Quantitative Macroeconomics I
Bootcamp 1: Introduction to Matlab

Grégoire Sempé

gregoire.sempe@psemail.eu

Paris School of Economics, Université Paris 1 Panthéon-Sorbonne

September 15, 2025

I thank Tobias Broer, Eustache Elina and Moritz Scheidenberger for useful materials and discussions.

Nice to meet you!

Administrative precisions:

- Fill-in your contact details using the spreadsheet !
- Guidelines to get access to Matlab:
  1. Download the free trial version of Matlab `https://www.mathworks.com/campaigns/products/trials.html?country=france`
  2. After registering to the class on course website. Contact IT (`support_info@psemail.eu`) with proof of registration.
  3. You will get an appointment with IT to set-up your computer.
- Check your emails very regularly (updates, schedule, ...)!

What **you** should expect from the tutorials

Format of the tutorials sessions:

Learning Objectives:

What **you** should expect from the tutorials

Format of the tutorials sessions:

- Designed as complementary to Prof Broer's class $\rightarrow$ attend both classes!

Learning Objectives:

What **you** should expect from the tutorials

Format of the tutorials sessions:

- Designed as complementary to Prof Broer's class $\rightarrow$ attend both classes!
- A mix of theory, coding & problem sets $\rightarrow$ expect **a lot** of work

Learning Objectives:

What **you** should expect from the tutorials

Format of the tutorials sessions:

- Designed as complementary to Prof Broer's class $\rightarrow$ attend both classes!

- A mix of theory, coding & problem sets $\rightarrow$ expect <u>a lot</u> of work

Learning Objectives:

1. Create your toolbox to solve macroeconomic models (QM1: representative agent)
   $\hookrightarrow$ Various numerical methods, with advantages and drawbacks
   $\Rightarrow$ **QM2 will build on QM1** and will focus on state-of-the-art heterogenous agent models

What **you** should expect from the tutorials

Format of the tutorials sessions:

- Designed as complementary to Prof Broer's class $\rightarrow$ attend both classes!
- A mix of theory, coding & problem sets $\rightarrow$ expect **a lot** of work

Learning Objectives:

1. Create your toolbox to solve macroeconomic models (QM1: representative agent)
   $\hookrightarrow$ Various numerical methods, with advantages and drawbacks
   $\Rightarrow$ **QM2 will build on QM1** and will focus on state-of-the-art heterogenous agent models
2. Become familiar with dynamic programming / recursive methods
   $\hookrightarrow$ Dominant in macro, widely used in labor, econ theory and structural econometrics …

What **you** should expect from the tutorials

Format of the tutorials sessions:

- Designed as complementary to Prof Broer's class $\rightarrow$ attend both classes!
- A mix of theory, coding & problem sets $\rightarrow$ expect a lot of work

Learning Objectives:

1. Create your toolbox to solve macroeconomic models (QM1: representative agent)
   $\hookrightarrow$ Various numerical methods, with advantages and drawbacks
   $\Rightarrow$ **QM2 will build on QM1** and will focus on state-of-the-art heterogenous agent models
2. Become familiar with dynamic programming / recursive methods
   $\hookrightarrow$ Dominant in macro, widely used in labor, econ theory and structural econometrics …

$\Rightarrow$ Be able to solve state-of-the-art models (used by central banks, academic research, …)

What **we** expect from you

How to succeed in this course?

1. Attend classes, and **ask questions** if you don't understand!

What **we** expect from you

How to succeed in this course?

1. Attend classes, and **ask questions** if you don't understand!

2. Study the exercises and problem sets, do them by yourself, code regularly

What **we** expect from you

How to succeed in this course?

1. Attend classes, and **ask questions** if you don't understand!

2. Study the exercises and problem sets, do them by yourself, code regularly

3. Connect the computational methods to economic models! Ask us if you don't see the connection

What **we** expect from you

How to succeed in this course?

1. Attend classes, and **ask questions** if you don't understand!
2. Study the exercises and problem sets, do them by yourself, code regularly
3. Connect the computational methods to economic models! Ask us if you don't see the connection
4. If you notice some typos or mistakes in the slides/assignments, send us directly an email!

   We are continuously improving the materials and are putting a lot of effort in teaching this class; typos/mistakes are unavoidable but we want to minimize them.

One more thing…

Some advice

Useful resources (more on the syllabus!)

One more thing…

Some advice

- Learning quantitative macro has a **large fixed cost** $\Rightarrow$ Need to invest to benefit from the class
- Help each other to understand the course and methods is the best way to learn

Useful resources (more on the syllabus!)

One more thing...

Some advice

- Learning quantitative macro has a **large fixed cost** $\Rightarrow$ Need to invest to benefit from the class
- Help each other to understand the course and methods is the best way to learn

Useful resources (more on the syllabus!)

- Textbooks: Heer and Maussner *Dynamic GE modelling*, Azzimonti et al *Macroeconomics*
- QuantEcon lecture notes (Sargent, Stachurski): `https://quantecon.org/lectures/`
- Advanced course materials (computational methods)
  - Jesus Fernandez-Villaverde (Princeton): `https://www.sas.upenn.edu/~jesusfv/teaching.html`
  - Fatih Guvenen (Minnesota): `https://www.fatihguvenen.com/phd-computational-methods`

Grading – *To be confirmed*

- End-of-semester exam 50% of the final grade
- Around 3/4 problem sets, do be done **by groups of 2** 50% of the final grade

  ↪ Even if you don't manage to solve the hardest problems, I expect to see some effort

  → Follow the general indications on the website on the formatting of the problem sets

Outline for today's bootcamp

1. Motivation & general takes on software
2. Matlab basics (matrices, operations)
3. First exercise: Putting Solow into the computer
   - Arrays, matrices, functions
   - Loops, conditional statements
   - Vectorization, plotting
3. Linear Interpolation & Vectorization

→ Exercise to warm-up!

Why learning numerical techniques?

1. Mathematical sciences always face a trade-off btw. realistic assumptions and solvability

   $\hookrightarrow$ Solving your model numerically *partially* solves this issue

Why learning numerical techniques?

1. Mathematical sciences always face a trade-off btw. realistic assumptions and solvability

   $\hookrightarrow$ Solving your model numerically *partially* solves this issue

2. Build an economic intuition by playing with your model

   a) In partial equilibrium, study the effects of prices on individual decisions

   b) In general equilibrium, study the effects of shocks (e.g. taxes) on prices & aggregates

Why learning numerical techniques?

1. Mathematical sciences always face a trade-off btw. realistic assumptions and solvability
   ↪ Solving your model numerically *partially* solves this issue
2. Build an economic intuition by playing with your model
   a) In partial equilibrium, study the effects of prices on individual decisions
   b) In general equilibrium, study the effects of shocks (e.g. taxes) on prices & aggregates
3. Makes you able to see the effects of a policy on the **distribution** (HA model)
   a) Effects of macro policies on inequalities (e.g. fiscal policy)
   b) Macroeconomic dynamics are heavily modified! (e.g monetary policy)

Why use Matlab?

<u>Pros</u>:

1. Intuitive language
2. Easy to debug: easy to know what you are manipulating
3. Very efficient at handling matrices
4. Widespread use among macroeconomists (e.g central banks)

$\Rightarrow$ Probably not the most efficient language but good enough for simple models

Why use Matlab?

Pros:

1. Intuitive language
2. Easy to debug: easy to know what you are manipulating
3. Very efficient at handling matrices
4. Widespread use among macroeconomists (e.g central banks)

$\Rightarrow$ Probably not the most efficient language but good enough for simple models

Cons:

1. Not open source $\rightarrow$ expensive, code can break across versions in the long run
2. Relatively slow compared to low-level languages…
3. Hard to use together with other languages

$\Rightarrow$ Alternatives: **Julia**, Python – Numba, C++, JAX (*see Fernandez Villaverde*: `https://www.sas.upenn.edu/~jesusfv/Lecture_HPC_5_Scientific_Computing_Languages.pdf`)

Matlab Interface

Divided in four parts:

1. Command window: where you can type and execute commands directly
2. Editor: where you end up writing your code if you want to keep track of it.
   Note 1: You only use the command window for tests or debugging
   Note 2: Use comments starting with % for your future readers and for yourself!
   Note 3: End a line of code with ; if you don't want to see it printed in the command window
   $\rightarrow$ To run a script : Editor $>$ Run
3. Workspace: all variables, functions, matrices, etc. available to work with
4. Current folder: what scripts you have direct access to
   Note 4: Keep functions you use in your current folder or in the folder that you have included in your *search path* (Home $>$ Environment $>$ Set Path $>$ Add folders)
   Search path : files Matlab have access to

General functions

- Want to clear the workspace?

```
clear
```

- Want to clear the command window?

```
clc
```

- Want to save your workspace into a file named backup?

```
save backup.mat
```

- Want to load your file backup?

```
load backup.mat
```

When in trouble

- You have access to detailed explanations of any function when writing help or doc followed by the name of the function in the command window. Ex with clear function:

```
help clear
```

```
doc clear
```

- LLMs are quite good at explaining how functions work / giving examples...

↪ ChatGPT, but also open source alternatives: Mistral Codestral, Llama...

⟹ But always check if the answer provided is right!

MATlab: Building scalars, vectors and matrices

- Build a scalar:

```
a = 2;
```

- Build a row vector:

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

- Build a column vector:

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

- Build a matrix:

```
d = [1 2; 3 4];
```

Direct command to build matrices

- Construct a matrix of 0 of size $m \times n$

```
zeros(m,n)
```

- Construct a matrix of 1 of size $m \times n$

```
ones(m,n)
```

- Construct a matrix of size $m \times n$ of random draws from an uniform distribution in $[0, 1]$

```
rand(m,n)
```

How to navigate in a matrix: indexing 1/2

How to choose specific element(s) in a matrix? Define:

```
h = rand(10,10);
```

- How to pick the element on the 6th row and 7th column:

```
h(6,7)
```

- How to pick all the elements on column 4:

```
h(:,4)
```

Note: In Matlab indexing starts at 1 and not 0! ($\neq$ Python)

How to navigate in a matrix: indexing 2/2

- How to pick the first three rows in column 4

```
h(1:3,4)
```

- How to exclude the first and the last column:

```
h(:,2:end-1)
```

Other object: arrays

Generalization of matrices in more than two dimensions. Ex for an array in 3 dimensions:

```
h = rand(3,5,8);
```

$\rightarrow$ Can be visualised as a book of 8 pages with 3×5 elements of each page

Other object: structure array

A structure array is composed of several fields that can each contain any type of data.

$\rightarrow$ Use the dot when naming a variable to create a structure.

```
par.alpha = 0.3;
par.beta = 0.95;
par.delta = 0.1;
```

$\Longrightarrow$ Creates a structure *par* with all your parameters.

$\Longrightarrow$ Useful to pass parameters in an *user-written* function (see last section)

Operations

- Standard (matrix or scalar) operators '+', '-', '/', '\'*' '^'
- **Element-by-element operators** by adding a **dot** in front of the operator : '.*', './', '.^'
- Comparison operators
  - equal $==$
  - not equal $\sim$=
  - bigger or equal $>=$
  - smaller or equal $<=$

$\Rightarrow$ A comparison operation will yield either **1 if the condition is true** and 0 if not

Run Section

You can subdivide your code in different sections and run your code only in one specific section

1. Start a line with '%%' to create a section
2. Select a section and click on Editor > Run Section to run it

```
%% 1st section
A = 1;
%% 2nd section
B = rand;
```

Measure the time to run a code: tic toc

You can measure the time a code takes to run using the 'tic toc' function

1. Write tic and jump a line
2. Include the code you want to measure
3. Jump a line and write toc

```
tic
A = rand(10);
B = inv(A);
toc
```

Matlab debugger

- You can set breakpoints in your code to stop the execution at a specific line
- Click on the left of the line number to set a breakpoint (a red circle appears)
- Run your code and it will stop at the breakpoint
- You can then check the value of your variables in the workspace and run your code line by line using F10
- To remove a breakpoint, click again on the red circle!

More on this next week!

Example: Learning Matlab with the Solow Growth Model

Example 1

Objective: Solve the Solow model numerically to go beyond usual assumptions

Q1. Solve the steady state level of per-capita capital k^* using the first-difference equation

Q2. Find the saving rate that maximizes the steady state level of per-capita consumption c^*

Environment:

The steady state is unique and stable under usual assumptions.

Example 1

Objective: Solve the Solow model numerically to go beyond usual assumptions

Q1. Solve the steady state level of per-capita capital k^* using the first-difference equation

Q2. Find the saving rate that maximizes the steady state level of per-capita consumption c^*

Environment:

- Equations: $Y_t = F(K_t, L_t) = K^\alpha (L)^{1-\alpha}$, $I_t = sY_t$, $K_{t+1} = (1-\delta)K_t + I_t$, $L_{t+1} = (1+n)L_t$
- Parameters: α = 0.3, $\delta = 0.05$, $n = 0.02$
- Exogenous variables: $s \in (0, 1)$, initial value $s = 0.2$
- Endogenous variables: K_t, L_t, Y_t, C_t

The steady state is unique and stable under usual assumptions.

Q1: Solving the steady state of the Solow model

Define $k_t = K_t / L_t$, $y_t = Y_t / L_t$ and $c_t = C_t / L_t$. Then, we have

First difference equation

$$k_{t+1} = \frac{s}{1+n} k_t^{\alpha} + \frac{(1-\delta)}{(1+n)} k_t := g(k_t)$$

Steady State Condition

k^* is defined by $k_{t+1} = k_t = k^*$

Solution Methods (today):

Q1: Solving the steady state of the Solow model

Define $k_t = K_t/L_t$, $y_t = Y_t/L_t$ and $c_t = C_t/L_t$. Then, we have

First difference equation

$$k_{t+1} = \frac{s}{1+n}k_t^{\alpha} + \frac{(1-\delta)}{(1+n)}k_t := g(k_t)$$

Steady State Condition

k^* is defined by $k_{t+1} = k_t = k^*$

Solution Methods (today):

1. Forward iteration: start from k_0 and iterate on the FD equation until convergence to k^*

Q1: Solving the steady state of the Solow model

Define $k_t = K_t / L_t$, $y_t = Y_t / L_t$ and $c_t = C_t / L_t$. Then, we have

First difference equation

$$k_{t+1} = \frac{s}{1+n} k_t^{\alpha} + \frac{(1-\delta)}{(1+n)} k_t := g(k_t)$$

Steady State Condition

k^* is defined by $k_{t+1} = k_t = k^*$

Solution Methods (today):

1. Forward iteration: start from k_0 and iterate on the FD equation until convergence to k^*

   $\Rightarrow$ Need loops, conditional statements and a stopping criterion

Q1: Solving the steady state of the Solow model

Define $k_t = K_t/L_t$, $y_t = Y_t/L_t$ and $c_t = C_t/L_t$. Then, we have

First difference equation

$$k_{t+1} = \frac{s}{1+n}k_t^{\alpha} + \frac{(1-\delta)}{(1+n)}k_t := g(k_t)$$

Steady State Condition

k^* is defined by $k_{t+1} = k_t = k^*$

Solution Methods (today):

1. Forward iteration: start from k_0 and iterate on the FD equation until convergence to k^*
   - $\Rightarrow$ Need loops, conditional statements and a stopping criterion
2. Root-finding: find the root of $FP(k) = \frac{s}{1+n}k^{\alpha} + \frac{(1-\delta)}{(1+n)}k - k$

Q1: Solving the steady state of the Solow model

Define $k_t = K_t / L_t$, $y_t = Y_t / L_t$ and $c_t = C_t / L_t$. Then, we have

First difference equation	Steady State Condition
$k_{t+1} = \frac{s}{1+n} k_t^\alpha + \frac{(1-\delta)}{(1+n)} k_t := g(k_t)$	k^* is defined by $k_{t+1} = k_t = k^*$

Solution Methods (today):

1. Forward iteration: start from k_0 and iterate on the FD equation until convergence to k^*

   $\Rightarrow$ Need loops, conditional statements and a stopping criterion

2. Root-finding: find the root of $FP(k) = \frac{s}{1+n} k^\alpha + \frac{(1-\delta)}{(1+n)} k - k$

   $\Rightarrow$ Introduce plots. Need anonymous functions and to build a solver

1.1. Towards a Forward iteration algorithm

Step 1: Define parameters and initial values

```
par.alpha = 0.3;
par.delta = 0.05;
par.n = 0.02;
par.s = 0.2;

k0 = 0.5; % initial value of capital
```

Step 2: We want to iterate on the FD equation $k_{t+1} = g(k_t)$ until convergence.

”Iterate” $\rightarrow$ apply multiple times the same operation $\rightarrow$ **loop FOR**

”Until convergence” $\rightarrow$ need a **stopping criterion** $\rightarrow$ **conditional statement IF**

1.1. Iterate forward the FD equation...

Write a loop FOR to iterate, and store values of k_t at each iteration

```
% Initialization



% Iterate on the FD equation
for

end
```

1.1. Iterate forward the FD equation...

Write a loop FOR to iterate, and store values of k_t at each iteration

```
% Initialization
T = 1000;        % max number of iterations
k = zeros(T,1);      % pre-allocate memory
k(1) = k0;      % initial value of capital

% Iterate on the FD equation
for

end
```

1.1. Iterate forward the FD equation…

Write a loop FOR to iterate, and store values of k_t at each iteration

```
% Initialization
T = 1000;        % max number of iterations
k = zeros(T,1);       % pre-allocate memory
k(1) = k0;       % initial value of capital

% Iterate on the FD equation
for  t=1:T-1

end
```

1.1. Iterate forward the FD equation…

Write a loop FOR to iterate, and store values of k_t at each iteration

```
% Initialization
T = 1000;        % max number of iterations
k = zeros(T,1);      % pre-allocate memory
k(1) = k0;      % initial value of capital

% Iterate on the FD equation
for  t=1:T-1
  k(t+1) = par.s/(1+par.n)*k(t)^par.alpha + (1-par.delta)/(1+par.n)*k(t);
end
```

1.1. Iterate forward the FD equation... **until convergence**

Write a FOR loop to iterate on the FD equation, and store values of k_t at each iteration

```
% Initialization
for t=1:T-1
  k(t+1) = ( par.s * k(t)^par.alpha + (1-par.delta) * k(t) ) /(1+par.n);
end
```

1.1. Iterate forward the FD equation... **until convergence**

Write a FOR loop to iterate on the FD equation, and store values of k_t at each iteration

```
% Initialization
for t=1:T-1
  k(t+1) = ( par.s * k(t)^par.alpha + (1-par.delta) * k(t) ) /(1+par.n);
  err = abs(k(t+1)-k(t));
  disp(['Iteration: ', num2str(t), ', Error: ', num2str(err)]);




end
```

1.1. Iterate forward the FD equation…**until convergence**

Write a FOR loop to iterate on the FD equation, and store values of k_t at each iteration

```
% Initialization
for t=1:T-1
  k(t+1) = ( par.s * k(t)^par.alpha + (1-par.delta) * k(t) ) /(1+par.n);
  err = abs(k(t+1)-k(t));
  disp(['Iteration: ', num2str(t), ', Error: ', num2str(err)]);

% Stopping criterion




end
```

1.1. Iterate forward the FD equation... **until convergence**

Write a FOR loop to iterate on the FD equation, and store values of k_t at each iteration

```
% Initialization
for t=1:T-1
  k(t+1) = ( par.s * k(t)^par.alpha + (1-par.delta) * k(t) ) /(1+par.n);
  err = abs(k(t+1)-k(t));
  disp(['Iteration: ', num2str(t), ', Error: ', num2str(err)]);

% Stopping criterion
  if err < 1e-8
      % stop if converged
    break
  end
end
```

1.1. Plot the results

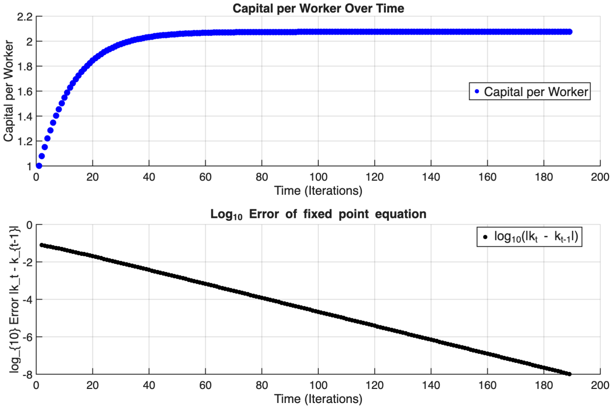


Figure: Convergence of k_t to k^* using forward iteration

- Very simple algorithm!
- Convergence takes 190 iterations at tolerance 10^{-8}
- Takes around 0.008 seconds to run

$\Rightarrow$ **Can we do better?**

1.2. Root-finding method – graphical intuition

Recall: Steady state condition k^* is defined by $k_{t+1} = k_t = k^*$

$\Rightarrow$ Let's plot $g(k) - k$ on a grid of k to find the root graphically

How would you do it?

1.2. Root-finding method – graphical intuition

Recall: Steady state condition k^* is defined by $k_{t+1} = k_t = k^*$

$\Rightarrow$ Let's plot $g(k) - k$ on a grid of k to find the root graphically

How would you do it?

1. Discretize the state space of k (*build a grid* $\rightarrow$ *vector in matlab*)

2. Compute $g(k) - k$ on the grid

3. Plot the function and find the root

1.2. Plot the Fixed Point function

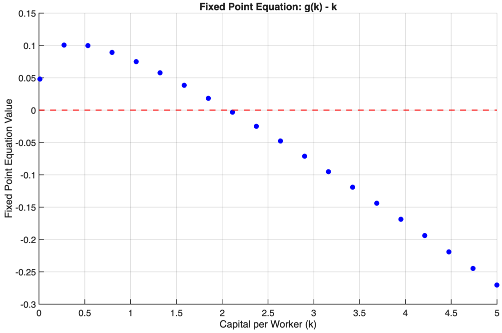


Figure: Plot of the fixed point function $g(k) - k$

```
% grid of 20 points btw. 0.01 and 5
k_grid = linspace(0.01, 5, 20);

% compute the fixed point function on the grid
fp_eq = @(k, par) (par.s * k.^(par.alpha )+ (1 - par.delta)
* k) / (1 + par.n) - k;
% evaluate the function on the grid

fp_val = fp_eq(k_grid, par);

% plot the function and the root...
...
```

$\Rightarrow$ There is clearly only one root that respects boundary conditions (i.e. $k^* > 0$)

Always useful to plot objects of interest in the process... *What can you notice here?*

1.2. Bisection vs Newton methods

Bisection method: *robust method, needs continuity, very slow*

- Start with an interval $[a, b]$ where the function changes sign (i.e. $f(a)f(b) < 0$)
- Compute the midpoint $c = (a + b)/2$ and evaluate $f(c)$
- If $f(a)f(c) < 0$ then set $b = c$, else set $a = c$
- Repeat until convergence

Newton method: *local method, needs a smooth function and a good initial guess, fast (aggressive)*

- Start from an initial guess k_0 and iterate on $k_{n+1} = k_n - FP(k_n)/FP'(k_n)$ until convergence
- Need to compute the derivative of the function $FP'(k)$ (1st order)

1.2. A detour to functions in Matlab

Anonymous functions: useful for simple functions you want to define quickly

```
fp_eq = @(k, par) (par.s * k.^(par.alpha )+ (1 - par.delta) * k) / (1 + par.n) - k;
```

User-defined functions: useful for more complex functions you want to keep and re-use

```
% In a separate file named 'my_function.m'
function y = my_function(x, par)
y = (par.s * x.^(par.alpha )+ (1 - par.delta) * x) / (1 + par.n) - x;
end
```

⇒ Call it in your main script as:

```
val = my_function(k_grid, par);
```

Script vs function: important differences

- What they use as **inputs**:
    - Functions only use the received inputs
    - Scripts have access to the whole workspace
- What they have as **output**:
    - Functions only give the demanded output and erase the rest $\Rightarrow$ local memory
    - Scripts return all variables used in it

Be careful with global variables in Matlab! They can lead to errors and be hard to debug...

1.2. Bisection vs Newton methods – results

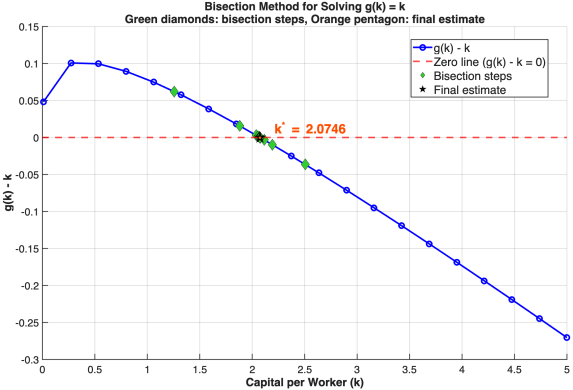


Figure: Bisection method

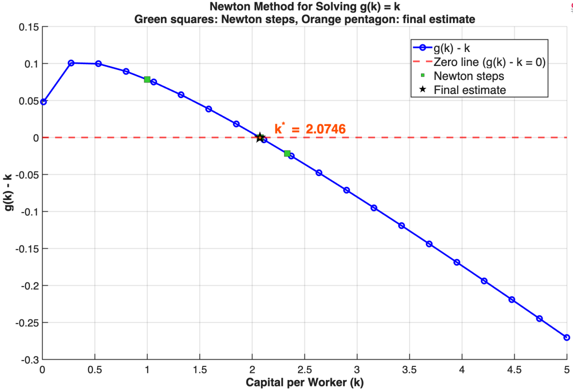


Figure: Newton method

- Bisection: 24 iterations to converge, 0.001 seconds to run (6x faster than forward iteration)
- Newton: 5 iterations to converge, 0.0004 seconds to run (4x faster than bisection)

Q2: Maximizing steady state consumption

Recall: Steady state consumption is defined as $c^* = (1 - s)f(k^*)$

$\Rightarrow$ We want to find the saving rate $s \in (0, 1)$ that maximizes c^*

Two solution methods:

1. Grid search: discretize the space of s and find the maximum
2. Off-grid solver : use a golden search solver to find the maximum

2.1. Grid search method

Step 1: Discretize the space of s

```
s_grid = linspace(0.01, 0.99, 100); % grid of 100 points between 0.01 and 0.99
```

Step 2: For each s in the grid, compute $k^*(s)$ using bisection and then compute $c^*(s)$

```
c_star = zeros(size(s_grid)); % pre-allocate memory
for i=1:length(s_grid)
  par.s = s_grid(i);
  k_star = bisection(@(k) fp_eq(k, par), 0.01, 5, 1e-8, 100);
  c_star(i) = (1 - par.s) * k_star^par.alpha;
end
```

Step 3: Find the maximum of c^* on the grid

```
c_max, idx = max(c_star);   s_opt = s_grid(idx);
```

2.2 Compare results

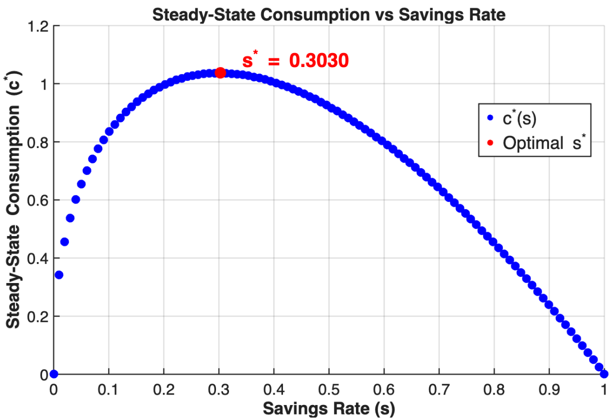


Figure: Maximizing c^* using grid search

- Grid search: over a pre-defined grid
- Golden search: 39 iterations, tolerance level

Golden: faster, control on tolerance, less issues on dimensionality

Writing high-performance code

High-Performance Code

Coding time + Fast execution time + Debugging time

Tricks to write a high-performance code

1. Vectorization
   $\rightarrow$ Matlab (just as Python / R) prefers vector/matrix operations than codes using loops

2. Write your own functions
   $\rightarrow$ Example: maximization problem with a solver vs golden algorithm

Remember: "Premature optimization is the root of all evil"... *(1) my code runs properly, (2) optimize if needed*

Optimize with vectorization

Example 1: Evaluate a function over a discrete interval:

```
a = linspace(0,10,1000);
f_a = zeros(1,10000);
tic
for i=1:size(a,1)
  f_a = exp(-a(i));
end
toc
tic
f_a_vec = exp(-a);
toc
```

Optimize with vectorization

Example 2: Element-by-element matrix operation:

```
A = rand(1000,1000);
B = zeros(1000,1000);
for i=1:size(A,1)
  for j=1:size(A,2)
     B(i,j) = A(i,j)*A(i,j);
  end
end
B_alt = A .* A;
```

Which method works best?

Exercice 1 - *do it at home!*

Ex 1: Solving the growth model with labor-saving technical progress and CES production function

Consider: $Y = [\alpha K^\rho + (1-\alpha)(AL)^\rho]^{1/\rho}$, with g the growth rate of A and ρ the elasticity parameter.

1. Define your parameters $\delta = 0.1$, $\alpha = 0.3$, $\rho = 2$, $g = 0.2$, $n = 0.1$, and $k_0 = 1$ using a structure
2. Write a function to solve the Solow model. It must have as
   Inputs: saving rate s, the parameters structure and the name of the algorithm to use (forward iteration, bisection or Newton)
   Outputs: the steady state capital stock per unit of effective labor $K/(AL) = k^*$,
3. Compute the golden rule saving rate using the Golden search method.
   Hint: your golden search function will operate on the function you wrote in the previous step.
4. Build two grids of 50 points each on $n \in [0.01, 0.1]$, $\delta \in [0.05, 0.4]$. Make a surface plot of the steady state k^* as a function of n and δ. Do it for each algorithm specified in question 2. Interpret the result.

   Hint: You can build a mesh grid using the function `meshgrid`.

Appendix: Generalities

1. Generalities on plotting
2. Generalities on functions
3. Generalities on conditional statements, loops and try-catch

Surface in 3D space

Objective: We want to plot z=f(x,y) for all possible (x,y)

- We need a value of z for each pair (x,y)
- x and y are vectors composed of the elements where the function is evaluated
- Z will be a matrix: for each given x, we need to compute z for all possible y; and for each given y we need to compute z for all possible x
- To get our matrix Z we need to transform X and Y into matrices

Surface in 3D space : transform vectors into matrices

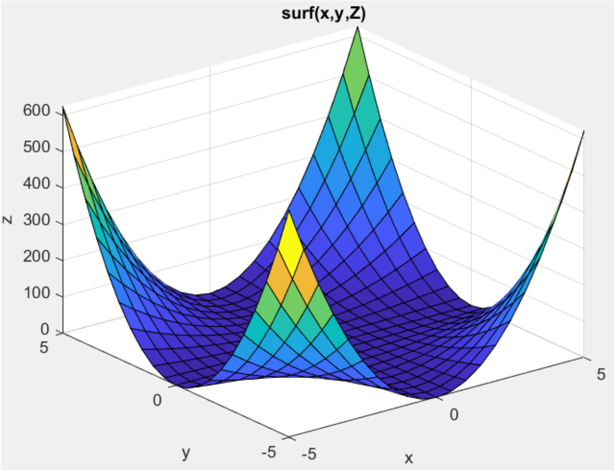
We want to transform x and y into matrices such that applying the transformation f(.,.) to X and Y yields Z

- The function [X,Y] = meshgrid(x,y) yields two matrices with the first having the rows filled of copies of the vector X and the second one having the columns filled of copies of the vector Y
- Now, applying the transformation f(.,.) to our X and Y will yield Z for all possible pairs (x,y)
- The function surf(x,y,Z) plots the values in matrix Z as heights above a grid in the x-y plane defined by X and Y

Surface in 3D space: example

```
x = -5:0.5:5;
y = -5:0.5:5;
[X,Y] = meshgrid(x,y);
Z = (X.*Y).^2;
figure(2)
surf(x,y,Z)
xlabel('x'); ylabel('y'); zlabel('Z');
title('surf(x,y,Z)')
```

Surface in 3D space: example



Discretization of an interval

- Equally spaced row vector from a to b with n elements:

```
e = linspace(a,b,n);
```

- Equally spaced row vector from a to b with an increment of x (stop before b if the increment does not fit):

```
f = a:x:b;
```

- Logarithmic spaced row vector from 10^a to 10^b with n elements:

```
g = logspace(a,b,n);
```

Functions : build-in and user-written

- Build-in functions are already available rand(.), diff(.) etc.
- Two types of user-written functions:
  1. Anonymous functions
  2. Functions (either saved in script or in a separate file)

Some useful build-in functions

- The function max(x) is one of the most useful function
- Extract the highest value in a vector and gives the index associated

```
xx = rand(1,5);
[max˙xx,i]=max(xx);
```

$\rightarrow$ Knowing the index gives the optimal policy function. More on that next class...

Some useful build-in functions

From matrix to vector to matrix:

```
% Define a matrix
A=[1,2,3;4,5,6]
% Vectorize it (column vector)
A_vec = A(:);
% Get back your original matrix
A_new = reshape(A_vec,2,3);
```

→ Useful to speed up codes to do operations on vectors than going for one cell at a time

Some useful build-in functions

- In simulations, it can be useful to always get the same sequence of random numbers
- In that case, you have to set a seed with any integer to the random number generator

```
rng(2);
x = rand(1,5)
```

Running the code above will always print the following vector:

```
x = [0.4360 0.0259 0.5497 0.4353 0.4204]
```

Some useful build-in functions

- The size(.) function returns a row vector whose elements are the lengths of the corresponding dimensions of A

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

$\rightarrow$ Returns a row vector $[2, 3]$

- size(.,x) returns a scalar of the length of the dimension x of our matrix

```
size(A,2)
```

$\rightarrow$ Returns 3

User-written functions

- You can write your own function as a script saved in a .m-file
- Your function must be saved in your current folder or in a folder that you have added to your search path if you want to use it
- The syntax must be the following:

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

(x1,...,xM) are the inputs to the function and (y1,...,yN) are the outputs that come out of it

User-written functions

PARIS SCHOOL OF ECONOMICS
ECOLE D'ECONOMIE DE PARIS

Build a function that takes a number and returns the square, the square root, and the factorial

```
function [a,b,c] = fun1(x)
  a = x^2;
  b = x^(1/2);
  c = prod(1:x);
end
```

To use it, write in a script or the command window:

```
fun1(x)
```

with x, any positive integer

Anonymous functions

- Anonymous functions are functions defined within a script (have a name but not their own .m file)
- Anonymous because they don't have their own .m-file but they do have a name

Example:

```
sqrt = @(x) x.^(1/2); sqrt(144)
```

Anonymous functions

Once a function is saved in the workspace, it can be easily plotted:

```
fun = @(x) 0.1*x.^2 + sin(x);
fplot(fun,[-5,5])
```

Loops and Conditional Statements

Conditional statement : if

Syntax example:

```
if x > 10
  % command block 1
elseif x > 5
  % command block 2
else
   %command block 3
```

1. If $x > 10$ then execute command 1
2. If not, then:
   2.1 If $x > 5$ then execute command 2
   2.2 If not then execute command 3

Loop for

- Runs the interior code a pre-specified number of times
- At each iteration the loop control variable is increased by one

Loop for : example 1

Generate 50 random number in uniform distribution over $[0, 1]$ and compute the average:

Loop for : example 1

Generate 50 random number in uniform distribution over $[0, 1]$ and compute the average:

```
a = rand(50,1);
mean_a = 0;
for i=1:size(a,1)
  mean_a = mean_a + a(i);
end
mean_a = mean_a/size(a,1);
```

Loop for : example 2

Compute the following sum:

$$\sum_{k=1}^{100} \sum_{i=1}^{k} i$$

Loop for : example 2

Compute the following sum:

$$\sum_{k=1}^{100} \sum_{i=1}^{k} i$$

```
x=0;
for k=1:100
  for i=1:k
    x=x+i;
  end
end
```

Loop for : example 3

Compute 100!

Loop for : example 3

Compute 100!

```
%Method1
x=1;
for i=1:100
    x=x*i;
end
%Method2
prod(1:100);
```

Loop while

- Runs the interior code as long as a condition is true. Exit the loop when it is false
- Ex-ante the number of iterations is unknown
  $\rightarrow$ Possible that it will keep running if the condition is always true
- Sometimes useful to include a maximum number of iterations

Loop while : example 1

Compute the limit of the following sequence: $u_{n+1} = -\frac{1}{2}u_n + 3$ with $u_0 = 5$

```
u = 5;
dif = 10;
i=0;
while dif > 1e-8
   u_prime = -0.5 * u + 3;
  dif = abs(u_prime - u);
  i = i + 1;
  u = u_prime;
end
```

If you prefer, you can use a maximum number of iterations + break