

MATLAB Introduction for Finance and Economics

red.laviste@unibas.ch

University of Basel, Switzerland
Faculty of Business and Economics

September 15, 2019

Content

- 1 Getting Started
- 2 MATLAB Fundamentals
 - Matrix Construction
 - Indexing Matrices
 - Matrix Operations
- 3 Programming Scripts
 - Writing and Running Scripts
 - Flow Control Structures
 - Importing Data
 - Plotting and Visualization
- 4 Functions
 - MATLAB Functions
 - Local and Nested Functions
 - Anonymous functions
 - Optimization in MATLAB

Getting Started

- Short for MATrix LABoratory, developed by 'The Mathworks'
- Matrix-based programming language and software for academia and industry
- Typical uses:
 - Math and computation
 - Data analysis and visualisation
 - Numerical simulations and modelling
 - Algorithm development
- Prepackaged functions and Toolboxes for statistics, econometrics, finance, optimisation,...

MATLAB Layout

- The GUI:
 - Current Folder
 - **Command Window**
 - **Editor** (*click on New Script button first*)
 - **Workspace**
 - File Details or Command History
- Home toolbar
 - New Script, New Function, Import Data, Preferences (Fonts), Set Path, Help
- Editor toolbar
 - New, Open, Save, Insert Section/Comment, Indent
 - **Run**, Run and Advance, Run Section, Advance

Getting Help

- When confronted with an unfamiliar function or command, type `help` or `doc` followed by the function name

```
help clear
```

```
clear Clear variables and functions from memory.  
clear removes all variables from the workspace.
```

```
...
```

```
help clc
```

```
clc Clear command window.  
clc clears the command window and homes the cursor.
```

MATLAB File Formats

- **Scripts** (.m) are for writing your program/script
- **Functions** (.m) can receive input arguments and return output values, unlike Scripts
 - Calls to functions are often made by trailing their name with parentheses between which are input arguments, e.g.
`functionname(x,y)`
- **Data** files (.mat) are variables you can import to, or save from, the Workspace
- The folder in which you save your main script is treated as the root directory

Matrix Types

- MATLAB is a matrix/array-based programming language
- **Elements** stored in an array can be numbers, characters (strings), logical states (true or false)
- A **matrix** is a rectangular array with m rows and n columns
- The square matrix is most common, where $m = n$
- A **row vector** has one row covering n columns
- A **column vector** has one column covering m rows
- A **scalar** is like a 1×1 matrix with only one element

Matrix Examples

$a = 2$ is a scalar

$\mathbf{a} = [1 \ 2 \ 3 \ 4 \ 5]$ is a 1×5 row vector

$\mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ is a 4×1 column vector

$\mathbf{A} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$ is a 3×2 matrix

Matrix Examples written in MATLAB

$$a = 2$$

$$a = 2$$

$$\mathbf{a} = [1 \ 2 \ 3 \ 4 \ 5]$$

$$a = [1 \ 2 \ 3 \ 4 \ 5]$$

$$\mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$b = [1;2;3;4]$$

$$\mathbf{A} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

$$A = [1 \ 4; \ 2 \ 5; \ 3 \ 6]$$

Creating Variables in MATLAB

- Use assignment operator '=' to assign values to a variable
- Naming conventions:
 - First character must be a letter (a-z, A-Z)
 - Any combination of letters, numbers or underscores can follow
 - Case sensitive (x1 is different from X1)
- Apostrophize non-numerical elements in string variables and character arrays

Matrix Construction in MATLAB

- For matrix construction, use square brackets []
- A comma ',' or a space ' ' separates row elements
- A semicolon ';' starts a new row in an array
- All rows should have the same number of elements

Variable Types in MATLAB

Variable type	Example
Scalar	<code>x=5</code>
String	<code>y='oranges'</code>
Row Vector	<code>a=[0, 2, 3, 4]</code>
Column Vector	<code>b=[0; 2; 3; 4]</code>
Matrix	<code>A=[0, 2; 3, 4]</code>
Character array	<code>y=['oranges' 'lemons']</code>
Logical arrays	<code>z=[0 1 1 0 1 0]</code>

Variable Assignment

- When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes/overwrites its contents, allocating more storage if necessary.

Example

```
x = [2; 3];  
y = x;  
x = x*3
```

Sequencing Numbers

- The colon operator ':' is useful for creating vectors quickly.
- Each of the following commands create the same array:

Examples

```
a = [1, 2, 3, 4, 5]
a = 1:5
a = 1:1:5
a = linspace(1, 5, 5)
```

- Sequence $0:2:10$ would run from 0 to 10 in increments of 2
- Reverse sequence with negative increment, $10:-2:0$.
(Default increment is 1.)

Matrix Concatenation

- Connect the following arrays together:

```
a = 1:5;
```

```
b = 6:10;
```

Examples

Horizontal concatenation (side-by-side columns):

```
[a,b] → [ 1  2  3  4  5  6  7  8  9 10 ]
```

Vertical concatenation (stack rows):

```
[a;b] →  $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix}$ 
```

Indexing Elements of a Matrix

- In MATLAB, indexing starts with 1, not 0
- To reference a particular element in a matrix, either:
 - Indicate the element's row and column position: $A(1,2)$
 - Or index of the actual storage sequence: $A(4)$

Examples

$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$	$A(0)$	→	error
	$A(1)$	→	1
	$A(1,2)$	→	4
	$A(4)$	→	4
	$A(\text{end})$	→	6

Indexing Vectors of a Matrix

- The ':' operator can also be used as an index argument
- Useful for extracting rows, columns or subsets of a matrix

Examples

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

$$\begin{array}{ll} A(:) & \rightarrow 6 \times 1 \text{ vector} \\ A(:, 2) & \rightarrow [4; 5; 6] \\ A(1:\text{end}, \text{end}) & \rightarrow [4; 5; 6] \\ A(1:2:\text{end}, 1) & \rightarrow [1; 3] \\ A([1, 3], :) & = \begin{bmatrix} 1 & 4 \\ 3 & 6 \end{bmatrix} \end{array}$$

Deleting Array Elements and Variables

- The command `who` returns a list of all existing variables, while `whos` gives more information about them
- Delete third element in vector `a` with `a(3) = []`
- Delete the first row of an array with `A(1, :) = []`
- To delete a variable, for example array `A`, type `clear A`
- To delete all variables whose names begin with 'A' use `clear A*`

Special Matrices

$$\begin{aligned} \text{ones}(2) &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ \text{zeros}(2) &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \\ \text{eye}(2) &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \text{diag}(x) &= \begin{bmatrix} 1 \\ 4 \end{bmatrix}, \text{ where } x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{aligned}$$

Special Matrices

$$\begin{aligned} \text{rand}(2) &= \begin{bmatrix} 0.6463 & 0.7547 \\ 0.7094 & 0.2760 \end{bmatrix} \\ \text{randn}(2) &= \begin{bmatrix} 0.0774 & -1.1135 \\ -1.2141 & -0.0068 \end{bmatrix} \\ \text{randi}(2) &= \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \\ \text{magic}(2) &= \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \end{aligned}$$

Useful Built-in Functions

- For inspecting array dimensions:
`length()`, `numel()`, `size()`
- For sorting and indexing:
`find()`, `sort()`, `sortrows()`, `flipud()`, `fliplr()`
- For rounding:
`ceil()`, `round()`, `floor()`
- Special math operations:
`sum()`, `cumsum()`, `prod()`, `cumprod()`

Operators

- 1 **Arithmetic Operators** (for numeric computations)
 $+, -, *, /, ^$
 - 2 **Relational Operators** (for quantitative comparisons)
 $>, <, <=, \dots$
 - 3 **Logical Operators** (for logical operations)
 $\&, |, \sim$
- Relational operators compare two arrays element-by-element, returns a same-sized logical array containing 1 (true) or 0 (false).
 - Logical operators and functions also return a logical array but by examining the elements of one or two arrays altogether.

Arithmetic Operators

Symbol	Operation	Example
+	Addition	$2+3$
-	Substraction	$2-3$
*	Multiplication	$2*3$
/	Division	$2/3$
^	Exponentiation	2^3

- The usual algebraic precedence ordering holds for these operators

Matrix and Array Arithmetics

- 2 approaches:
 - 1 Matrix arithmetic - defined by the rules of **linear algebra**
 - 2 Array arithmetic - carried out **element-wise**
- Putting a `'.'` in front of operator instructs MATLAB to use 2nd approach, i.e. `.*`, `./`, `.^`
- To add/subtract two arrays with `+` and `-`, array sizes must match unless one is a scalar because done element-wise for both approaches

Matrix and Array Arithmetics

$$\mathbf{A} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Examples

$A = [1 \ 4; 2 \ 5; 3 \ 6] \rightarrow 3 \times 2$ matrix

$B = [1 \ 2 \ 3; 4 \ 5 \ 6] \rightarrow 2 \times 3$ matrix

$A+B \rightarrow$ Invalid operation $A/B \rightarrow$ Invalid

$A+B' \rightarrow$ Valid operation $A./B' \rightarrow$ Valid

$A.*B \rightarrow$ Invalid $B.^2 \rightarrow$ Invalid

$A*B \rightarrow$ Valid $B.^2 \rightarrow$ Valid

Rules for Matrix Operations

- To multiply matrices with `'*'`, number of columns in A must equal `#` rows in B, unless one is scalar
- To multiply/divide matrices with `'.*'` or `'./'`, both matrices must have same size, unless one is scalar
- Use the transpose operator `'` or the `reshape` and `repmat` functions to make sizes compatible
- Only square matrices and scalars can be raised `'^'` to a power
- Exponent can be another matrix for `'.^'` operation as long as both matrices are the same size, unless one is a scalar

Relational Operators

Operation	MATLAB
equal	==
not equal	~=
greater than	>
less than	<
greater or equal	>=
less or equal	<=

Example 1

```
a = 1;   b = 4;
a < b    → 1 (true)
a == b   → 0 (false)
```

Logical Arrays

Example 2

Given two datasets:

```
x = [2 4 6 8];
```

```
y = [9 7 3 1];
```

Which elements are greater than 5?

```
a = x > 5           → logical([0 0 1 1])
```

```
b = y > 5           → logical([1 1 0 0])
```

Logical arrays can be used to index other arrays, and single out elements that meet our condition:

```
x(a)           x(x > 5)  → [6 8]
```

```
y(b)           y(y > 5)  → [9 7]
```

Logical Operators

Example 2 (cont'd)

Which elements are true in both **a** and **b**?

`a&b` `and(a,b)` \rightarrow `logical([0 0 0 0])`

Which elements are true in either **a** or **b**?

`a|b` `or(a,b)` \rightarrow `logical([1 1 1 1])`

What is the opposite of **b** (\neg **b**)?

`~b` `not(b)` \rightarrow `logical([0 0 1 1])`

Are all/any elements in **a** true?

`all(a)` \rightarrow 0

`any(a)` \rightarrow 1

Built-in Functions and Constants

- Many standard mathematical functions such as `sin`, `cos`, `log`, `sqrt`, `exp` are built-in
- Built-in math constants include: `pi`, `nan`, `inf`, `i`

Example 3

```
x = [0; pi/2; pi; 3*pi/2; 2*pi]
y = sin(x)
```

Statistics Toolbox

- Built-in statistical functions in MATLAB:
mean, median, min, max, mode, std, var, cov,
corrcoef
- The **Statistics Toolbox** is an expansion pack containing a wider range of statistical tools
- Higher moments: skewness, kurtosis
- Gaussian distribution functions:
normrnd, normcdf, norminv, ...
- Hypothesis tests:
jbtest, kstest, ttest, ttest2, ztest, anova1,
anova2, ...

Running Scripts

- A script is a saved MATLAB file (.m) that contains a sequence of commands or code
- Press 'New → Script' in the Home toolbar. This opens the MATLAB Editor
- Pressing **Run** will execute in sequence all lines of code in a script. The script must be saved first

Background for Script Example

- Firms may issue bonds to investors to raise money without giving them ownership. Bond investors are essentially lenders
- During the life of the bond, the investor might collect periodic payments called coupons
- At maturity, the investor receives the face value of the bond plus the final coupon
- What is the present value of a \$100 bond that pays an \$8 coupon annually for 5 years if the interest rate is 7%?

$$PV = \sum_{t=0}^{T-1} \frac{C_t}{(1+r_f)^t} + \frac{FV_T + C_T}{(1+r_f)^T}$$

Script Example

Example: Present Value of a Bond

```
clear; clc; clf;  
  
CF=[0 8 8 8 8 108];  
r=0.07;  
T=5;  
  
DCF=CF./(1+r).^ (0:T)  
PV=sum(CF./(1+r).^ (0:T))
```

- Save the file as 'bondprice.m'
- To run the script, enter 'bondprice' at the command prompt, or press Run

Clean Slate

- After running a script, variables, files and functions are retained in the Workspace and background
- Can either leave these artifacts there, or start from clean slate
- The `clear`, `clc` and `clf` functions automatically cleared the Workspace at the start of the script
- To manually start from a clean slate after each Run:
 - 1 Right-click on Workspace and select **Clear Workspace**
 - 2 Right-click on Command Window and select **Clear Command Window**

Comments

- Add descriptions to the code or instructions on how to use it
- Typing a comment symbol '`%`' at the front of a line of code tells MATLAB to skip that line when running the script
- Double comment symbols '`%%`' specify a new section in the script. Press **Run Section** to execute section by itself
- Comment-out multiple lines of code by selecting them and pressing **Comment** in the Editor toolbar
- To write a multi-line comment, type `%{` for first line then end last line with `%}`

Debugging

- After running a script, any errors are addressed in Command Window with hints to fix them
- Debugging is a methodical process of finding and resolving bugs in a programming script
- Examine the line-by-line execution by setting **Breakpoints** (red ball) next to line numbers. Unpause by pressing 'Continue' or 'Quit Debugging'
- Commenting out lines of code can help narrow in on problem
- Search your error/question online. Best answers usually found on Stackoverflow

Control Structures

- For the implementation of algorithms, a programming language requires control structures for the tasks listed below
- MATLAB offers the following constructs for each task
 - 1 Conditional Execution
 - `if ... end`
 - `if ... else ... end`
 - `if ... elseif ... else ... end`
 - (switch can be used instead of the above)
 - 2 Repetition, looping and iteration
 - `for ... end`
 - `while ... end`
 - `continue`, `break`, `return`
 - 3 Comparison

if ... end

- Syntax:

```
if expr
    cmd
end
```

- If the evaluation of `expr` yields logical 1 (true) or a non-zero result, MATLAB executes one or more commands denoted by `cmd`. Otherwise, MATLAB skips `cmd`.

Example Script

```
x = rand()-0.5;
if x<0
    disp('x is negative')
end
```

if ... else ... end

- Syntax:

```
if expr
    cmd1
else
    cmd2
end
```

Example Script

```
x = rand()-0.5;
if x < 0
    disp('x is negative')
else
    disp('x is positive or zero')
end
```

if ... elseif ... else ... end

- Use this construct if there are over 2 possibilities

Example Script

```
m = 90;  
if m >= 70  
    disp('First Class')  
elseif m >= 60  
    disp('Second Class')  
elseif m >= 50  
    disp('Third Class')  
else  
    disp('Fail')  
end
```

Repetition, Looping and Iteration

- A sequence of calculations is repeated until either
 - All elements in a vector or matrix have been processed
 - The calculations have produced a result that meets a predetermined termination criterion
- Loop constructs in MATLAB:
 - **for ... end**
 - **while ... end**

for loop

- The statements in the **for** loop repeat continuously for a specific number of times
- Syntax:
for index = first:step:last
 cmd
end
- Loop variable
 - defined as a vector
 - is a scalar within the command block
 - does not need to have consecutive values

- 1 Save the following as any name other than 'ou.m':

Example: Ornstein-Uhlenbeck Process

$$dX_t = \kappa(\theta - X_t)dt + \sigma dW_t$$

```
kappa = 0.04; theta = 0; v = 0.2; ou = 0;

for t=2:100
    ou(1,t) = ou(1,t-1) + kappa.*(theta-ou(1,t-1)) ...
        + v.*randn(1,1);
end
plot(ou')
```

- 2 Run the code. Next, put a breakpoint by `end` and iterate by repeatedly pressing **Run** and **Continue** to watch variable `ou` fill

while loop

- The **while** loop executes a statement or a group of statements repeatedly as long as the controlling expression is true
- The number of iterations required for the termination criterion/criteria to be met is not necessarily known in advance
- Beware of infinite loops! Pressing **CTRL+C** stops code execution
- Syntax:

```
while expr  
    cmd  
end
```

Example 1

```
maxIt = 100; sumX = 0; it = 1;
while it <= maxIt
    sumX = sumX + it;
    it = it+1;
end
```

Example 2

```
min_error = 0.1; x_true = 0.5;
it = 1;
while error > min_error
    x_hat = rand();
    error = abs(x_true-x_hat);
end
```

break, continue and return

- The **continue** statement passes control to the next iteration of the loop in which it appears, skipping any remaining statements in the body of the loop.
- The **break** statement terminates the execution of a **for** loop or **while** loop. When a break statement is encountered, execution continues with the next statement outside of the loop.
- The **return** command forces early termination allowing us to exit a function/script prior to the point of normal completion

continue vs. break

- What will be the value of `sum_x`?

Example

```
sum_x = 0;
for k = 1:10
    if sum_x == 5
        continue;
    end
    sum_x = sum_x + x(k);
end
```

- Now try replacing `continue` with `break`

Importing Data into MATLAB

- Commonly used data formats are .csv, .xls, .xlsx, .txt and .mat
- Alternative ways to import data:
 - Use the **Open** button
 - Double-click the data file in **Current Folder** panel
 - Press on **Import Data** button to use Import Wizard
 - Open the file in outside editor like Excel, select and copy data, then create **New Variable** in MATLAB and paste data into its array
 - Use file-opening function like `csvread()` or `load()`

Importing Data with `csvread()`

- Let's try importing stock market data from Yahoo Finance
 - 1 Go to <http://finance.yahoo.com> and search for 'AAPL' in **Quote Lookup** on the Nasdaq
 - 2 Click on **Historical Data** tab in Yahoo Finance
 - 3 For time period, change start date to 1/1/2000 and end date to 6/30/2018 then press **Apply** button
 - 4 Press **Download Data** and save the .csv file to your MATLAB current folder
 - 5 Could now use
`price=csvread('AAPL.csv',nrow,ncol);` . Skip to next step to prepare a .mat file instead

Importing Data with `load()`

- 6 Go to MATLAB and press the **Import Data** button to open the downloaded .csv file
- 7 In Import Wizard, select the closing price column (Close) only and adjust the selection range to exclude non-numerical headers (e.g. E2:E139)
- 8 Change **Output Type** to 'Numeric Matrix'
- 9 Press **Import Selection**
- 10 View the imported variable by double-clicking AAPL variable in the **Workspace**
- 11 Right-click on AAPL variable in the Workspace and save as file AAPL.mat
- 12 In a script you can now use `price=load('AAPL.mat');`

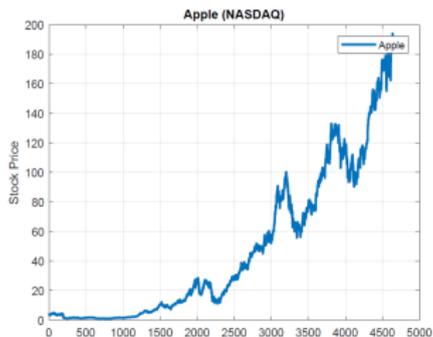
plot Function

- **plot (x)**, where x is some vector of length N , draws the data points contained according to their index $\{(1, x(1)), (2, x(2)), \dots, (N, x(N))\}$
- **plot (x, y)** where x and y are vectors of the same length, draws the points $\{(x(1), y(1)), (x(2), y(2)), \dots, (x(N), y(N))\}$
- Data points are connected with a straight line

Plot Data

Example: Stock Price Chart

```
figure  
plot(AAPL,'linewidth',2);  
grid on  
ylabel('Stock Price')  
title('Apple (NASDAQ)')  
legend('Apple');
```



Customize Plot

- Third argument S in `plot(x, y, S)` is a string that combines line style, color, marker preferences. See all options with `help plot`

k	black	.	dots	-	solid line	-.	dash-dot
g	green	o	circles	:	dotted line	v	triangle (down)
r	red	x	x-mark	--	dashed line	*	star

Examples

```
plot(x, 'k.-') plots a black line marked with dots  
plot(x, y, ' .') plots different colored dots with no line  
plot(x, y, 'c+--') plots a cyan dashed line marked with +'s  
plot(x, y, 'bd') plots blue diamonds with no line
```

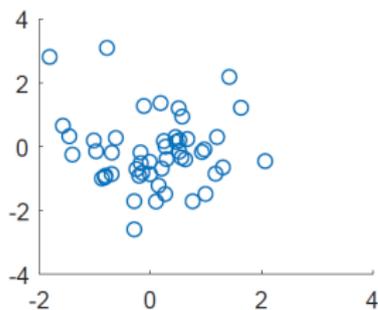
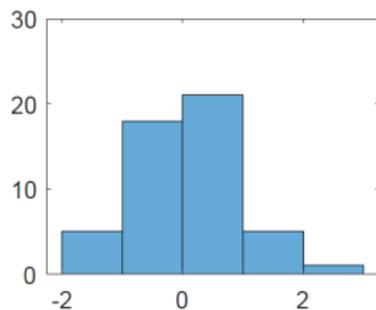
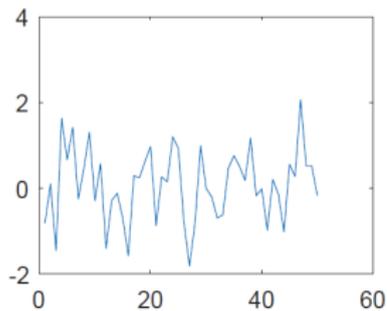
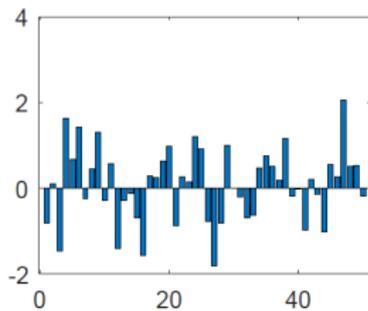
subplot Function

- `subplot(nrows, ncols, thisplot)` is used to make multiple plots in the same figure

Example

```
y = normrnd(0, 1, 50, 1);  
z = trnd(4, 50, 1);  
subplot(2,2,1); bar(y)           % bar plot  
subplot(2,2,2); plot(y)         % line plot  
subplot(2,2,3); histogram(y)   % histogram  
subplot(2,2,4); scatter(y,z)   % scatter plot
```

Subplot Output



MATLAB Tables

- The script below generates a table in the Command Window displaying data from Workspace variables

Example

```
Country = {'Austria'; 'Germany'; 'Liechtenstein';  
'Switzerland'};  
Imports = [14.1; 82.1; 0.47; 16.8];  
Range = [14.5 0.06; 98.4 0.35; 0.9 0.15; 18 0.27];  
inEUR = logical([1;1;0;0]);  
  
T = table(Country, Imports, Range, inEUR)
```

3D Plotting Functions

- `plot3(X, Y, Z)` displays a 3D line plot of a set of data points
- `surf(X, Y, Z)` creates a 3D surface plot colored for height

Examples

```
t = 0:pi/50:10*pi; st = sin(t); ct = cos(t);  
plot3(st, ct, t) % 3D Helix Line Plot
```

```
Z = peaks(30) % 30x30 data matrix  
surf(Z) % Surface Plot of peaks(30)
```

fplot function

- `fplot(fname, lims)` plots the function `fname`, as long as it is univariate
- Default for axis limits is `[-5 5]`, but can be customized with `lims = [xmin xmax]` argument
- `fplot(x, y, lims)` plots coordinates $x(t)$ and $y(t)$ for t between `[xmin xmax]`

Examples

```
fplot(@sin)
fplot(@(x) sin(1./x), [0.01 0.1])
fplot(@(t) cos(3*t), @(t) sin(2*t))
```

Functions

- A MATLAB function (.m) receives **input variables** as parameters for processing, then returns **output variables**
- Functions can be called from Command Window, a script, or other functions
- Offloading code to a function reduces length and clutter in the main script, making debugging easier

Declaring and Calling Functions

- Declare a function in MATLAB with:

```
function [y1, y2, ...] = fname(x1, x2, ...)
```

- This is just the first line. Below it would be the algorithm
 - The reserved word "**function**" must be at the beginning
 - The name of the function should match its file name, excluding .m
- Invoke the function with:

```
[y1, y2, ...] = fname(x1, x2, ...)
```

- The variable names called in as inputs and outputs can be different than how they are named within the function

Background for Function Example

- The **Black-Scholes-Merton model** computes the value of an option based on underlying stock price information

Valuing a Call Option

$$C_0 = S e^{-\delta\tau} \times N(d_1) - X e^{-r_f\tau} \times N(d_2)$$

$$\text{where } d_1 = \frac{\ln(S/X) + (r_f - \delta + \sigma^2/2)\tau}{\sigma\sqrt{\tau}} \quad \text{and} \quad d_2 = d_1 - \sigma\sqrt{\tau}$$

S is the stock price

r_f is the risk-free rate

X is the exercise or strike price

δ is the dividend yield

τ is the time to maturity, $T - t$

v is the stock variance, σ^2

Function Example

- 1 Go to 'New', 'Function' and type the following in Editor:

Example: Call Option Pricing Model

```
function C0 = callBSM(S,X,tau,rf,delta,v)

d1 = (log(S./X) + (rf - delta + v / 2) .* tau) ...
    ./ (sqrt(v * tau));
d2 = d1 - sqrt(v*tau);

C0 = S.*exp(-delta.*tau) .*normcdf(d1,0,1) ...
    - X.*exp(-rf.*tau) .*normcdf(d2,0,1);
```

- 2 Save as callBSM.m and try to run
- 3 Type `C0 = callBSM(75,70,0.75,0.01,0.05,0.35)`

Types of Functions

1 Local functions

- Contained within the main script/same file, but not tabbed
- In a function file, they can appear in any order after the main function in the file
- In a script file, they must be at the end of the file

2 Nested functions

- Contained (tabbed) completely within a parent function
- Can use variables defined in parent function without explicitly passing those variables in as input arguments

3 Anonymous functions

- Called with a defined handle without saving to a function file
- Handy for defining and evaluating a mathematical expression over a range of values

Example: Local Functions

```
u=randn(20,1);  
[avg, med] = newstats(u)
```

```
function[avg, med] = newstats(u) % Parent function  
n = length(u);  
avg = mean(u, n); med = median(u, n);  
end
```

```
function a = mean(v,n) % Local function  
a = sum(v)/n;  
end
```

```
function m = median(v,n) % Local function  
w = sort(v);  
    if rem(n,2) == 1  
        m= w((n+1)/2);  
    else  
        m=(w(n/2)+w(n/2+1))/2;  
    end  
end
```

- In Command Window enter type `taxDemo.m`

Example: Nested Function

```
function y = taxDemo(income)
AdjustedIncome = income - 6000;
y = computeTax;

    function y = computeTax
        y = 0.28 * AdjustedIncome;
    end
end
```

- Now type
`y = taxDemo(90e3) % What is the tax on $90k?`

Anonymous functions

- Handy for defining and evaluating a math formula over a range of values
- Can be constructed at command prompt, in a script, or in a function
- Syntax:

`fhandle = @(arglist) expr`

- `expr`: expression/formula to be evaluated containing variables
- `arglist`: list of input arguments to be passed to the function
- `@` sign: operator that assigns function handle used to invoke the function

Example: Parabola as Anonymous Function

```
a=1; b=0; c=0;
y = @(x) a*x.^2 + b*x + c
fplot(y)
```

Example: Range of Parabolas

```
figure
hold on
b=0; c=0;
for a = 1:5
    fplot(@(x) a*x.^2 + b*x + c, 'b--')
end
hold off
```

Optimization Background

- *What are the optimal inputs for a function that minimize/maximize its outputs?*
- Unconstrained optimization problem:

$$\mathbf{x}^* = \min_{\mathbf{x}} f(\mathbf{x})$$

- $\mathbf{x} = [x_1, x_2, \dots, x_n]$ is decision variable, or input
- $f(\mathbf{x})$ is objective function
- Global optimum \mathbf{x}^* is the stationary point where $f(\mathbf{x}^*) \leq f(\mathbf{x}) \forall \mathbf{x} \in S \subseteq \mathbb{R}^N$ because $f'(\mathbf{x}^*) = 0$
- Local optima only hold in a neighborhood of \mathbf{x}^*
- Unconstrained problem puts no constraints on the range of \mathbf{x}

Optimization Background

- Constrained optimization problem:

$$\mathbf{x}^* = \min_{\mathbf{x}} f(\mathbf{x}) \quad \text{s.t.} \quad \mathbf{x} \geq \mathbf{0}$$

- Objective function is now subject to a non-negativity constraint on \mathbf{x}
- Equality constraints: $g(\mathbf{x}) = \mathbf{0}$
- Inequality constraints: $g(\mathbf{x}) \geq \mathbf{0}$
- Bound constraints: $lb \leq g(\mathbf{x}) \leq ub$

Optimization Background

1 Gradient-based methods

- Iterate through sequence of candidate solutions $\mathbf{x}^{(k)}$ until convergence towards stationary point \mathbf{x}^*
- Approximate $f(\mathbf{x})$ based on its partial derivative(s) $f'(\mathbf{x})$
- Reliable if $f(\mathbf{x})$ is smooth
- Doesn't work if $f(\mathbf{x})$ is non-differentiable

2 Derivative-free methods

- Direct local search evaluates trial solutions in a local area until no improvement can be made
- Use when $f(\mathbf{x})$ is non-smooth or noisy
- Includes Nelder-Mead simplex, genetic algorithms, metaheuristics (simulated annealing)

Optimization Toolbox

- The following functions and routines can be implemented as an M-file, function handle, or inline function
 - Linear programming: `linprog`
 - Quadratic programming: `quadprog`
 - Unconstrained optimization: `fminunc`
 - Constrained optimization: `fmincon`
 - Derivative-free optimization: `fminsearch`
- Search Matlab documentation to see each's objective function
- Matlab optimizers minimize. To maximize, solve $\min [-f(\mathbf{x})]$

- Convex problem ($f''(\mathbf{x}) > 0$) ensures local minimum is a global minimum

Example: Convex Optimization problem

```
tc = @(q) q+q.^2  
  
figure  
fplot(tc)  
  
min = fminunc(tc,0)  
tc(min)
```

- Also try `fminsearch(tc,0)`

- Non-convex problem ($f''(\mathbf{x}) < 0$) might trap optimizer at a local minima

Example: Non-convex Optimization problem

```
tc = @(q) (1/5)*q.^5 + (1/2)*q.^4 - 4*q.^3  
... - q.^2 + 6*q;  
mc = @(q) q.^4 + 2*q.^3 - 12*q.^2 - 2*q + 6;  
  
figure  
fplot(tc)  
figure  
fplot(mc)  
  
min1 = fminunc(mc,0)  
min2 = fminunc(mc,-4)  
tc(min2)
```

Third-party Functions and Packages

- Researchers sometimes share their code on their homepage
 - Gilli, Maringer, Schumann - NMOF
<http://www.nmof.info>
 - Kevin Sheppard (Oxford) - MFE toolbox
http://www.kevinsheppard.com/MFE_MATLAB
 - Jondeau and Rockinger (Lausanne) - Financial applications
<http://www.hec.unil.ch/MatlabCodes>
 - Kendrick, Mercado, Amman (Texas) - Comp. economics
<http://www.laits.utexas.edu/compeco>
- MATLAB's File Exchange website has many user-created packages

References

- P. Brandimarte (2006) *Numerical Methods in Finance and Economics: A Matlab-Based Introduction*
- Duan, Härdle, Gentle *Handbook of Computational Finance*. Chapter 28: MATLAB as a Tool in Computational Finance
- C. Moler (2004) *Numerical Computing with MATLAB*
- K. Nyholm (2008) *Strategic asset allocation in fixed-income markets*. Chapter 2: Essential Elements of MATLAB
- P. Getreuer, *Writing Fast MATLAB Code*
- D.F. Griffithis, *An Introduction to MATLAB*
- Mathworks website, MATLAB Help (printable documentation)