Advanced Macro Theory
ECON 5300, University of Oslo:

A Matlab Primer

Andreas Müller, Fall 2014

I. Why Matlab?

- Matlab is optimized for matrix computations that are often used in many fields of economics.
- Many other applied economists use Matlab to solve and simulate numerical models. You can learn and profit from other people's codes.
- Matlab is a high-level programming language that does the computational housekeeping for you.
- Matlab is a proprietary software and well documented. Advanced users can also use Octave the open source clone of Matlab.
- The University of Oslo has a campus license for Matlab that you can access through the UiO Program Kiosk

  http://www.uio.no/english/services/it/computer/software/servers

  But the best option is to get it installed on your own computer (consult the IT help desk!).

II. The Basics

A. *Some Basic Commands*

- `help`:
  Lists all toolboxes and subfolders that are on your search path. With the command `help cmd` you get access to the help file of the specific command `cmd`. A more user-friendly documentation can be accessed by typing `helpbrowser` in the command line.

- `edit`:
  Opens a new script file with .m extension in the text editor. Save all your code in scripts. Type `edit fun` to open the existing file fun.m in the text editor.

- `%`:
  It is very important to document your code. Put a `%` and Matlab will ignore the rest of the command line. This leaves space for comments and explanations.

B. *Some Matrix Commands*

- `A=[1,3;2,4];` initializes the 2 by 2 matrix

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}.$$

- Let's look at some matrix operators, +, -, *, .*,./, .^, where a dot indicates an element-wise operation. Take the previous matrix A, then

$$B \equiv A{*}A = \begin{bmatrix} 7 & 15 \\ 10 & 22 \end{bmatrix} \quad \text{and} \quad C \equiv A.{*}A = \begin{bmatrix} 1 & 9 \\ 4 & 16 \end{bmatrix}$$

- `D = A(1,:);` returns the first row of matrix A, $[1\ 3]$.
- `E = A(:,2);` returns the second column of matrix A, $[3\ 4]'$.
- `i = 2:2:10;` creates the row vector $i = [2\,4\,6\,8\,10]$.
- `F = max(A);` returns the max along the columns of A, $[2\ 4]$.
- `G = max(A,[],2);` returns the max along the rows of A, $[3\ 4]'$.

C. *Conditional Statements and Looping*

```
%% draws random grades for the macro students

% cell of strings with students' names
students = {'Frikk''s', 'Federica''s','Yudi''s','Ola''s'};
n = size(students,2);   % class size

for i=1:n % loop over n students
    student = students{i};       % reference ith student
    points  = randi(100,1,1);    % random integer between 1 and 100
    if points>50
        grade = 6;               % max grade in Switzerland
    else
        grade = 4;               % pass grade in Switzerland
    end
    % print grade for ith student
    fprintf('%-10s grade is a %i.\n',student,grade)
end
```

D. *Scripts and Functions*

```
%% allocate deterministic grades for the macro students

% cell of strings with student names
students = {'Frikk''s', 'Federica''s','Yudi''s','Ola''s'};
n = size(students,2); % class size

for i=1:n % loop over n students
    student = students{i};          % reference ith student
    grade   = get_grade(student);   % calls function get_grade()
    % print results
    fprintf('%-10s grade is a %8.6f.\n',student,grade)
end

% function get_grade() is located in a separate .m file in the
% same folder or on the search path.
```

```matlab
%% returns deterministic grades for the macro students

function grade = get_grade(student)

% search for student name that matches and allocate grade
switch student

    case 'Frikk''s'
      grade = 5.999999;
    case 'Federica''s'
      grade = 5.999998;
    case 'Yudi''s'
      grade = 5.999997;
    case 'Ola''s'
      grade = 5.999996;
    end

end
```

E. *Programming Style*

- Document your code, choose meaningful names for your variables, be transparent.
- "Get it run right, then get it run fast."
- Avoid loops whenever possible. In Matlab loops are generally slower than direct matrix operations. The code

  ```
  n = 10^6;
  for i=1:n, y(i)=log(i); end
  ```

  is on my PC around 7 times slower than

  ```
  y = log(1:1:10^6);
  ```

  The problem is squared if you work with nested loops. Use `tic` before and `toc` after a statement to measure running times.

F. *Rootfinding and Anonymous Functions*

- Consider the steady-state condition for the capital stock in the Solow Model

$$sf(k^\star) = \delta k^\star.$$

- Solving for $k^\star$ is the same as finding the root (the zero) of

$$h(k) \equiv sf(k) - \delta k.$$

- Solution Algorithm:

  1. Set parameters.
  2. Define anonymous functions (not located in a separate file).
  3. Use the Matlab built-in routine `fsolve` to find the root of $h(k)$.

```matlab
%% solves for the steady-state capital stock in the Solow Model

% parameters
alpha = .33;    % capital income share
delta = .10;    % depreciation rate
s     = .30;    % saving rate

% define anonymous functions
f = @(k) k.^alpha;
h = @(k) s*f(k)-delta*k;

% solve for the root of h(k)
k0 = 5;                                    % inital guess
options = optimset('Display','iter');  % display iterations
kstar = fsolve(h,k0,options);
```

G. *Further Literature*

- Miranda, Mario J., and Paul L. Fackler (2002). "Applied Computational Economics and Finance," MIT Press. Their book is accompanied by the CompEcon Toolbox for Matlab

  http://www4.ncsu.edu/∼ pfackler/compecon

- Adda, Jérôme, and Russell Cooper, "Dynamic Economics, " MIT Press.
- Sigmon, Kermit (1993). "Matlab Primer (Third Edition)," University of Florida.
- Official website, www.mathworks.com.
- Judd, Kenneth (1998). "Numerical Methodes in Economics," MIT Press.

III. An Example

A. *Neoclassical Growth Model*

- Bellman equation

$$V(k) = \max_{0 \le k' \le f(k) + (1-\delta)k} u(c(k,k')) + \beta V(k'), \quad c(k,k') = f(k) + (1-\delta)k - k'.$$

- Equilibrium conditions are a system of three functional equations in $V(k)$, $k' = g(k)$ and $c(k)$

$$\begin{aligned} V(k) &= u(c(k)) + \beta V(g(k)) \\ \frac{u_1(c(k)}{u_1(c(g(k)))} &= \beta[f_1(g(k)) + (1-\delta)] \\ c(k) &= f(k) + (1-\delta)k - g(k), \end{aligned}$$

where k is the state variable while $u(c)$ and $f(k)$ are known functions to be specified.

B. *Solution Algorithm*

1. Choose functional forms and parameters of preferences, $u(c)$, and technology, $f(k)$.

2. Compute the steady-state capital stock, $k^\star = g(k^\star)$.

3. Discretize the state space, $k \in (0, \infty)$, on the subgrid $[k_{min}, k_{max}]$ around the steady-state, $k^\star$.

4. Iterate on the value function,

$$V^{j+1}(k) = \max_{k' \in [k_{min}, f(k)+(1-\delta)k]} u(c(k,k')) + \beta V^j(k'),$$

   until convergence, starting with a guess $V^0(k')$.

5. Plot $V(k), g(k)$ and $c(k)$ over k.

```
%% 1.  functional forms and parameters

% parameters
alpha  = .36;     % capital income share
beta   = .95;     % subjective discount factor
delta  = .10;     % depreciation rate
sigma  = 2;       % relative risk aversion parameter

% functional forms
u  = @(c) c.^(1-sigma)/(1-sigma);     % preferences
f  = @(k) k.^alpha;                   % technology
f1 = @(k) alpha*k.^(alpha-1);         % rental rate
```

```
%% 2.  solve for steady-state capital stock

% use anonymous function
steady_state_condition = @(k) beta*(f1(k)+(1-delta))-1;

k0 = 4;                              % initial guess
options = optimset('Display','iter'); % display iterations
% use built-in rootfinding
kstar = fsolve(steady_state_condition,k0,options);
```

```
%% 3.  state space discretization

nk = 500;       % number of grid points

% set up grid around the steady state
range = .10;
kmin  = (1-range)*kstar;    % lower bound of the grid
kmax  = (1+range)*kstar;    % upper bound of the grid

% equally spaced grid
kgrid = linspace(kmin,kmax,nk)';

% transformation into matrices that can be used for the evaluation
% of functions.  kmat varies along columns, kpmat varies along rows.
[kmat,kpmat] = ndgrid(kgrid,kgrid);     % kp denotes k'
```

Let's look into the matrices:

- kgrid $= [k_1\, k_2 \ldots k_{n_k}]'$ is the discretized state k with $k_1 =$ kmin, $k_{n_k} =$ kmax and n_k grid points.

- kmat is the n_k by n_k matrix (grid points vary along columns, k)

$$\begin{bmatrix} k_1 & \cdots & k_1 \\ \vdots & \ddots & \vdots \\ k_{n_k} & \cdots & k_{n_k} \end{bmatrix}.$$

- kpmat is the n_k by n_k matrix (grid points vary along rows, k')

$$\begin{bmatrix} k_1 & \cdots & k_{n_k} \\ \vdots & \ddots & \vdots \\ k_1 & \cdots & k_{n_k} \end{bmatrix}.$$

- $c(\text{kmat}, \text{kpmat})$ then delivers the consumption level for any combination of k and k' on kgrid.

```
%% 4.  value function iteration

% initial guess:  one-period problem, thus kp=0.
V = u(f(kgrid)+(1-delta)*kgrid);

% continuation values and consumption possibilities using any
% combination of k and kp.
Vmat = u(f(kpmat)+(1-delta)*kpmat);
cmat = f(kmat)+(1-delta)*kmat-kpmat;

% momentary utility
% map negative consumption values into very low utility level.
umat = (cmat<=0).*(-10^6)+(cmat>0).*u(cmat);

% set convergence tolerance
tol   = 10^(-8);
maxit = 500;
fprintf('iter \t norm \n');
```

```
tic
for j=1:maxit
    % search for kp = argmax umat+beta*Vmat for each k
    % max is along dimension two (row)
    [Vnext,indices] = max(umat+beta*Vmat,[],2);
    error = norm(Vnext-V);
    fprintf('%4i %2.1e \n',j,error);
    % stopping rule
    if error < tol
        fprintf('Elapsed Time = %4.2f Seconds\n',toc);
        break;
    else
        V = Vnext;
        Vmat = repmat(V',nk,1);
    end
end

k = kgrid; kp = kgrid(indices); c = f(k)+(1-delta)*k-kp;
```

Let's look into the matrices (I chose indices arbitrarily!):

- Grid search for the maximum returns:

k	$V^{j+1}(k)$	indices^{j+1}	$k' = g^{j+1}(k)$	$V^{j+1}(k')$
k_1	$V^{j+1}(k_1)$	2	k_2	$V^{j+1}(k_2)$
k_2	$V^{j+1}(k_2)$	3	k_3	$V^{j+1}(k_3)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
k_{n_k}	$V^{j+1}(k_{n_k})$	$n_k - 1$	k_{n_k-1}	$V^{j+1}(k_{n_k-1})$

- Value function matrix for the next iteration only varies with k' but not with k.

$$\text{Vmat}^{j+1} = \begin{bmatrix} V^{j+1}(k_1) & \cdots & V^{j+1}(k_{n_k}) \\ \vdots & \ddots & \vdots \\ V^{j+1}(k_1) & \cdots & V^{j+1}(k_{n_k}) \end{bmatrix} = \begin{bmatrix} V^{j+1}(k)' \\ \vdots \\ V^{j+1}(k)' \end{bmatrix}.$$

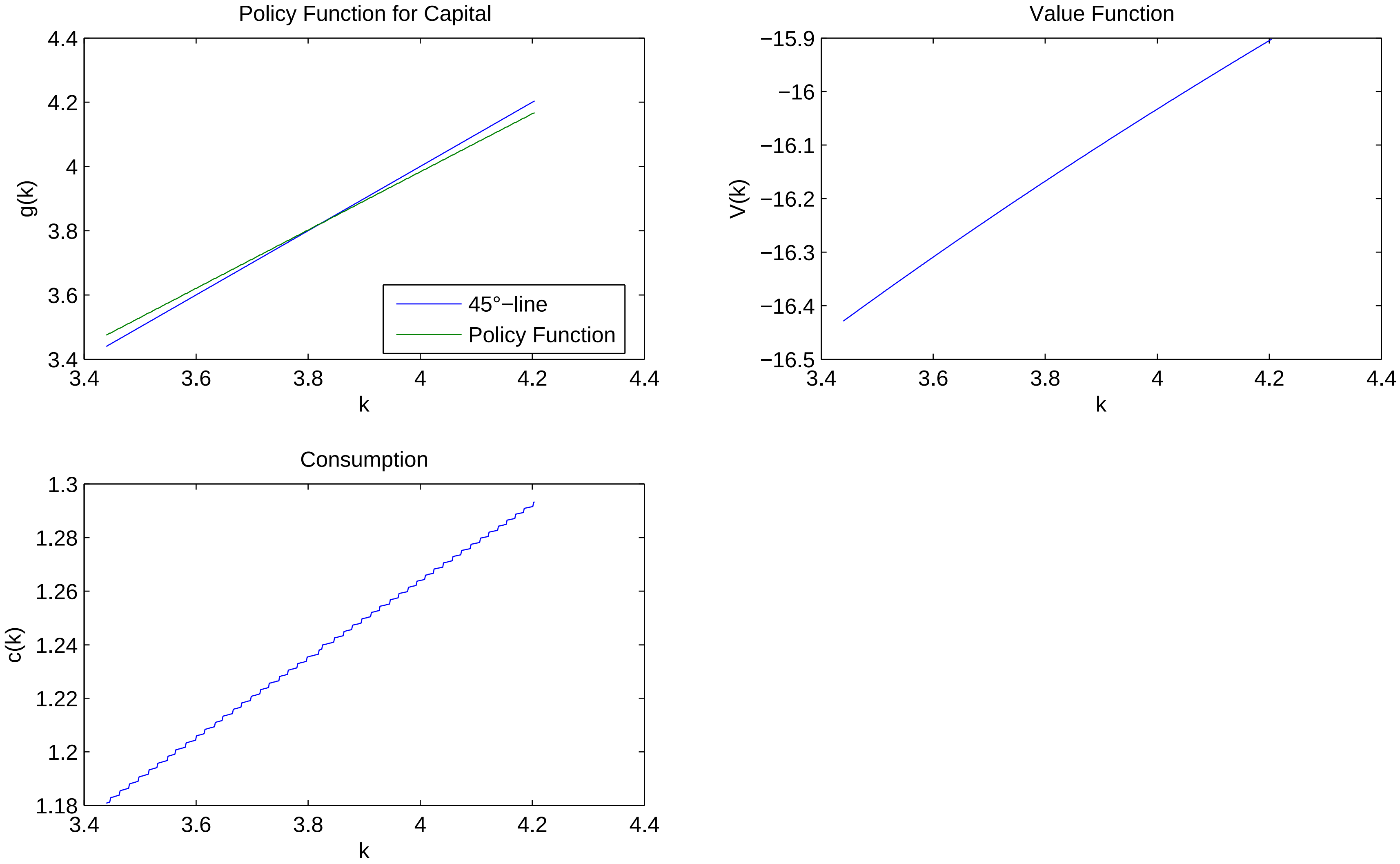
```matlab
%% 5.  plot unknown functions

scrsz = get(0,'ScreenSize');
figure('Position',[scrsz(3)*1/4 scrsz(4)*1/4 scrsz(3)*1/2 ...
     scrsz(4)*1/2]);

% policy function
subplot(2,2,1)
plot(k,[k,kp]);        % plot k and kp at the same time
title('Policy Function for Capital');
xlabel('k'); ylabel('g(k)');
legend('45°-line','Policy Function','Location','Best');

% value function
subplot(2,2,2)
plot(k,V);
title('Value Function');
xlabel('k'); ylabel('V(k)');
```

```
% consumption
subplot(2,2,3)
plot(k,c);
title('Consumption');
xlabel('k'); ylabel('c(k)');
```


Policy Function for Capital
g(k)
k
45°−line
Policy Function
4.4
4.2
4
3.8
3.6
3.4
3.4
3.6
3.8
4
4.2
4.4
Value Function
V(k)
k
−15.9
−16
−16.1
−16.2
−16.3
−16.4
−16.5
3.4
3.6
3.8
4
4.2
4.4
Consumption
c(k)
k
1.3
1.28
1.26
1.24
1.22
1.2
1.18
3.4
3.6
3.8
4
4.2
4.4

IV. Exercise 3.3: Howard's Policy Iteration

- The most time consuming part in the grid search algorithm of Section III is to find the policy function $g^j(k)$ for each state k in each iteration j.
- You can speed up the algorithm by iterating on the policy function $g^j(k)$ to update the value function $V^j(k)$ many times

$$V^{j,h+1}(k) = u(c^j(k)) + \beta V^{j,h}(g^j(k)), \quad c^j(k) = f(k) + (1-\delta)k - g^j(k),$$

before you continue to update the policy function.

- Exercise: implement Howard's policy iteration in the algorithm of Section III.

V. Summary

- Matlab provides powerful tools to solve dynamic economic models.
- Learn from other people's codes.
- Write code that other people can learn from.