Computational Tools for Macroeconomics using MATLAB

Week 2 – Programming Basics: Loops, Conditionals, Functions

Cristiano Cantore

Sapienza University of Rome

Recap – Week 1

- Introduction to MATLAB interface: Command Window, Workspace, Editor.
- Created simple scripts and ran them from the Editor.
- Defined and manipulated scalars, vectors, and matrices.
- Used MATLAB Help functions: `help`, `doc`, `lookfor`.
- Produced basic plots.

Recap – Week 1 (cont.)

- Learned to save and reload the Workspace.
- Saved and loaded data from `.mat` and `.csv` files.
- Started with simple economic examples (production function, GDP data).
- **Homework**: How did it go?

Why Programming Structures?

- **Automation:** Let MATLAB do repetitive tasks for you.
- **Avoid Repetition:** Write code once, use it many times.
- **Modularity:** Break problems into smaller, reusable functions.
- **Clarity:** Well-structured code is easier to read, debug, and maintain.
- **Efficiency:** Loops and conditionals allow complex computations with little code.

Week2 Learning Outcomes

By the end of this week, students will be able to:

1. Use `if`/`else` statements.
2. Write `for` and `while` loops.
3. Create MATLAB functions.
4. Debug and profile MATLAB code.
5. Understand good coding practices for reproducibility.

Example: GDP Growth over 10 Years

Manual approach (tedious):

```
GDP1 = GDP0 * (1+g);
GDP2 = GDP1 * (1+g);
GDP3 = GDP2 * (1+g);
...
GDP10 = GDP9 * (1+g);
```

Example: GDP Growth over 10 Years

Manual approach (tedious):

```
GDP1 = GDP0 * (1+g);
GDP2 = GDP1 * (1+g);
GDP3 = GDP2 * (1+g);
...
GDP10 = GDP9 * (1+g);
```

With a loop (efficient):

```
GDP(1) = GDP0;
for t = 2:10
    GDP(t) = GDP(t-1) * (1+g);
end
```

Example: GDP Growth over 10 Years

Manual approach (tedious):

```
GDP1 = GDP0 * (1+g);
GDP2 = GDP1 * (1+g);
GDP3 = GDP2 * (1+g);
...
GDP10 = GDP9 * (1+g);
```

With a loop (efficient):

```
GDP(1) = GDP0;
for t = 2:10
    GDP(t) = GDP(t-1) * (1+g);
end
```

- Manual: repetitive, error-prone, not scalable.
- Loop: concise, flexible (works for any horizon).

Booleans and Logical Operators

- A **Boolean** is a truth value: `true` (1) or `false` (0).
- Generated by **relational operators:** `<`, `<=`, `>`, `>=`, `==`, `=`.
- Combined with **logical operators:**
  - `&` – AND (elementwise)
  - `|` – OR (elementwise)
  - `~` – NOT (negation)
- Example:

```
x = 5; y = 10;
(x < y) & (y < 20)    % true
~(x == y)             % true
```

If / Else: Syntax

Basic pattern

```
if condition
    % statements
elseif other_condition
    % statements
else
    % statements
end
```

Relational: `<` `<=` `>` `>=` `==` `=` **Logical:** `&` `|` `~` **Short-circuit:** `&&` `||`

- `&` / `|` operate elementwise on arrays; `&&` / `||` compare single booleans.
- Use `isequal`, `isnan`, `isempty` for robust checks.

Short-Circuit Operators: `&&` and `||`

- Used only with scalar Booleans.
- MATLAB stops evaluating once the result is known.
- Helps avoid unnecessary or unsafe computations.

Example: Safe Division

```
x = 5;
y = 0;

if (y ~= 0) && (x/y > 1)
    disp('True')
else
    disp('False')
end
```

Short-Circuit Operators: `&&` and `||`

- Used only with scalar Booleans.
- MATLAB stops evaluating once the result is known.
- Helps avoid unnecessary or unsafe computations.

Example: Safe Division

```
x = 5;
y = 0;

if (y ~= 0) && (x/y > 1)
    disp('True')
else
    disp('False')
end
```

- With `&&`, second condition is skipped if `y =0` is false.
- Prevents division by zero error.
- With `&`, both conditions are always evaluated $\rightarrow$ error.

Example: Classify Growth

```
g = 100 * (GDP(2:end)./GDP(1:end-1) - 1);  % percent growth
lab = strings(size(g));
for t = 1:numel(g)
    if g(t) > 0
        lab(t) = "Expansion";
    elseif g(t) < 0
        lab(t) = "Contraction";
    else
        lab(t) = "Flat";
    end
end
```

- Replace the loop with vectorised logic later (practice).
- Discuss ties / near-zero: thresholding with `abs(g)<1e-6`.

Input Validation Pattern (for functions) - we will return on this

```
function FV = compound_interest(P, r, n)
% COMPOUND_INTEREST  FV = P * (1+r)^n
% Example: FV = compound_interest(100, 0.05, 10);

% --- input checks
if ~isscalar(P) || P <= 0,      error(’P must be positive scalar
if ~isscalar(r) || r <= -1,     error(’r > -1 required’);
if ~isscalar(n) || n < 0 || fix(n) ~= n
    error(’n must be a nonnegative integer’);
end

FV = P * (1 + r)^n;
end
```

- Use `error`/`warning`/`assert` to fail fast.
- Add a help block (first commented lines) for documentation.

Debugging Tip for Conditionals

- Set a **breakpoint** on the `if` line; inspect variables when the branch is taken.
- Stop automatically on errors:

```
>> dbstop if error
>> run('week2_debug.m')
K>>  % MATLAB is now paused in debug mode at the error line
```

- Step through with `Step In/Over/Out`; watch `lab(t)` change.
- Common gotcha: using `&&` on vectors (works only for scalars) — use `&` for elementwise logical operations.

Practical Example: Debugging Growth Classification

Buggy code:

```
g = 100 * (GDP(2:end)./GDP(1:end-1) - 1);
lab = strings(size(g));
if g > 0 && g < 5
    lab = "Moderate expansion";
end
```

Practical Example: Debugging Growth Classification

Buggy code:

```
g = 100 * (GDP(2:end)./GDP(1:end-1) - 1);
lab = strings(size(g));
if g > 0 && g < 5
    lab = "Moderate expansion";
end
```

Problem:

- `g` is a **vector**, but `&&` only works for scalars.
- MATLAB throws: `Operands to the && operator must be convertible to logical scalar values.`

Practical Example: Debugging Growth Classification

Buggy code:

```
g = 100 * (GDP(2:end)./GDP(1:end-1) - 1);
lab = strings(size(g));
if g > 0 && g < 5
    lab = "Moderate expansion";
end
```

Problem:

- `g` is a **vector**, but `&&` only works for scalars.
- MATLAB throws: `Operands to the && operator must be convertible to logical scalar values.`

Debugging with breakpoints:

- Place a breakpoint on the `if` line.
- Inspect `g`: confirm it's a vector.
- Fix by using elementwise logic:

```
idx = (g > 0) & (g < 5);
lab(idx) = "Moderate expansion";
```

For Loops — Syntax

Pattern

```
for i = start:step:finish
    % statements using i
end
```

- Common shorthand: `for i = 1:N` (step defaults to 1).
- Loop variable `i` is a scalar (changes each iteration).
- Prefer preallocation for arrays you fill inside loops.

For Loops — Demo: First 10 Squares

```
N = 10;
sq = zeros(1, N);          % preallocate
for i = 1:N
    sq(i) = i^2;
end

disp(sq)    % [1 4 9 16 25 36 49 64 81 100]
```

- Preallocation avoids growing `sq` in each iteration.
- Equivalent vectorised form: `sq = (1:N).^2;`

Exercise — First 10 Fibonacci Numbers

- Compute the sequence $F_1 = 1, F_2 = 1, \; F_t = F_{t-1} + F_{t-2}$ for $t = 3, \ldots, 10$.
- Store results in a row vector `F` of length 10.

```
N = 10;
F = zeros(1, N);    % preallocate
F(1) = 1; F(2) = 1;

for t = 3:N
    F(t) = F(t-1) + F(t-2);
end

disp(F)
```

While Loops — Syntax

Pattern

```
while condition
    % statements
end
```

- Use when the number of iterations is not known in advance.
- Always ensure the condition will eventually become `false`.

While Loops — Demo: Doubling GDP to a Threshold

```
GDP0      = 100;      % initial level
g         = 0.05;     % 5% growth per period
threshold = 200;

GDP = GDP0;
t   = 0;
while GDP < threshold
    GDP = GDP * (1 + g);
    t   = t + 1;
end

fprintf('Reached %.1f after %d periods.\n', GDP, t);
```

- Classic use case: keep iterating until a stopping rule is met.

Performance Note — Loops vs. Vectorisation

```
rng(123);                    % reproducibility
x = rand(1e6,1);             % 1 million draws

% Loop sum
tic
s1 = 0;
for i = 1:numel(x)
    s1 = s1 + x(i);
end
t_loop = toc;

% Vectorised sum
tic
s2 = sum(x);
t_vec = toc;

fprintf('loop: %.4fs | vectorised: %.4fs | diff = %.3g\n', ...
        t_loop, t_vec, abs(s1-s2));
```

- Vectorised code is usually faster and clearer.
- Use loops when logic is sequential or complex; still preallocate.

Functions: Concept

- **Modular blocks of code**: encapsulate a task once, reuse many times.
- **Inputs → outputs**: clear interfaces make code reliable and testable.
- **Local workspace**: variables inside a function do not leak to base workspace.
- **Reproducibility**: functions + fixed `rng` seeds + saved scripts.

Function Syntax (Anatomy)

Basic pattern (in a file `myFunction.m`)

```
function out = myFunction(in1, in2)
% MYFUNCTION  One-line description
%   out = MYFUNCTION(in1, in2) returns in1 + in2.

    out = in1 + in2;
end
```

- File name must match the main function name.
- First comment block is the **help text** shown by `help myFunction`.

Demo: `compound_interest(P,r,n)`

File: `compound_interest.m`

```
function FV = compound_interest(P, r, n)
% COMPOUND_INTEREST  FV = P * (1 + r)^n
% FV = COMPOUND_INTEREST(P, r, n) computes the final value
% of principal P after n periods at rate r (per period).

    FV = P * (1 + r)^n;
end
```

- Call from script or Command Window:

```
FV = compound_interest(100, 0.05, 10);
```

Exercise: Input Validation with `if`

Extend `compound_interest.m`

```
function FV = compound_interest(P, r, n)
% FV = COMPOUND_INTEREST(P, r, n) computes the final value
% of principal P after n periods at rate r (per period).
% Example: FV = compound_interest(100, 0.05, 10);

% --- input checks
if ~isscalar(P) || P <= 0,      error('P must be positive scalar
if ~isscalar(r) || r <= -1,     error('r > -1 required');
if ~isscalar(n) || n < 0 || fix(n) ~= n
    error('n must be a nonnegative integer');
end

FV = P * (1 + r)^n;
end
```

- Add a brief **help text** at the top and 1–2 usage examples.

Debugging Functions: `dbstop` and `dbstep`

- Stop at the source of errors:

```
dbstop if error
```

- Run your script that calls the function; MATLAB pauses on the error line inside the function.
- Inspect variables in the Workspace; hover for tooltips.
- Step through execution:

```
dbstep        % step to next line
dbstep in     % step into a called function
dbstep out    % step out to caller
dbquit        % exit debug mode
```

- Tip: place a **breakpoint** on a suspicious line or use `dbstop in compound_interest at 10`.

Anonymous Functions

- **One-liner functions**, defined directly in the Command Window or script.
- Syntax:

```
f = @(x) x.^2 + 1;
f(3)    % returns 10
```

- Accept multiple inputs:

```
u = @(c, alpha) (c.^(1-alpha) - 1) / (1-alpha);
u(2, 0.5)    % CRRA utility
```

- Useful for:
  - Quick experiments without creating a new `.m` file.
  - Passing functions as arguments (e.g. to solvers or optimizers).
  - Compact mathematical expressions.

Why Good Practices Matter

- Code is read more often than it is written.
- Clear structure makes debugging and collaboration easier.
- Reproducibility: you (and others) can rerun analyses later.

Comments and Help Text

- Use `%` for inline comments.
- At the start of a function, write a block of comments as documentation.

```
function FV = compound_interest(P, r, n)
% COMPOUND_INTEREST computes future value of an investment.
%   FV = compound_interest(P, r, n) returns the value of
%   principal P invested at rate r for n periods.
%
%   Example:
%   FV = compound_interest(100, 0.05, 10);
```

Naming Conventions

- Use descriptive names: `GDP_growth`, not `x`.
- Functions: verbs (`computeGDP`, `plotResults`).
- Constants: ALL_CAPS if useful (`PI`, `MAX_ITER`).
- Stick to consistent style (camelCase, snake_case, etc.).

Reproducibility

- Save scripts and functions with meaningful names.
- Save figures (`saveas`, `exportgraphics`).
- Keep track of versions (consider Git later).
- Record random seeds if simulations are used (`rng(123)`).

Project Structure

Suggested layout:

- `/code` – scripts, functions
- `/data` – raw and processed data
- `/figures` – plots, outputs
- `/docs` – notes, reports

Good Practices Checklist

- **Comment your code:** explain why, not just what.
- **Write help text:** every function should start with documentation.
- **Name things clearly:** avoid `x1`, `x2`, use descriptive names.
- **Stay consistent:** pick a naming style and stick to it.
- **Save outputs:** scripts, figures, and data files.
- **Organise projects:** separate code, data, and results into folders.
- **Reproducibility:** set random seeds, keep track of versions.

"Write code your future self will thank you for."

Challenge (15 min)

Task: Simulate a GDP path with shocks and plot it.

- Write a function `gdp_simulate(G0, g, sigma, T, seed)` that returns a vector `G` of length `T+1`.
- Model: $G_{t+1} = G_t \cdot (1 + g + \varepsilon_t)$, where $\varepsilon_t \sim \mathcal{N}(0, \sigma^2)$.
- Use `rng(123)` for reproducibility and `randn` for shocks.
- Run for `T=20` periods with chosen `G0, g, sigma`, and **plot** the path.

Deliverables:

- Function file: `gdp_simulate.m`
- Driver script: `week2_challenge.m` that calls the function and makes the plot.

Starter Code (students complete)

```
% File: gdp_simulate_starter.m
function G = gdp_simulate_starter(G0, g, sigma, T,seed)
% GDP_SIMULATE  Simulate GDP path with shocks over T periods.
%   G = GDP_SIMULATE(G0, g, sigma, T, seed) returns a column vector of
%   length T+1 with G(1) = G0.
%
%   Model: G(t+1) = G(t) * (1 + g + eps_t),  eps_t ~ N(0, sigma^2).
%   Optional: provide 'seed' for reproducibility.

% --- Input checks (optional) ---
%   if ~isscalar(G0) || G0 <= 0, error('G0>0 required'); end
%   if ~isscalar(T)  || T < 1 || fix(T)~=T, error('T integer >= 1'); end


    % --- Generate shocks ---
    %>>> Students complete the update below <<<
    % eps = ...

    % --- Initialize path ---
    G    = zeros(T+1,1);
    G(1) = G0;

    % --- Loop to simulate path ---
    for t = 1:T
        % >>> Students complete the update below <<<
        % G(t+1) = ...
    end
end
```

Starter Code (students complete)

```
% File: week2_challenge_starter.m (driver)
% choose values
    %>>> Students complete below <<<

% Call the function (students will need to complete it first!)
    %>>> Students complete below <<<

% Plot the result
    %>>> Students complete below <<<
```

Homework / Practice

1. Write a function that simulates GDP growth over T periods with shocks. (can reuse the one from class)
2. Create a script that:
   * Simulates **100** GDP paths.
   * Computes **mean** & **variance** of final GDP (GDP in last period T).
   * Plots a **histogram** of final GDP values.
3. Document with comments and **save plots** as PNG and MATLAB figure.

Files & Deliverables

- **Function**: `gdp_simulate.m`
- **Driver script (starter)**: `week2_homework_starter.m`
- **Expected outputs**:
  - `week2_homework_solution.m`
  - `week2_final_gdp_hist.png`, `week2_final_gdp_hist.fig`
  - (Optional) `week2_homework_workspace.mat`
- Keep your code reproducible: set `rng(123)` in the driver.

Next Week

Next week: **Block B – Numerical Tools & Data Handling**
Week 3 – Data Input/Output & Plotting