure:

```
nu=3
T=random('T',nu,3,2)
```

independent draws from gamma distribution with shape α and scale β :

```
alpha=1.5
beta=2.5
G=random('Gamma',alpha,beta,6,3)
```

for other distributions search documentation (*'F1'*) or use *help* command:

```
help random
```

generating dependent draws requires more complicated formulas (or other procedures).

4.3 Logical operations

all logical operators work on matrices, output is a binary matrix, let us check if $B > 2$:

```
B=[1 2 3; 4 5 6; 7 8 9]
B > 2
```

now check if $B > 2$ **and** (operator: '&') simultaneously $B \leq 7$:

```
(B > 2) & (B <= 7)
```

check if $B \leq 2$ **or** (operator: '|') $B \geq 5$:

```
(B <= 2) | (B >= 5)
```

check if B is **equal** to C (operator: '=='):

```
B
C=[0 -1 14; 15 -3 9; 21 8 9]
B==C
```

the **not** operator (operator: '~') negates binary values (zeros become ones, ones become zeros):

```
~(B==C)
```

check if **any** element of a vector is negative:

```
z=C(1,:)
id1=any(z<0)
```

and if **all** elements of a vector are negative :

```
id2=all(z<0)
```

we may use *true* and *false* to indicate which rows (2nd, 3rd) and columns (2nd) we want:

```
D=C([false true true],[false true false])
```

the same done with **logical()** function, converting binary to logical values (*true*, *false*):

```
D=C(logical([0 1 1]),logical([0 1 0]))
```

use **isnan()** logical function to identify which data in the matrix is missing:

```
id=isnan(A)
```

function **all()** identifies columns with no missing values:

```
id1=all(~isnan(A))
```

we can do the same with function **any()**:

```
id1=~any(isnan(A))
```

take transposition of A to identify rows with no missing values:

```
id2=all(~isnan(A'))
```

the same with function **any()**:

```
id2=~any(isnan(A'))
```

removing columns with missing values from matrix A :

```
B=A(:,id1)
```

removing rows with missing values from matrix A :

```
B=A(id2,:)
```

4.4 Conditional statements and loops

in conditional statement **if** a list of commands is executed if a logical condition is being met:

```
a=2;
if a>=2
    a=a+1
end    % <- end of 'if'
```

conditional statements can also take into account an alternative:

```
a=1;
if a>=2
    a=a+1 % <- now this line is not executed
else
    a=a-1
end      % <- end of 'if'
```

or a couple of alternatives:

```
a=5;
if a <= 2
    a=a+1 % <- now this line is not executed
elseif 2 < a & a <=4
    a=a-1 % <- this line is not executed
elseif a>4
    a=(a-1)^2
end      % <- end of 'if'
```

in the **for** loop a list of commands is executed for value of index (e.g. i) in a given range (e.g. from 1 up to 10), so we know how many times the command(s) will be run:

```

for i=1:10
    i % <- this command will be executed
end % <- end of 'for' loop

```

execute the command(s) for value of index i from m up to n :

```

m=3; % <- now this is parameter
n=11;
id=zeros(n,1);
for i=m:n
    id(i)=i;
end % <- end of 'for' loop
id'

```

in the **while** loop a list of commands is executed as long as a logical condition (e.g. expressed by binary variables) is met, so the command(s) will be run as many times as necessary:

```

b=1
while b<5
    b=b+2 % <- this command will be executed
end % <- end of 'while' loop

```

while loop with a binary variable expressing logical condition:

```

b=0;
flag=1;

while flag
    b=b+1;
    if b>5
        b
        flag=0
    end % <- end of 'if'
end % <- end of 'while' loop

```

an **infinite** loop (which will freeze your computer, press **Ctrl+C** to exit):

```

flag=1;
a=0;
while flag % <- same as (flag==1), always true
    a=a+1
end % <- end of 'while' loop

```

using while loop to obtain draws from a censored t-distribution with $\nu = 3$ df.:

```

n=1000; % <- that many draws I want
got=0; % <- draws I already have
nu=3; % <- number of degrees of freedom
thr=-3.182; % <- a 2.5 percent critical level

```

```

ct=zeros(n,1);
while got<n
    t=random('T',nu,n-got,1); % <- we still need (n-got) draws
    t=t<thr,1); % <- I retain only values, smaller than thr
    m=size(t,1);
    ct(got+1:got+m,1)=t;
    got=got+m;
end % <- end of while loop

es=(1/n)*sum(ct) % <- expected shortfall ES_T(thr)

```

the output of the last loop is the *expected shortfall*, a risk measure which is an expected loss, given that the loss has already exceeded a given threshold.

4.5 Scripts and functions

To create a script select *New* → *Script*, once the new script file is open select *Save* → *Save as* to save it to your desktop (save it as: *'myscript.m'* or whatever you fancy). Now when you will run the script (from desktop), MATLAB will be able to 'see' all the functions saved in the same location. Running code as a script is often more convenient than running it from the command window. You may copy the following bits of code to separate files and run them as scripts. In a script it is often convenient to add the following as the first line:

```
clear; clc;
```

The first command clears the memory. The second clears the output window. In your script you may also want to add extra comments.

let us compare what is faster – a for loop or a cumulated sum, this code is provided as *'example1.m'* script:

```

clear; clc;
n=100000;
m=100;
z=randn(n,1);

tic
for i=1:m
    id=zeros(n,1);
    id(1,1)=z(1,1);
    for j=2:n
        id(j,1)=id(j-1,1)+z(j,1);
    end
end
toc

tic
for i=1:m
    id=cumsum(z);

```

```
end
toc
```

(clearly, using `cumsum()` is a lot faster – this is no wonder since this is a procedure operating on matrices, already optimized)

finding a moving average and moving variance of random draws:

```
n=100000;
m=100;
z=randn(n,1);
ma=zeros(n-m+1,1);
mv=zeros(n-m+1,1);

tic % <- here we start the clock
for j=m:n
    ma(j-m+1,1)=(1/m)*sum(z(j-m+1:j,1));
    mv(j-m+1,1)=(1/(m-1))*(z(j-m+1:j,1)'*z(j-m+1:j,1)-m*ma(j-m+1,1)^2);
end
toc % <- output time
```

moving average and moving variance (level: advanced, useful for fast codes):

```
n=100000;
m=100;
z=randn(n,1);
ma=zeros(n-m+1,1);
mv=zeros(n-m+1,1);

tic % <- here we start the clock
ma(1,1)=sum(z(1:m,1));
mv(1,1)=z(1:m,1)'*z(1:m,1); % <- scalar product, vector operations are fast
for j=m+1:n
    ma(j-m+1,1)=ma(j-m,1)+z(j,1)-z(j-m,1);
    mv(j-m+1,1)=mv(j-m,1)+z(j,1)^2-z(j-m,1)^2;
end
ma=(1/m)*ma;
mv=(1/(m-1))*(mv-m*ma.^2);
toc % <- output time
```

we may put both codes to a single script to conveniently compare which one is faster, this example is provided as `'example2.m'` script:

```
clear; clc;
n=100000;
m=100;

z=randn(n,1);
ma=zeros(n-m+1,1);
```

```

mv=zeros(n-m+1,1);

tic % <- here we start the clock
for j=m:n
    ma(j-m+1,1)=(1/m)*sum(z(j-m+1:j,1));
    mv(j-m+1,1)=(1/(m-1))*(z(j-m+1:j,1)'*z(j-m+1:j,1)-m*ma(j-m+1,1)^2);
end
toc % <- output time

tic % <- here we start the clock
ma(1,1)=sum(z(1:m,1));
mv(1,1)=z(1:m,1)'*z(1:m,1); % <- scalar product, vector operations are fast
for j=m+1:n
    ma(j-m+1,1)=ma(j-m,1)+z(j,1)-z(j-m,1);
    mv(j-m+1,1)=mv(j-m,1)+z(j,1)^2-z(j-m,1)^2;
end
ma=(1/m)*ma;
mv=(1/(m-1))*(mv-m*ma.^2);
toc % <- output time

```

let us make the code from the previous section, the one which calculates the Expected Shortfall, more general; I would like it to work for a **vector** of different threshold values – so I will need one more loop; this example is provided as *'example3.m'* script:

```

clear; clc;
n=1000; % <- that many draws I want
nu=3; % <- number of degrees of freedom

% now we will repeat the calculation for a vector of values
THR=[-1.638 -2.353 -3.182 -4.541 -5.841] % <- critical levels

colsTHR=size(THR,2);
ES=zeros(size(THR));
ct=zeros(n,1);

for i=1:colsTHR
    got=0; % <- draws I already have
    while got<n
        t=random('T',nu,n-got,1); % <- we still need (n-got) draws
        t=t(t<THR(i),1); % <- I retain only values, smaller than THR(i)
        m=size(t,1);
        ct(got+1:got+m,1)=t;
        got=got+m;
    end % <- end of while loop
    ES(i)=(1/n)*sum(ct); % <- expected shortfall ES_T
end

ES

```

Functions from given inputs produce a vector of outputs. Definition of a function needs to be preceded by a keyword **function** and ended by a keyword **end**. If you select *New* → *Function*, a new tab in the editor will open with the following default contents:

```
function [ output_args ] = Untitled1( input_args )
%UNTITLED1 Summary of this function goes here
% Detailed explanation goes here
end
```

Above *input_args* is a list of parameter(s), separated by commas, *[output_args]* is a vector (as you might have guessed) of outputs. Elements of this vector can be basically anything – numbers, matrices, strings, cells (the last two we have not covered). Procedures are usually saved in separate *.m* files, where name of the file is the same as the name of the procedure.

A function which calculates the Newton symbol (n choose k):

```
function c=my_nchoosek(n,k)

% This function computes n choose k in an efficient (non - matlab) way
%
% INPUT
% n -integer
% k -integer, lesser or equal to n
%
% OUTPUT
% c - integer, 'n' choose 'k'

if (n<=1)|(n==k)|(k==0)
    c=1;
else
    k=max(k,n-k);
    c=prod(k+1:n)/prod(1:n-k);
end
```

let's check which is faster – this one or the standard MATLAB **nchoosek()** function – this example is provided as *'example4.m'* script:

```
clear; clc;
m=1000;
n=70;
k=35;

tic
for i=1:m
    nchoosek(n,k);
end
toc
```



```
tic
for i=1:m
    my_nchoosek(n,k);
end
toc
```

to locate the MATLAB code for the **nchoosek()** function use:

```
which nchoosek
```

if you look into this function – it is very complicated, this is because it also works on vectors. The reasons it is so slow – it plots output to the screen (which slows things down, since screens need to be refreshed), and does a lot of logical operations (which are slow in MATLAB). Apart from that – it is badly coded.

4.6 Data basics

One of the easiest ways is to load the data from MS Excel (but you can also import it from almost anything else, like STATA). To read the data from .xlsx file use the **xlsread()** procedure. Since we do not know the syntax, we will use help:

```
help xlsread
```

the easiest way to read the data (just numeric values):

```
A = xlsread('it.xlsx');
```

extracting the data from matrix *A* (I want it in chronological order, **flipud()** function flips the vector):

```
g=flipud(A(:,1)); % <- Alphabet Inc. prices, first column of A
a=flipud(A(:,2)); % <- Apple Inc. prices, second column of A
```

logarithms (element-by-element):

```
log_g=log(g)
```

1st order lags:

```
lag_g=g(2:end)
```

1st order differences:

```
d_g=g(1:end-1)-g(2:end);
```

log-returns:

```
r_g=log(g(1:end-1))-log(g(2:end));
r_a=log(a(1:end-1))-log(a(2:end));
```

simple returns (here './' is element-by-element division):

```
R_g=(g(1:end-1)-g(2:end))./g(2:end)
```

Operators preceded by a **dot** are executed element-by-element. The codes for simple data processing are provided in one script as *'example5.m'*.

4.7 Plotting charts

To learn more about the basic `plot()` function (it is **overloaded**: may be called in a number of ways with different sets of parameters), use:

```
help plot
```

basic data plot (lazy version):

```
plot(g) % <- solid blue line by default, note mess with x axis
```

scatter plot:

```
scatter(r_a,r_g)
```

Alphabet Inc. log-returns, sorted according to the first column, in ascending order:

```
sortrows(r_g,1)
```

quantile-to-quantile (Q-Q) of Apple Inc. log-returns versus Alphabet Inc. log-returns:

```
plot(sortrows(r_a,1),sortrows(r_g,1))
```

two (simple) plots on one graph (red solid line for Alphabet Inc., black dashed for Apple Inc.):

```
x=cumsum(ones(size(a))); % <- time count
plot(x,a,'k--',x,g,'r-')
xlabel('TIME (IN TRADING DAYS)') % <- label on the x axis
ylabel('PRICE (ADJ. CLOSE, IN USD)') % <- label on the y axis
legend('Apple Inc.','Alphabet Inc.','Location','NorthWest') % <- legend
axis tight % <- axis tightened to the largest observation
```

the same, but with on the top of each other:

```
x=cumsum(ones(size(a))); % <- time count
hold on % <- from now on stack charts one on another
plot(x,a,'k--') % <- black dashed line for Apple Inc.
plot(x,g,'r-') % <- red solid line for Allphabet Inc.
hold off % <- from now on plot charts as separate
xlabel('TIME (IN TRADING DAYS)')
ylabel('PRICE (ADJ. CLOSE, IN USD)')
legend('Apple Inc.','Alphabet Inc.','Location','NorthWest') % <- legend in top left
axis tight % <- axis tightened to the largest observation
```

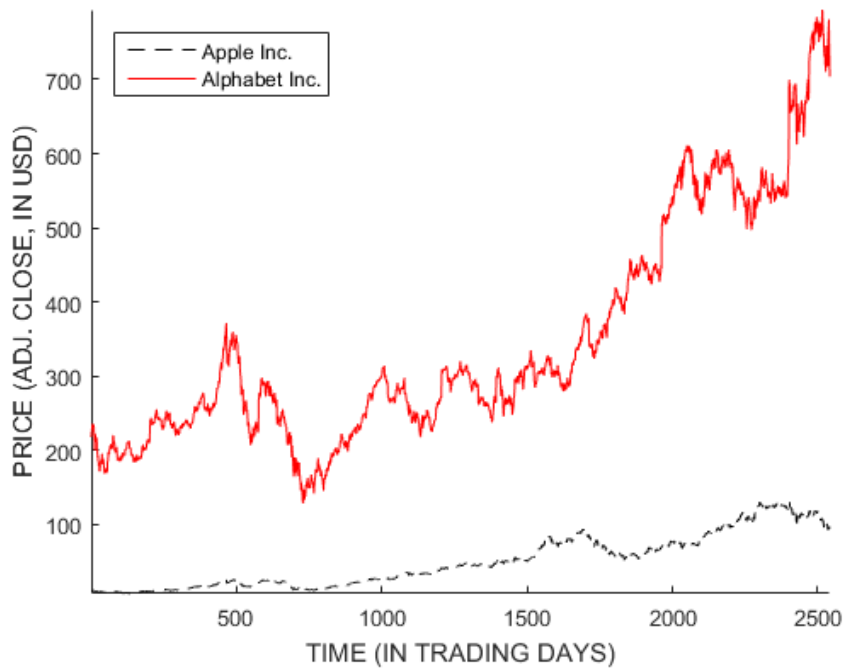


Figure 2: Multiple series on a single chart.

charts in multiple windows:

```
figure % <- initialization, required for subplot() command

subplot(1,3,1) % <- display graphs in 1 row, 3 columns; plot 1
x=cumsum(ones(size(a)));
plot(x,g,'b') % <- (different) blue line, colid by default
xlabel('TIME (IN TRADING DAYS)')
ylabel('ALPHABET PRICE (ADJ. CLOSE, IN USD)')
title('STOCK PRICE') % <- title
axis tight

subplot(1,3,2) % <- display graphs in 1 row, 3 columns; plot 2
x=cumsum(ones(size(r_a)));
plot(x,r_g,'m') % <- magenta line (solid by default)
xlabel('TIME (IN TRADING DAYS)')
ylabel('ALPHABET RETURNS')
title('LOG RETURNS')
axis tight

subplot(1,3,3) % <- display graphs in 1 row, 3 columns; plot 3
x=cumsum(ones(size(r_a)));
z=sortrows(randn(size(r_a)),1)
r_g_std=(r_g-mean(r_g))./std(r_g); % <- mean() returns sample mean, std() - std. dev.
scatter(z,sortrows(r_g,1),'c+') % <- use cyan '+' as marker
```

```

xlabel('QUANTILES OF STD. NORMAL')
ylabel('QUANTILES OF STD. R\F')
title('Q-Q PLOT')
axis tight

```

This is useful when to evaluate quality of your results you need to simultaneously asses a number of charts – then you can do it quickly, in a single glance. All charts can be easily saved (I recommend the .png format, these files are small and do not loose quality).

Note: charts may save differently, depending on whether you save them directly or from the full screen view (one of these two may look better). In publication quality graphics do not use graph titles since they do not scale well – instead I recommend using captions.

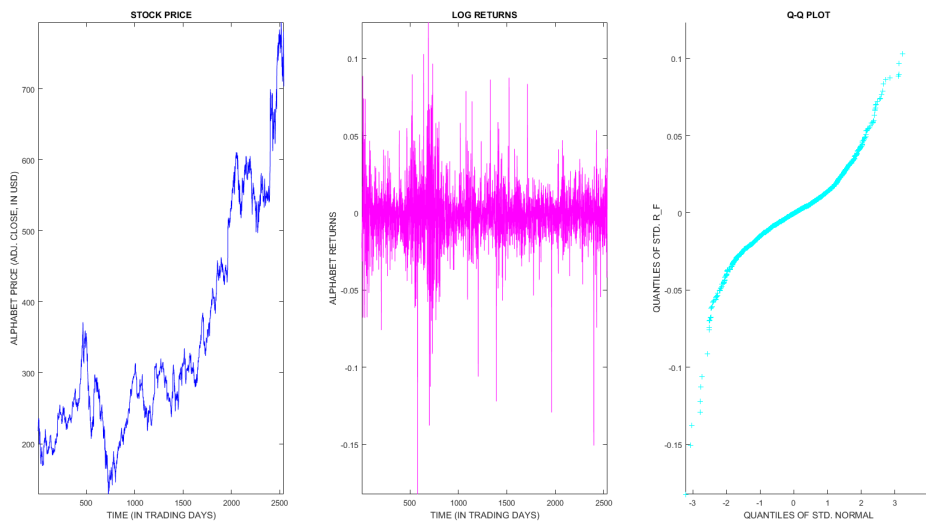


Figure 3: Different series in multiple windows.

5 Addendum: Generating pseudo ex-post forecasts

In the *Home* tab select *Preferences*, go to *MATLAB* → *Editor/Debugger* → *Display*, make sure that the *Show line numbers* option is ticked, confirm with *OK*. Pseudo ex-post forecasting script:

```

clear; % <- clear the memory
clc; % <- clear the command/output window

% DEFINE PARAMETERS
% forecats horizon (in periods)
h=5;

% number of forecasts
nf=250;

```

```

% rolling window length
w=100;

% lag order
l=1;

% scaling/stratching factor (for plotting charts)
scale=1.05;

% LOADING DATA
A = xlsread('it.xlsx');

% reverse the order of the data
g=flipud(A(:,1));
a=flipud(A(:,2));

% find log-returns
r_a=log(a(1:end-1))-log(a(2:end));
r_g=log(g(1:end-1))-log(g(2:end));

% data length
n=size(r_a,1);

% HERE WE WILL SAVE FORECASTS AND BENCHMARKS
fcast=zeros(nf,h);
bmark0=zeros(nf,h);
bmark1=zeros(nf,h);

% FORECASTING
for i=1:nf; % <- for a required number of forecasts

    first=n-(nf-i+1)-w-h+1;
    last=n-(nf-i+1)-h;
    y=r_a(first:last);

    for j=1:h % <- for up to h forecast horizons

        % BENCHMARK - AR(1)
        % select the data
        x=[ones(w,1) r_a(first-(h-j+1):last-(h-j+1),1)];
        % ols estimation
        b=(x'*x)\x'*y; % <- the same as: b=inv(x'*x)*x'*y;
        % take the last row of x's, multiply by betas
        f=x(end,:)*b;
        % save forecasts
        bmark1(i,j)=f;
    end
end

```

```

% MAIN MODEL - ARX
% select the data; for l=1 the term (l-1)below can be removed
x=[ones(w,1) r_a(first-(h-j+1)+(l-1):last-(h-j+1)+(l-1),1)...
r_g(first-(h-j+1)+(l-1):last-(h-j+1)+(l-1),1)];
% ols estimation
b=(x'*x)\x'*y;
% take the last row of x's, multiply by betas
f=x(end,:)*b;
% save forecasts
fcast(i,j)=f;

    end
end

% HERE WE SAVE ACTUAL VALUES
actual=zeros(nf,h);

for i=1:nf; % <- for a required number of forecasts
    last=n-(nf-i+1)-h;
    for j=1:h % <- for up to h forecast horizons
        actual(i,j)=r_a(last+j);
    end
end

% FORECAST ERRORS AND ACCURACY
% errors
bmark0_errors=bmark0-actual;
bmark1_errors=bmark1-actual;
fcast_errors=fcast-actual;

% standard deviations in different horizons
bmark0_std=zeros(1,h);
bmark1_std=zeros(1,h);
fcast_std=zeros(1,h);

% standard deviations in different horizons
for i=1:h
    bmark0_std(1,i)=std(bmark0_errors(:,i));
    bmark1_std(1,i)=std(bmark1_errors(:,i));
    fcast_std(1,i)=std(fcast_errors(:,i));
end

% forecast accuracy
fprintf(1,'Comparing accuracy of the model with AR(1)\n');
fcast_acc=fcast_std./bmark0_std
fprintf(1,'Comparing accuracy of the model with naive forecasts\n');
fmark1_acc=bmark1_std./bmark0_std

```

```

% FORECASTED ASSET PRICES
P0=a(end-h-nf+1:end-h);

fcast_prices=[P0 exp(fcast)]; % <- this is exponent element by element
for i=1:nf
    for j=2:h+1
        fcast_prices(i,j)=fcast_prices(i,j)*fcast_prices(i,j-1);
    end
end

% PLOTTING THE FORECASTS
plotfcast(a(end-300:end),fcast_prices,scale)

The definition of plotfcast() function (saved as: 'plotfcast.m' file):

function plotfcast(y,fcast_prices,scale)
% This is my function which plots the forecasted trajectories.

n=size(y,1);
nf=size(fcast_prices,1);
h=size(fcast_prices,2)-1;

x=ones(n,1);
for i=2:n
    x(i)=i;
end

plot(x,y)

hold on;
for i=1:nf
    plot(x(n-h-(nf-i):n-(nf-i)),fcast_prices(i,:), 'b');
end
hold off;

xlabel('TIME (IN TRADING DAYS)'); % <- label on the x axis
ylabel('PRICE OF ALPHABET INC. SHARES'); % <- label on the y axis

axis tight;
axis([min(x) max(x) (1/scale)*min(y) scale*max(y)]); % <- defining range of x and y

end

```

To forecast from a different model (e.g. ARMAX) all that you need to modify is lines 77-84 in the *forecasting.m* script. The graph of the generated forecast trajectories is presented on the next page.

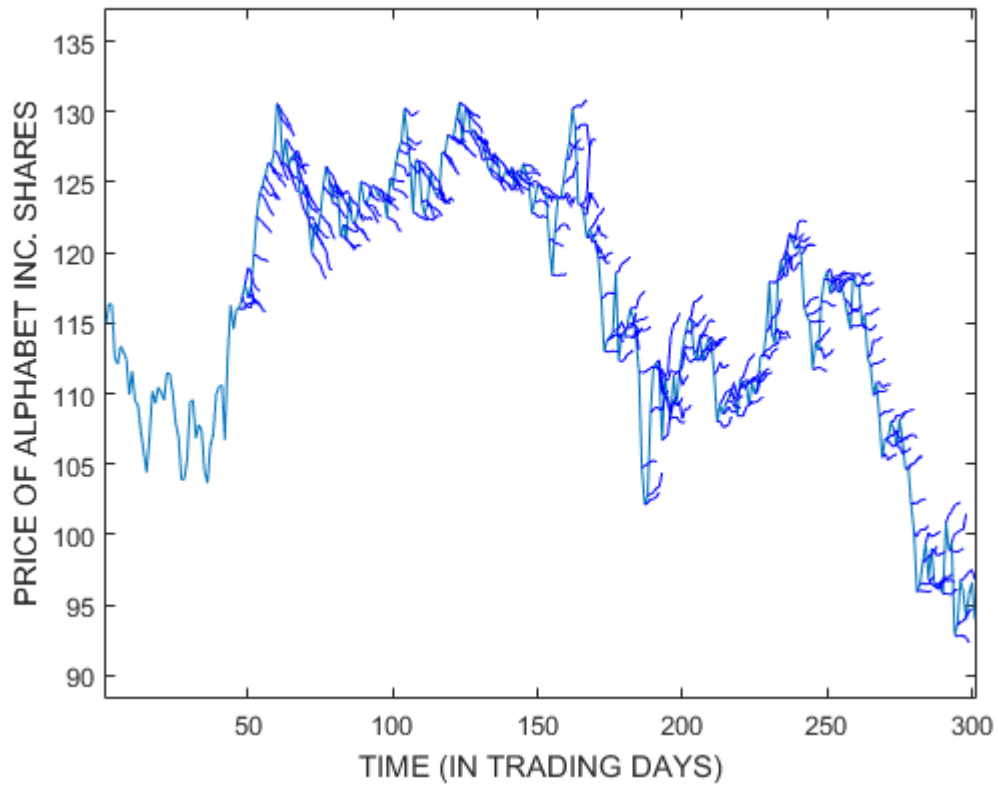


Figure 4: Pine in the wind: ex-post Alphabet Inc. share price forecasts (pretty bad, as you see).

That is all for today. Thank you!