Computational Bootcamp

Basics in Matlab

Kathrin Schlafmann

Copenhagen Business School, DFI, and CEPR

Outline

Outline

Matlab Interface

- Command Window:
  here you can type and execute your commands manually one by one

- Workspace:
  shows all the variables, matrices, functions, etc. that are currently available to work with
  - after constructing an object (using the command window or by executing commands in a script) it shows up here
  - when stopped in debugging mode: all the objects that are currently available for the function to work with (very useful for finding mistakes, see “debugging” later)

- Current Folder:
  - shows all the contents of the current folder, most importantly data-files (ending `.mat`), scripts (ending `.m`)
  - which folder you are working in: set it near the top of Matlab where you can choose your folder

General Functions

- `clear`: clears everything from the workspace
  `clearvars -except` *`varname`* clears everything except the variable *varname*
- `clc`: clears the command window
- `save`: save current workspace (or a subset) to a `.mat`-file
- `load`: load a `.mat`-file into the workspace
- `close all`: close all open graphs
- `help` *`functionname`* displays explanatory text for function *functionname*

  $\rightarrow$ however: often easier to directly open extensive help functionality and search for function there (easier to read)

Scalars, Vectors, and Matrices

- constructing a scalar:

```
a = 2
b = 3*a
```

Do you want to suppress the output? → end the line with a semicolon: `c = a*b;`

- constructing a vector:
  - row vector:
    - `d = [1 2 5 4]`
    - `d = [1, 2, 5, 4]`
  - column vector:
    - `e = [1; 2; 5; 4]`

- constructing a matrix

```
f = [1, 2, 5, 4; 5, 6, 2, 7]
f = [1, 2, 5, 4; ...
     5, 6, 2, 7]
```

→ triple dots `...` tell Matlab to ignore the rest of the line and treat the next line as continuation of the current one (very useful for making your scripts readable!)

Scalars, Vectors, and Matrices

- automatic initialization:
  - `f = linspace(1,10,19)`
    → creates equally spaced row vector with 19 elements, starting from 1 to 10
  - `f = (1:0.5:10)`
    → creates row vector with elements starting from 1, increments of 0.5, up to 10 (or less)

    note: increments can be negative!
  - `f = logspace(0,2,20)`
    → creates logarithmically spaced row vector with 20 elements, starting from 10^0 to 10^2

Scalars, Vectors, and Matrices

Special Matrices

- `zeros(m,n)`: constructs a matrix of zeros of size m-by-n
- `ones(m,n)`: constructs a matrix of ones of size m-by-n
- `eye(m)`: constructs a identity matrix of size m-by-m
- `NaN(m,n)`: constructs a matrix of `NaN` of size m-by-n
  - `NaN` = "not a number"
  - if you get `NaN` as result: typically a sign that something is wrong
  - useful to initialize matrices with (to ensure that you replace all cells and don't forget any)
- `magic(n)`: constructs a square matrix of n-by-n constructed from the integers 1 through n^2 with equal row, column, and diagonal sums
- `rand(m,n)` / `randn(m,n)`: constructs a matrix of size m-by-n filled with random draws from a uniform / a standard normal distribution

note: all of these commands can also be called with only one argument (e.g. `zeros(m)`)
→ constructs square matrices of size m-by-m

Scalars, Vectors, and Matrices

Indexing

referring to submatrices in `g = magic(5)`:

- `g(2,3)` refers to the element in row 2, column 3
- `g(:,3)` refers to all elements in column 3 $\rightarrow$ column vector
- `g(2,:)` refers to all elements in row 2 $\rightarrow$ row vector
- `g(1:3,3)` refers to the elements in row 1-3 in column 3 $\rightarrow$ column vector
- `g(2:end,:)` refers to all elements in row 2 to the last row, all columns $\rightarrow$ matrix that excludes the first row
- `g(2:end-1,:)` $\rightarrow$ matrix that excludes the first and the last row

Other objects

There are more types of objects in Matlab apart from scalars, vectors and matrices, in particular:

Structures:

- they are like categories of things, which you can use to organize your variables
- construction example: `parameters.beta = 0.9;`
  $\rightarrow$ generates a structure with the name "parameters" (if it doesn't exist yet) which has a field that contains the variable "beta"
- very useful in function calls:
  - without structures: you will have to supply all your parameters, grids, and other variables separately to your function
    $\rightarrow$ with a lot of objects this can get very tedious and prone to cause mistakes!
  - with structures: organize your objects into a few structures, then you can simply supply the structures to your functions

Other objects

Cell Arrays:

- unlike matrices cell arrays can hold a variety of objects as elements, not just numbers
- useful for example to hold lists of strings
- constructed by using curly brackets:
  ```
  Acell = {5,'some string'};
  ```
- objects can then also be accessed by using curly brackets:
  ```
  a = Acell{1,1};
  ```

Operators

- standard (matrix) operators:
  - $+$ plus
  - $-$ minus
  - $*$ (matrix) multiplication
  - $/$ division
  - $\wedge$ (matrix) exponential
- element-by-element operators:
  - $.*$ multiplication
  - $./$ division
  - $.\wedge$ exponential
- comparison operators:
  - $==$ equal
  - $\sim=$ not equal
  - $>$ larger than
  - $<$ smaller than
  - $>=$ larger or equal
  - $<=$ smaller or equal

Outline

Loops

- loops repeat a particular set of commands until a criterion is met
- types:
  - for-loops:
    - in each iteration: the loop control variable takes on another pre-specified value
    - runs the interior code a pre-specified number of times (as often as values given for the loop control variable)
  - while-loops:
    - keeps repeating the interior code until a stopping criterion is met
    - ex ante you do not know how often the loop will repeat!

For-Loop

- syntax example:

```
for i = 1:10
    interior command block
end
```

  → repeats the loop 10 times, where `i` takes on values 1,2,...,10 successively

- values that `i` will take can be stated very flexibly:
  - `i = 10:-1:1`
  - `i = linspace(1,10,10)`
  - `i = 1:length(x)`
  - `i = 1:size(X,n)`
  - any other way you can define a vector (including `i = x` where `x` is a vector)

For-Loop

note: using several for-loops inside each other...

- makes codes easy to understand (might be a good way to start thinking about how you want to compute things)
- makes codes very slow

$\rightarrow$ we will use both loops and vectorization

$\rightarrow$ most often it will be faster to vectorize

While-Loop

- syntax example:

```
convCrit = 10
while convCrit > 1e-3
     interior command block
     convCrit = ...
end
```

  $\rightarrow$ repeats the loop until the scalar `convCrit` is smaller or equal to 0.001

- note that `convCrit` needs to be updated in each iteration, otherwise the code will never stop!

While-Loop

- sometimes useful to add a statement which ensures a maximum number of iterations:

```
convCrit = 10
count = 1
while (convCrit > 1e-3) && (count <= 1000)
    interior code
    convCrit = ...
    count = count + 1;
end
```

If-Statements

- syntax example:

```
if  x > 10
    interior command block 1

elseif  x > 5
    interior command block 2

else
    interior command block 3

end
```

- code checks: is x larger than 10?
  - If yes: execute *interior command block 1*
  - If no: code checks: is x is larger than 5?
    - If yes: execute *interior command block 2*
    - If no: execute *interior command block 3*

If-Statements

Note:

- Matlab will never check the **`elseif`**-condition if the **`if`**-condition was already true!
- can have many **elseif**-blocks but only one **if**-block and only one **else**-block
- does not have to contain an **elseif**- or **else**-block

Outline

Scripts

- collection of commands
- saved in a `.m`-file
- extremely useful: apart from small tests or during debugging, you won't use the command line directly
- can be executed
  - all at once (using F5 key)
  - in part: highlight part of code and execute only that part (using F9 key)
- should contain thorough commenting!!!
  - comments start with `%`
  - can be whole line or at the end of a line
- can contain cells of code:
  - cells are constructed by adding a line which starts with `%%`
  - nice way to structure your code for
    - readability
    - execution of one cell after another (execute individual cell with ctrl+enter)

Functions

- two types of functions: build-in and user-written
  - build-in: we have already seen quite a few of those
  - user-written:
    - another way of structuring your code
    - particularly useful if you are going to do the same thing several times (at different points in your code)
- saved in a `.m`-file just like scripts

Functions

- specific syntax of `.m`-file of a function:

```
function [y1,y2,...,yN] = myfun(x1,x2,...xM)
     interior command block
end
```

  - `myfun` is the function name (`.m`-file should have the same name!)
  - `x1,x2,...,xM` are inputs to the function (scalars, vectors, matrices, etc.)
  - `y1,y2,...,yN` are outputs of the function (need to be defined within the function)

Functions vs Scripts

What is the difference between scripts and functions?

- what they use as inputs:
  - function: when called only receives input parameters, has nothing else to work with
  - script: when called or executed has whole workspace available
- what they have as output:
  - function: only returns the objects defined in the **`function`**-statement, all other objects in the function file are temporary (local variables)
  - script: returns *all* variables that are used in the script

Functions vs Scripts

- how they interact with the workspace:
  - function: does not interact at all (does not have access to it)
    $\Rightarrow$ cannot alter it apart from returning the output variables
  - script: works directly in the workspace
    $\Rightarrow$ any change you make in the script will permanently alter the variables in the workspace!
    $\Rightarrow$ dangerous, since you can easily overwrite variables if you are not careful!

Useful Build-in Functions

Matlab has a vast number of build-in functions. Here are a few useful ones you might use a lot:

- `sum(X)`: computes the sum of each column of matrix `X`
- `min(X)`: computes the minimum of each column of matrix `X`
- `max(X)`: computes the maximum of each column of matrix `X`
- `mean(X)`: computes the mean of each column of matrix `X`

Note:
by adding a second argument to the function call (e.g. `sum(X,n)`) you can tell matlab which dimension it should act on (very useful!); works on `sum(), min(), max()` and many more. If in doubt look at the help files!

Useful Build-in Functions

More useful functions for manipulating matrices

- `repmat()`: generates a matrix by repeating an existing matrix, can be very flexible with dimensions
- `reshape(X)`: changes the dimensions of a matrix, but keeps all elements (and doesn't add new ones)
  $\rightarrow$ useful shortcut to stack all columns of a matrix on top of each other to form a column vector: `Xvec = Xmat(:);`
- `permute()`: swaps the order of dimensions

Note:
For the exact syntax of these functions see the help files!

Useful Build-in Functions

Setting a seed for the random number generator:

- When you simulate your models, it is important that you can reproduce your results exactly
  - but if your simulations involve random draws from distributions, then Matlab will draw different numbers each time you run your code

    $\Rightarrow$ your results won't be identical! (Your main findings should of course not depend on the random draws, but the finer details might vary a bit with the random draws)

  - but at least at two stages of your research it is important that you can reproduce your results exactly:
    1. debugging your code: if you run into an error, you want to be able to reproduce the error exactly so that you can find out what is wrong
    2. publication: you want to make sure that you can reproduce your results if you want to publish them
- set a seed in Matlab to always get the same "random" numbers when you run your code: `rng(`*`integer`*`)`

Outline

Debugging

Within a Script

Easiest case: you are working in a single Matlab script

- execute code line by line
- each time: check that the objects you create are what you want to create
  - open objects in Variables-window: do they look like you expect them to?
  - are the dimensions of the the newly created objects what you expected?
- if Matlab runs into an error, it displays the error message directly (and you know in which line the error occurred)
  - error messages are typically quite informative!
  - note: just because an error occurs in a particular line does *not* mean that your bug is in that line (could have occurred (much) earlier in previous lines!)

  $\Rightarrow$ might need to go back to previous lines and check all their output

Debugging

When using Functions

More advanced (but more common and more useful): you are working with functions

- if you execute your code and the error occurs in a function:
  - Matlab tells you exactly in which line in the function (very useful!)
  - problem: after the error Matlab leaves the function, so you don't know what the (local) variables look like that the function is working with at this point
- solution: use debugging functionality in Matlab:
  - set a breakpoint in the line of the function where the error occurred
  - when you execute your code, Matlab will stop right before executing that line
  - you will now see all variables in your workspace that the function can use at this point (so you can now investigate what is wrong)

Debugging

(Most) Useful Function: `size()`

- most common error messages involve a statement like “sizes don’t match”, “dimensions not consistent”, etc.
- most useful Matlab function (in my experience):
  - `[d1,d2,d3,...] = size(X)`
    - `size(X)` returns the dimensions of the object `X`
    - `size(X,n)` returns the length of the `n`th dimension of the object `X`
  - `length(x)`
    $\rightarrow$ returns the length of the vector `x`
- apply `size`-function separately to different terms in your (potentially complicated) expressions
  - if you run into the same error on a subset of your expression you can narrow down which part is causing problems
  - once narrowed down: compare dimensions that these terms have with what you expect them to have!!!

Further Resources

- MIT OpenCourseWare:
  - "Introduction to Matlab": `https://ocw.mit.edu/courses/6-057-introduction-to-matlab-january-iap-2019/`
  - "Introduction To MATLAB Programming": `https://ocw.mit.edu/courses/mathematics/18-s997-introduction-to-matlab-programming-fall-2011/index.htm`
- books:
  - "Getting Started with MATLAB 7: A Quick Introduction for Scientists and Engineers"; Rudra Pratap; Oxford University Press
  - "A Guide to Matlab for Beginners and Experienced Users"; Brian Hunt, Ronald Lipsman, Jonathan Rosenberg; Cambridge University Press