Class 1: Introduction to the course and MATLAB

Contents

- Big Picture of the Course
- Why do we need computational methods?
- MATLAB
- Assignment
- Basic Operations
- Built in functions
- Indexing
- Conditional statements
- Scripts and Functions
- Loops
- No Types
- A Note on Efficiency
- Solving Linear Equations

Big Picture of the Course

- Learn and develop computational skills that will help you with your research.
- Knowlwdge is applicable across fields.
- Traditional structural fields like IO, Trade, and Macro.
- Theory -- simulate calibrated models and graph results.
- Metrics -- Monte Carlo simulations. Code own estimator.
- Even you you don't use computational methods too much, you will have to referee papers that do.

Why do we need computational methods?

An Example: Consider a demand fucntion `q=p^{-0.2}`. We could easily compute the quantity for any given price with a handheld calculator. We could also solve for price given a quantuty.

Now consider a slightly different demand: `q = 0.5p^{-0.2} + 0.5p^{-0.5}`. We could easily compute demand for a given price, but if you wanted to know what price cleared the market for `q=2`, no closed form inverse demand function exists. We need computational methods to find the price that clears the market for a given quantity.

MATLAB

MATLAB is a high-level language for numerical analysis. Code can be written concisely, and basic code can be easily readable by anyone who knows basic computer programming and linear algebra. We will go over some basic uses of MATLAB, but this is not a comprehensive overview. See the appendix of MF for a better overview, but the best way to learn is to do things yourself on the homework.

MATLAB is one of many languages used to do numerical and statistical work in economics. Other popular choices are R, python (with numpy), and sometimes julia or C or even (shudder) FORTRAN. These all have pros and cons. We're using MATLAB because it is user friendly and something of an industry standard. It also is very well documented relative to open source options. Most importantly, all our sample code is written in MATLAB. You are free to program in whatever you want, just don't ask me to debug it.

Assignment

```
a = 3
```

```
a =

     3
```

Terminate lines with semicolon to suppress output

```
b = 2;
```

Matrix assignment

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

```
A =

     1     2     3
     4     5     6
     7     8     9
```

Or

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

```
B =

     1     2     3
     4     5     6
     7     8     9
```

Quick Vector of an index

```
c = 1:10
```

```
c =

     1     2     3     4     5     6     7     8     9    10
```

is the same as

```
d = 1:1:10
```

```
d =

     1     2     3     4     5     6     7     8     9    10
```

is the same as

```
e = linspace(1,10,10)
```

```
e =

     1     2     3     4     5     6     7     8     9    10
```

Assign vectors to matrices

```
C = [c;d;e]
```

```
C =

     1     2     3     4     5     6     7     8     9    10
     1     2     3     4     5     6     7     8     9    10
     1     2     3     4     5     6     7     8     9    10
```

Basic Operations

```
b = [1; 2; 3]

a+b

z = a+b

A = magic(3)

D = A+B
```

```
b =

     1
     2
     3


ans =
```

```
     4
     5
     6
```

```
z =

     4
     5
     6
```

```
A =

     8     1     6
     3     5     7
     4     9     2
```

```
D =

     9     3     9
     7    10    13
    11    17    11
```

Matrix Multiplication

```
A*B
```

```
ans =

    54    69    84
    72    87   102
    54    69    84
```

Array multiplication (element-wise)

```
A.*B
```

```
ans =

     8     2    18
    12    25    42
    28    72    18
```

Transpose

```
A'.*B
```

```
ans =

     8     6    12
     4    25    54
    42    56    18
```

Inverse

```
inv(A)
```

```
ans =

    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028
```

Built in functions

There are many built in functions. No need to add libraries or packages. Just google what you need and there is likely a package already in MATLAB. Some built in functions work well for only specific applications, and sometimes there are much better thrid-party packages available.

We will make use of an added library called COMPECON Toolbox, http://www4.ncsu.edu/~pfackler/compecon/toolbox.html. This should already be loaded on the lab computers in the building. It is open source, so download for your own personal computer. Put the folder in somewhere and tell MATLAB to look for the packages by adding `addpath('path of folder')` to the top of your script or function

Useful buil-in functions: `exp`, `inv`, `diag`, `ln`, `abs`, `size`, `rand`, `zeros` etc.

How to find a function? Use the documentation. Always available on the menu bar or you can just google what you want with `MATLAB' in the search string.

Indexing

```
A
```

```
A =

     8     1     6
     3     5     7
     4     9     2
```

Refer to the first element of a matrix:

```
A(1)
```

```
ans =

     8
```

Refer to a specific element (row X column):

```
A(2,3)
```

```
ans =

     7
```

Refer to a vector in a matrix:

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

```
ans =

     8
     3
     4
```

Refer to a matrix in a matrix:

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

```
ans =

     8     1     6
     3     5     7
     4     9     2
```

Conditional statements

You can execute blocks of codes conditionally

```
a=0;
if a<3
    disp('a is less than 3')
else
    disp('a is not less than 3')
end
```

```
a is less than 3
```

Scripts and Functions

A script is a file with a set of commands. If you execute the script it will execute those commands in order. There is no input, and the function does not return any output, although it can create output by printing things to screen or saving files.

A function is a file that is defined to take input(s) and give output(s). We will often define user functions.

For example, let's say I was working on a problem that involved a demand function $q = e^{-p}$. I can write a function in a separate file called `demand.m` for this:

```
function q = demand(p)

q = exp(-p);

end
```

Then we can call `demand.m` to evaluate any given `p`.

```
demand(3)
```

```
ans =

    0.0498
```

We can assign the output of a function to a variable in the local space (for example in a script)

```
q1 = demand(3)

p = 1:1:10;
Q = demand(p)
```

```
q1 =

    0.0498


Q =

  Columns 1 through 7

    0.3679    0.1353    0.0498    0.0183    0.0067    0.0025    0.0009

  Columns 8 through 10

    0.0003    0.0001    0.0000
```

Loops

```
Qloop = zeros(1,10);
Qlen = length(Qloop);

for ix = 1:Qlen
        Qloop(ix)=demand(ix);
end

disp('This is the vector we just filled')
disp(Qloop)
```

```
This is the vector we just filled
  Columns 1 through 7

    0.3679    0.1353    0.0498    0.0183    0.0067    0.0025    0.0009

  Columns 8 through 10

    0.0003    0.0001    0.0000
```

No Types

Unlike lower level languages, no need to preallocate types. Also, you can write over variables, even with different types. Before we defined a variable `c`. We can just write over it (this is not neccesarily a good thing!).

To see this, look how an innocent matrix multiplication becomes an error. Also, notice how MATLAB lets me gracefully handle the error using try-catch.

```
b = [1 2 3];

out = b*C

b = 'happy'

try
   out = b*C
catch e
    fprintf('MATLAB is not happy: %s\n', e.message);
end
```

```
out =

     6    12    18    24    30    36    42    48    54    60


b =

happy

MATLAB is not happy: Inner matrix dimensions must agree.
```

A Note on Efficiency

MATLAB is built differently than other languages in that it is written with linear algebra in mind. In many other languages you might use loops to construct matrices or do algebra, but in MATLAB it is best to use the matrix and array operators. Also, MATLAB accesses memory differently than lower level langauages like C/Fortran. Generally, you want to avoid loops. Note: There is a recently developed language called **Julia** that has the user-friendly features and syntax of MATLAB, but performs similarly to C/Fortran.

Solving Linear Equations

There are many methods for solving a system of linear equations, $Ax=b$. In general, MATLAB understands the properties of your system of equations and uses a method that is most efficient, yet still accurate. In general, MATLAB will use an L-U factorizaiton algorithm, but will also adjust the matrix if it is ill-conditioned.

The most efficient way to solve a system of linear equations:

Solve system of linear equations

```
b = [1; 2; 3];
x = A\b
```

```
x =

    0.0500
    0.3000
    0.0500
```

Above, MATLAB knows what is going on with the equation, and will perform LU factorization (or another method if applicable) to solve for x. This is not the same as

```
y = inv(A)*b;
```

although they will likely result in the same answer for a well conditioned A matrix.

Always beware when inverting matrices. Large matricies with a wide range in scaling, or with columns that are nearly correlated, can cause huge computational issues, and the root cause can remain hidden because MATLAB will technically be able to take the inverse.

Published with MATLAB® R2016a