Computational Methods Lecture 2:

Getting Started With MATLAB

February 4, 2026

What Is MATLAB?

MATLAB is a programming language for numerical work

- We use it to implement economic models
- We use it to solve equations and optimizations
- We use it to make figures and tables

Quick qualification

- I'll cover only a small subset of things that are useful in MATLAB
- My intention for now is just to get you started
- Check MathWorks or ask ChatGPT
- Coding is not about memorizing libraries of functions
- It's about knowing where to find stuff and being able to use it

The MATLAB Screen

- **Command Window** runs one line at a time
- **Editor** runs a saved script of many lines
- **Workspace** shows current variables
- **Current Folder** shows files on disk

Scripts

A script is a file of MATLAB commands

- Save scripts as `something.m`
- Run scripts from the Editor
- Scripts create variables in the Workspace

Numbers and Variables

Let's do some simple operations/assignments in MATLAB's command window

```
2 + 4       % You can use MATLAB as a calculator. This line yields ans = 6
x = 7       % From now on, whenever you type x that is the same as typing 7
x * 2       % This line returns ans = 14
x = 8       % This line overwrites x. Now typing x is typing 8
y = 5 * 3   % Variables can be the result of operations
z = 2 * x   % These operations can involve previosuly assigned variables
x = 2 * x   % Variables can be overwritten self-referentially
x = x + 1;  % A semicolon surpresses printing. The workspace is updated, though
```

- A variable is a named value and stored in your workspace
- In a given line, MATLAB doesn't run anything following a % (commenting)
- `ans` stores the most recent output you generated and is overwritten by the next
- Note that in MATLAB `x = 2 * x` is not an equation that implies $x = 0$
- Instead, for any given `x`, the variable `x` is overwritten with its doubled value

Common Mathematical Functions

MATLAB has a large library of built-in math functions used constantly in modeling

```
x = 2;

exp(x)          % e^x  returns ans = 7.389...
log(x)          % Natural log returns and = 0.693...

sin(x)          % Sine of x
cos(x)          % Cosine

sqrt(x)         % Square root
abs(-3)         % Absolute value = 3

y = [1, 4, 9];
sqrt(y)         % Functions apply elementwise to vectors/matrices
```

- These functions automatically act elementwise on arrays
- `log` always means natural log; for base changes use `log(x)/log(b)`

Starting a Script

Most scripts start by resetting the session

```
clear all
close all
clc

x = 7
x = x^2
```

- Save as `sample_script.m`
- Run via editor or by typing `sample_script` in the command window

What housekeeping does

- `clear all` removes old variables
- `close all` closes old figures
- `clc` clears the command window text

Vectors and Matrices

Vectors are lists and matrices are tables

```
vc = [10; 20; 30; 40]  % Separating numbers by ; generates a column vector
vr = [10, 20, 30, 40]  % Separating numbers by , generates a row vector
v  = vr'               % ' transposes a vector such that v = vc
A  = [1, 2; 3, 4]      % This stores a matrix with rows [1, 2] and [3, 4]
```

Indexing selects entries

```
v(1)            % This selects the first entry from v so that ans = 10
v(2:3)          % This selects the second and third entry from v so that ans = [20; 30]
v(end)          % This selects the last entry from v so that ans = 40
v(end - 1)      % This selects the second-to-last entry from v so that ans = 30

A(1,2)          % This selects the first row, second column entry of A so that ans = 2
A(:,1)          % This selects the entire first column of A so that ans = [1; 3]
A(2,:)          % This selects the entire second row of A so that ans = [3; 4]
b = A(2,2)      % This assigns the (2,2) entry of A to b so that b = 4
```

Creating Vectors

Vectors are everywhere in numerical work, so MATLAB has shortcuts

```
v1 = [1; 2; 3; 4];        % Column vector typed entry-by-entry
v2 = [1, 2, 3, 4];        % Row vector typed entry-by-entry

v3 = 1:4;                 % Row vector [1, 2, 3, 4]
v4 = 0:0.5:2;             % Start:step:end [0, 0.5, 1.0, 1.5, 2.0]

v5 = linspace(0,1,5);     % 5 evenly spaced points between 0 and 1
```

- `a:b:c` gives a row vector starting at `a`, ending near `c`, with step `b`
- `linspace(a,b,n)` gives exactly `n` points from `a` to `b`

Practical rule

- Use `a:b:c` for grids with a natural step size.
- Use `linspace` when you care about the number of points, not the step.

Basic Vector Operations

Vector arithmetic looks like the math you know

```
x = [1; 2; 3];
y = [4; 5; 6];

x + y          % Vector addition: [5; 7; 9]
x - y          % Vector subtraction: [-3; -3; -3]
3 * x          % Scalar multiplication: [3; 6; 9]
```

Useful summaries:

```
sum(x)         % 1 + 2 + 3  = 6
mean(x)        % (1 + 2 + 3)/3  = 2
max(x)         % Maximum entry: 3
min(x)         % Minimum entry: 1
```

Consistency check

- Vector addition and subtraction require same length. If a simple operation fails, check `size(x)`.

Vector Norms and Dot Products

The dot product and norm are fundamental

```
x = [1; 2; 3];
y = [4; 5; 6];

dot_xy  = x' * y          % Dot product = 1*4 + 2*5 + 3*6 = 32
dot_xy  = dot(x, y);      % Built-in dot product (same result)

nx = norm(x)              % Euclidean norm: sqrt(1^2 + 2^2 + 3^2)
ny = norm(y)              % Same for y
```

- `x' * y` is the standard matrix formula for the dot product
- `norm(x)` gives the Euclidean length of `x`

Sanity check

- Dot products are scalars. If you get a vector or matrix, you did something else.

Building and Inspecting Matrices

Matrices collect vectors into tables.

```
A = [1, 2; 3, 4];          % 2-by-2 matrix
B = [5, 6; 7, 8];          % Another 2-by-2 matrix

size(A)                    % Returns [2 2]

I = eye(3);                % 3-by-3 identity matrix
Z = zeros(2,3);            % 2-by-3 matrix of zeros
O = ones(2,3);             % 2-by-3 matrix of ones
```

- `size` is the first tool when debugging matrix code
- `eye`, `zeros`, and `ones` are used constantly for initialization

Quick habit

- Before any complicated operation, check dimensions with `size`.
- When you see a cryptic dimension error, print `size` for each object involved.

Matrix Algebra

Usual rules of matrix algebra apply.

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

A + B          % Entrywise addition (same size required)
2 * A          % Scalar times matrix

C = A * B      % Matrix product: (2-by-2) * (2-by-2) -> (2-by-2)
D = B * A      % Different product (matrix multiplication is not commutative)
```

Dimension rule:

```
A        % m-by-n
B        % n-by-k

A * B    % Defined, result is m-by-k
B * A    % Only defined if k = m
```

More Matrix Operations

Useful built-in operations on matrices

```
A = [1, 2, 3; 4, 5, 6];

sum(A)          % Column sums   [5, 7, 9]
sum(A,2)        % Row sums      [6; 15]

mean(A)         % Column means
mean(A, 2)      % Row means
A'              % Transpose

d  = diag(A);   % Take main diagonal as a column vector
D  = diag(d);   % Put d on the diagonal of a square matrix
```

- `sum` and `mean` default to working down columns.
- `diag` switches between a vector of diagonal entries and a diagonal matrix.

Elementwise vs Matrix Operations

Dots mean element-by-element operations

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

x.^2            % Squares each entry: [1; 4; 9]
x .* x          % Same as x.^2

x * x'          % (3-by-1)*(1-by-3) = 3-by-3 matrix
x' * x          % (1-by-3)*(3-by-1) = 1-by-1 scalar (dot product)
```

- A dot before an operation gives you the element-wise version of this operation
- Use elementwise operations when you want the same formula applied entry-by-entry
- Use matrix operations when you want linear algebra
- Remark: In linear algebra element-wise operations are Hadamard products $\circ$

Logical Expressions

Conditions in MATLAB are numeric: 1 (true) or 0 (false)

```
x = 3;
y = 5;

x > 2                % 1 (true)
x < 2                % 0 (false)
x == 3               % 1 (true)
x ~= y               % 1 (true)    ~= means not equal

(x > 1) && (y < 10)  % Logical and, true if both sides are true
(x < 1) || (y < 10)  % Logical or, true if at least one side is true
```

Vector conditions

- For vectors, `x > 0` returns a vector of 0/1 flags
- To test if *all* entries are positive use `all(x > 0)`
- To test if *any* entry is positive use `any(x > 0)`

If Statements

An `if` block runs code only when a condition is true

```matlab
x = 3;

if x > 2
    y = 10;
else
    y = -10;
end % returns y = 10
```

Common mistakes

- Use == for equality, not =
- Always close `if` blocks with `end` or MATLAB will complain
- Compare floating-point numbers with tolerances, not exact equality, in serious numerical work

For Loops

A `for` loop repeats code a fixed number of times

```
s = 0;
for k = 1:5
    s = s + k;
end % returns s = 15
```

Looping over indices of a vector

```
x = [10; 20; 30; 40];
n = length(x);
for i = 1:n
    y(i) = 2 * x(i);
end
```

Performance tip

- In many cases, vectorization (avoiding loops) is faster and clearer: here `y = 2 * x;` is all you need.

While Loops

A `while` loop repeats until a condition fails

```
x = 1;
while x < 100
    x = 2 * x;
end % returns first value >= 100
```

Using a tolerance (common in numerical algorithms)

```
x    = 1;
diff = 1;
tol  = 1e-6;
while diff > tol
    x_new = 0.5 * (x + 2/x);    % Example update
    diff  = abs(x_new - x);
    x     = x_new;
end
```

Why Loops Matter

Models are full of repeated tasks:

- Simulating a time series
- Solving a fixed point by iteration
- Computing moments across many households
- Stepping through time in dynamic programming

Loop vs vectorization

- Use loops when the current step depends on the previous step (e.g., simulations).
- Use vectorized operations when all elements can be updated independently.

Why Write Functions?

A function packages a task you will reuse:

- Cleaner scripts
- Fewer copy-paste mistakes
- Easier debugging
- Natural place to test components in isolation

Economic workflow

- Put model-specific code (parameters, calibration choices) in scripts.
- Put generic computations (utility, production, transitions) in functions.

Function Handles

A function handle stores a formula in a variable.

```
f = @(x) x.^2;           % Anonymous function: f(x) = x^2
f(2)                     % Returns scalar ans = 4
f([-2, -1, 0, 1, 2])     % Returns vector ans = [4, 1, 0, 1, 4]
```

With parameters:

```
alpha = 0.3;             % Capital elasticity of output
f = @(k) k.^alpha;       % DRS production function
f(4)                     % Returns scalar ans = 1.5157
f([1, 2, 3, 4])          % Returns vector ans = [1 1.2311 1.3903 1.5157]
```

When are handles useful?

- Passing functions to solvers like `fzero`, `fminsearch`, etc. (We'll see this later)
- Quickly trying different formulas without creating new files.

Function Files

A function file is a separate `.m` file.

- The first line starts with `function`
- The file name must match the function name
- Inputs and outputs are local to the function
- Functions do not see your workspace unless you pass variables in

Think about this as

- A script is a story of what you are doing
- A function is a well-defined operation you can reuse anywhere

Example Function File

Save this as `square.m`

```
function y = square(x)
    y = x.^2;
end
```

Then call it from a script or the Command Window

```
square(3)              % Returns scalar ans = 9
square([-2, 0, 2])    % Returns vector ans = [4, 0 , 4]
```

Functions With Multiple Outputs

Functions can return several objects at once

```
function [u, mu] = utility(c)
% CRRA utility with CRRA = 2 and marginal utility

    gamma = 2;                                  % Assigns CRRA = 2
    u     = (c.^(1-gamma) - 1) ./ (1-gamma);    % Computes utility
    mu    = c.^(-gamma);                        % Computes marginal utility

end
```

```
[u, mu]   = utility(3);   % Capture both outputs
u         = utility(3);   % Only the first output
```

Rule

- Use multiple outputs when you naturally compute several related objects
- Avoid putting unrelated stuff in the same function just because you can

Functions With Parameters

When you have parameters in your main script, you need to pass them explicitly

```
function out = ces_agg(x, y, alpha, sigma)
% CES aggregator in two goods.
    out = ( alpha^(1/sigma)  * x.^((sigma-1)/sigma) ...     % ... allows a linebreak
        + (1-alpha)^(1/sigma) * y.^((sigma-1)/sigma) ).^(sigma/(sigma-1));

end
```

```
alpha = 0.3;
sigma = 0.8;

u = ces_agg(1, 2, alpha, sigma);
```

Good practice

- Always pass parameters as arguments
- This makes functions easier to test, reuse, and reason about

Scripts Calling Functions

Write a short script that uses some functions utility and production

```
clear all; close all; clc;
% Parameters
alpha = 0.36;
gamma = 2;
% Grid
k = linspace(0.1, 5, 100)';
% Computations
y  = production(k, alpha);
u  = utility(y, gamma);
% Plot
plot(k, u)
```

Separation of roles

- Scripts: set parameters, build grids, call functions, make plots
- Functions: take inputs, return outputs

Basic Plots

Plots are how we see what our code is doing

```
x = 0:0.1:10;
y = sin(x);

figure;                     % Open a new figure window
plot(x, y);                 % Simple line plot

title('Sine function');
xlabel('x');
ylabel('sin(x)');
```

- `figure` opens a fresh plotting window (optional but often useful)
- `plot` takes x- and y-coordinates of the curve
- Your grid determines how fine your plot is. MATLAB interpolates linearly
- Always label axes and give a title in anything you show other people

Multiple Lines

You will often compare several series in the same figure

```
x  = 0:0.1:10;
y1 = sin(x);
y2 = cos(x);

figure;
hold on;                           % Layering multiple plots
plot(x, y1, '-',);                 % Solid line
plot(x, y2, '--');                 % Dashed line
grid on;                           % Add grid lines
legend('sin(x)', 'cos(x)', ...
'Location', 'best');
xlabel('x');
ylabel('Value');
title('Sine and Cosine');
hold off;
```

Alice and Bob

The Economy

Two goods: apples x and potatoes y

- Alice has $(\bar{x}_A, \bar{y}_A)$ and utility $u_A(x, y) = \alpha_A \log x + (1 - \alpha_A) \log y$
- Bob has $(\bar{x}_B, \bar{y}_B)$ and utility $u_B(x, y) = \alpha_B \log x + (1 - \alpha_B) \log y$
- Apples are the numeraire: $p_x = 1$ and p_y is the potato price

Utility and marginal utility

With Cobb-Douglas utility

$$u(x, y) = \alpha \log x + (1 - \alpha) \log y$$

marginal utility is given as

$$MU_x(x, y) = \frac{\partial u}{\partial x} = \frac{\alpha}{x} \quad \text{and} \quad MU_y(x, y) = \frac{\partial u}{\partial y} = \frac{1 - \alpha}{y}$$

Log Utility and Marginal Utility

We start with Cobb–Douglas (log) utility.

Utility and marginal utility

$$u(x, y) = \alpha \log x + (1 - \alpha) \log y$$

$$MU_x(x, y) = \frac{\partial u}{\partial x} = \frac{\alpha}{x}, \qquad MU_y(x, y) = \frac{\partial u}{\partial y} = \frac{1 - \alpha}{y}$$

- MU_x and MU_y measure how utility changes with a marginal unit of each good.
- They are the building blocks for the MRS and ultimately demand.

Step 1: Parameters and Utility Code

Put primitives and endowments into a script

```
clear all; close all; clc

xbar_A = 2;        % Alice's endowment of apples
xbar_B = 4;        % Bob's endowment of apples

ybar_A = 5;        % Alice's endowment of potatoes
ybar_B = 3;        % Bob's endowment of potatoes

alpha_A = 1/2;     % Alice's preference for apples
alpha_B = 1/3;     % Bob's preference for apples

px = 1;            % Prices (apples are numeraire, py is endogenous)

u      = @(x,y,alpha) alpha.*log(x) + (1-alpha).*log(y); % Utility
mu_x   = @(x,y,alpha) alpha./x;                          % Marginal utility apples
mu_y   = @(x,y,alpha) (1-alpha)./y;                      % Marginal utility potatoes
```

Let's Plot Some Utilities

Put primitives and endowments into a script

```
x_grid = linspace(0,4,1000);     % Grid for apples from 0 to 4
y_grid = linspace(0,4,1000);     % Grid for potatoes from 0 to 4

plot(xgrid,u(x_grid,1,alpha_A)) % Alice's utility from 0 to 4 apples at 1 potato
plot(ygrid,u(2,y_grid,alpha_A)) % Alice's utility from 0 to 4 pototoes at 2 apples
plot(ygrid,u(2,y_grid,alpha_B)) % Bob's utility from 0 to 4 pototoes at 2 apples

plot(xgrid,mu_x(x_grid,1,alpha_A)) % Alice's mu from 0 to 4 apples at 1 potato
plot(xgrid,mu_y(x_grid,1,alpha_A)) % Alice's mu from 0 to 4 apples at 1 potato
```

Step 2: MRS and Willingness to Trade

The marginal rate of substitution (MRS)

$$\text{MRS}_{xy}(x,y) = \frac{MU_x(x,y)}{MU_y(x,y)} = \frac{\alpha}{1-\alpha} \cdot \frac{y}{x}$$

```
% MRS as a function of apples, potatoes, and preference parameters
mrs_xy = @(x,y,alpha) mu_x(x,y,alpha)./mu_y(x,y,alpha);

% Alice's MRS at her endowment
mrs_A  = mrs_xy(xbar_A,ybar_A,alpha_A)

% Bob's MRS at his endowment
mrs_B  = mrs_xy(xbar_B,ybar_B,alpha_B)
```

- Do Alice and Bob want to trade?
- Who would like to exchange apples for potatoes?

Step 3: Individual Demand

Cobb–Douglas (log) demand

- Budget is $\bar{x} + p_y \bar{y}$. Demand for potatoes:

$$y(p_y) = (1 - \alpha)\,\frac{\bar{x} + p_y \bar{y}}{p_y}$$

- Similarly, apples demand is $x(p_y) = \alpha\,(\bar{x} + p_y \bar{y})$

```
mA = @(py) xbar_A + py.*ybar_A;    % Alice's budget as a function of potato price
mB = @(py) xbar_B + py.*ybar_B;    % Bob's budget as a function of potato price

% Individual potato demands
y_demand = @(py,alpha,xbar,ybar) (1-alpha).*(xbar + py.*ybar)./py;

yA = @(py) y_demand(py,alpha_A,xbar_A,ybar_A); % Alice's demand as a function of py
yB = @(py) y_demand(py,alpha_B,xbar_B,ybar_B); % Bob's demand as a function of py
```

Step 4: Aggregate (Excess) Demand

Aggregate and excess demand

- Total endowment of potatoes: $\bar{y} = \bar{y}_A + \bar{y}_B$
- Aggregate demand: $y(p_y) = y_A(p_y) + y_B(p_y)$
- Excess demand: $z(p_y) = y(p_y) - \bar{y}$

```
Ybar   = ybar_A + ybar_B;

agg_y  = @(py) yA(py) + yB(py);   % Aggregate demand
excess = @(py) agg_y(py) - Ybar; % Excess demand in potatoes
```

- Market clearing requires $z(p_y) = 0$
- Here, we could solve $z(p_y) = 0$ analytically (and we did, in the last lecture)
- But we can also solve it numerically using MATLAB's `fzero`
- Before solving, it is good practice to plot $z(p_y)$

Step 5: Plot and Equilibrium Price

Visualize the zero of excess demand and then solve precisely

```
py_grid = linspace(0.1,5,500);

figure;
plot(py_grid, excess(py_grid));
yline(0);
xlabel('p_y'); ylabel('Excess demand for y');
title('Excess demand in Cobb-Douglas economy');
```

```
% This is an numeric root finder, we will spend some time on this
py_star = fzero(excess,1);    % 1 is an initial guess

% Equilibrium allocations
yA_star = yA(py_star);
yB_star = yB(py_star);
```

- The code walks from primitives $\rightarrow$ MU $\rightarrow$ MRS $\rightarrow$ demand $\rightarrow$ equilibrium

Comparative Statics

What happens when we change some primitive: think of shifts in demand and supply

- Higher $\bar{y}_A$ or $\bar{y}_B$ (more potatoes) lowers $p_y^\star$.
- Higher α (stronger taste for apples) lowers potato demand and $p_y^\star$.
- Higher $\bar{x}$ (more apples) raises income and hence potato demand, increasing $p_y^\star$.

How to check in code

- Change one primitive, rerun the script, and record $p_y^\star$
- This is exactly the computational comparative statics you will do in models

CES Preferences

CES Utility and MRS

CES utility changes substitution behavior.

CES utility and MRS

- Utility:

$$u(x, y) = \left(\alpha^{1/\sigma} x^{(\sigma-1)/\sigma} + (1-\alpha)^{1/\sigma} y^{(\sigma-1)/\sigma}\right)^{\sigma/(\sigma-1)}$$

- Marginal utilities are messy, but the MRS has a simple form:

$$\mathrm{MRS}_{xy}(x, y) = \frac{MU_x}{MU_y} = \left(\frac{\alpha}{1-\alpha}\right)^{1/\sigma} \left(\frac{y}{x}\right)^{1/\sigma}.$$

- As σ increases, goods become closer substitutes.

CES Step 1: Parameters and Utility in Code

We first add substitution parameters and a CES utility handle

```
sigma_A = 0.8;      % Alice's elasticity of substitution
sigma_B = 1.2;      % Bob's elasticity of substitution

% Example: CES utility for Alice
ces_u_A = @(x,y) ( alpha_A^(1/sigma_A).*x.^((sigma_A-1)/sigma_A) ...
                 + (1-alpha_A)^(1/sigma_A).*y.^((sigma_A-1)/sigma_A) ) ...
                 .^(sigma_A/(sigma_A-1));
```

- We could code marginal utilities explicitly, but we will jump straight to demand
- The equilibrium logic is the same: prices, income, and optimal shares

CES Step 2: Demand Shares

In a CES world, demands can be written in terms of expenditure shares

CES expenditure shares (two-good case)

- Prices are $(p_x, p_y) = (1, p_y)$ and income is $m = \bar{x} + p_y \bar{y}$
- Expenditure share on apples:

$$s_x(p_y) = \frac{\alpha^\sigma p_x^{1-\sigma}}{\alpha^\sigma p_x^{1-\sigma} + (1-\alpha)^\sigma p_y^{1-\sigma}}$$

- Expenditure share on potatoes:

$$s_y(p_y) = \frac{(1-\alpha)^\sigma p_y^{1-\sigma}}{\alpha^\sigma p_x^{1-\sigma} + (1-\alpha)^\sigma p_y^{1-\sigma}}$$

- Quantity demanded of potatoes:

$$y(p_y) = \frac{s_y(p_y)\, m}{p_y}$$

CES Step 3: Individual and Aggregate Demand

Implement CES demand for potatoes for Alice and Bob

```
% CES share of expenditure on potatoes
ces_share_y = @(py,alpha,sig) ...
    ( (1-alpha).^sig .* py.^(1-sig) ) ./ ...
    ( alpha.^sig .* 1.^(1-sig) + (1-alpha).^sig .* py.^(1-sig) );

% CES potato demand for one agent
ces_y_demand = @(py,alpha,sig,xbar,ybar) ...
    ces_share_y(py,alpha,sig) .* (xbar + py.*ybar)./py;

% Alice and Bob
yA_ces = @(py) ces_y_demand(py,alpha_A,sigma_A,xbar_A,ybar_A);
yB_ces = @(py) ces_y_demand(py,alpha_B,sigma_B,xbar_B,ybar_B);

% Aggregate CES demand and excess demand
agg_y_ces  = @(py) yA_ces(py) + yB_ces(py);
excess_ces = @(py) agg_y_ces(py) - Ybar;
```

- Same structure as in the Cobb-Douglas case; only the share formula changes

CES Step 4: Equilibrium Price via Root-Finding

We solve again for the price that clears the potato market.

```
py_grid = linspace(0.1,5,500);

figure;
plot(py_grid, excess_ces(py_grid));
yline(0);
xlabel('p_y'); ylabel('Excess demand for y (CES)');
title('Excess demand under CES preferences');
```

```
% Numerical equilibrium price under CES
py_star_ces = fzero(excess_ces,1);  % initial guess at 1

% Equilibrium CES allocations
yA_star_ces = yA_ces(py_star_ces);
yB_star_ces = yB_ces(py_star_ces);
```

- Pipeline is unchanged: demand → excess demand → root-finding.
- σ controls how sensitive demand is to relative prices.