Simulating the Classical Model with Capital using MATLAB®

(response to a temporary technology shock)

Contents

- Introduction
- Step 1 - Set Parameters of the Model
- Step 2 - Specify the System Matrices
- Step 3 - Apply Jordan Decomposition
- Step 4 - Check whether the Blanchard-Kahn Condition is satisfied
- Step 5 - Computing the Solution Time Paths for k_t and c_t
- Step 6 - Plots
- Limitations of the Jordan Decomposition

Introduction

The purpose of the following problem set is said to be a *step-by-step instruction* to write a MATLAB-file (`*.m`-file) containing the required commands to simulate the classical model with capital in response to a temporary technology shock ($\rho_a = 0.2$) numerically. It has a specific structure:

- Every section begins with more or less detailed instructions concerning the current task
- followed by a grey-shaded box/area which represents lines of missing code.
- Below you can find lines of the corresponding MATLAB-output[1] which would occur if you would have run the code above. Therefore, the output serves as a direct hint for the solution and should be used as guidance.

The general exercise is to replicate the given MATLAB-output using the instructions, MATLAB's `help`- and/or `doc`-command, the mathworks documentation center (http://www.mathworks.de/de/help/) and, of course, the *manuscript*.
So, please write an `*.m`-file containing the missing MATLAB-Code. You can either use the prepared `*.m`-file (from the folder "`X:\Kursmaterial`") or create a new one by opening a new MATLAB-script-file and save it perhaps as `rbc_model.m`.
Note that, except for parameters and exogenously given variables, it is not sufficient just to adopt the given lines of output since, in general, MATLAB-output shows evaluated numerical values, i.e. the result of a certain calculation which depends on these previously defined parameters/variables.

[1] MATLAB-output can be numerical values and also plots.

Step 1 - Set Parameters of the Model

At first set the parameters of the model according to the following parameterization (or calibration) of (1.2.93):

$$\alpha = 0.3, \quad \delta = 0.025, \quad \sigma = \eta = 1, \quad \tau = 0.7, \quad \beta = 0.99, \quad \rho_a = 0.2$$

```
1  alpha  = 0.3
2  delta  = 0.025
3  sigma  = 1
4  eta    = 1
5  tau    = 0.7
6  beta   = 0.99
7  AR_par = 0.2
```

```
alpha  = 0.3000
delta  = 0.0250
sigma  = 1
eta    = 1
tau    = 0.7000
beta   = 0.9900
AR_par = 0.2000
```

Step 2 - Specify the System Matrices B, C and $A = B^{-1}C$

Once one has derived the state equations (1.2.60) and (1.2.62) the resulting dynamic equation system has to be transformed into its matrix notation, i.e. its *(explicit) state space representation* of the form (1.2.77)

$$B \begin{pmatrix} E_t k_{t+1} \\ E_t c_{t+1} \end{pmatrix} = C \begin{pmatrix} k_t \\ c_t \end{pmatrix} + \begin{pmatrix} \tilde{d}_1 \\ \tilde{d}_2 \end{pmatrix} a_t$$

$$B^{-1} B \begin{pmatrix} E_t k_{t+1} \\ E_t c_{t+1} \end{pmatrix} = B^{-1} C \begin{pmatrix} k_t \\ c_t \end{pmatrix} + B^{-1} \begin{pmatrix} \tilde{d}_1 \\ \tilde{d}_2 \end{pmatrix} a_t$$

$$\begin{pmatrix} E_t k_{t+1} \\ E_t c_{t+1} \end{pmatrix} = A \begin{pmatrix} k_t \\ c_t \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} a_t \qquad \text{with } A = B^{-1}C \text{ and } \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = B^{-1} \begin{pmatrix} \tilde{d}_1 \\ \tilde{d}_2 \end{pmatrix}$$

Specify the system matrices B, C and A as well as $d_i = B^{-1} \begin{pmatrix} \tilde{d}_1 \\ \tilde{d}_2 \end{pmatrix}$:

```
8  % System is: B * [k; c](+1) = C * [k; c] + [d1; d2] * a
9   b11  = (1 - tau) * inv(delta) * (alpha + eta)
10  b12  = (alpha + eta) * tau + sigma * (1 - alpha)
11  b21  = (1 - tau) * inv(delta)
12  b22  = tau - sigma * inv(1 - beta + beta * delta)
13  B    = [ b11 b12 ; b21 b22 ]
14
15  c11  = alpha * (1 + eta) + (1 - tau) * (1 - delta) * inv(delta) * (alpha + eta)
16  c12  = 0
17  c21  = 1 + (1 - tau) * inv(delta) * (1 - delta)
18  c22  = -sigma * inv(1 - beta + beta * delta)
19  C    = [ c11 c12 ; c21 c22 ]
20
21  A        = inv(B) * C
22
23  d1_tilde = (1 + eta) * AR_par
24  d2_tilde = 0
25  di       = inv(B) * [d1_tilde ; d2_tilde]
```

```
b11  =   15.6000
b12  =    1.6100
b21  =   12.0000
b22  =  -28.0770
B    =   15.6000    1.6100
         12.0000  -28.0770

c11  =   15.8100
c12  =    0
c21  =   12.7000
c22  =  -28.7770
C    =   15.8100         0
         12.7000  -28.7770

A    =    1.0154   -0.1013
         -0.0184    0.9816

d1_tilde = 0.4000
d2_tilde = 0
di       = 0.0246
           0.0105
```

Step 3 - Application of the Jordan Decomposition

In order to solve the dynamic equations of the system in its explicit state space representation

$$\begin{pmatrix} E_t k_{t+1} \\ E_t c_{t+1} \end{pmatrix} = A \begin{pmatrix} k_t \\ c_t \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} a_t$$

the system matrix A must be decomposed to obtain its eigenvalues. Additionally the application of the Jordan decomposition transforms the equation system in such a way that two subsystems emerge which can be solved separately, i.e. the subsystem containing the predetermined (non-predetermined) variable with $|\lambda| < 1$ ($|\lambda| > 1$) can be solved backward (forward).

Usually one could just apply the already implemented MATLAB-command "`jordan(A)`" to the system matrix A. It would directly calculate the *diagonal matrix* of the eigenvalues Λ as well as the *modal matrix* H whose columns are the (already normalized) corresponding eigenvectors. Additionally, the command transforms the equation system in such a way that the upper subsystem contains the *stable* eigenvalue. Therefore, the following MATLAB-command

```
[H, Lambda] = jordan(A)
```

would produce the corresponding output

```
H = 1.6035   -3.4396
    1.0000    1.0000

Lambda = 0.9522         0
              0    1.0448
```

Unfortunately, due to the (limited) existing MATLAB-licences in Room 114, we lack a certain MATLAB-toolbox which provides this command. Therefore, you have to conduct the Jordan decomposition *manually*, i.e. you have

- to compute the eigenvalues and eigenvectors of A
- to normalize the eigenvectors
- to transform the calculated matrices Λ and H in such a way that the upper subsystem contains the stable eigenvalue (use the `fliplr`- and the `flipud`-command to flip the matrices A and H appropriately).

```
26  fprintf('Jordan decomposition for AR_par = %g:\n', AR_par);
27  [H, Lambda] = eig(A)
28  
29  % normalization of eigenvectors
30  H(:,1) = H(:,1)./H(2,1)
31  H(:,2) = H(:,2)./H(2,2)
32  
33  % flip upper/lower subsystem
34  H      = fliplr(H)
35  Lambda = fliplr(flipud(Lambda))
```

```
Jordan decomposition for AR_par = 0.2:

  H     =   0.9602    0.8485
           -0.2792    0.5292

  Lambda = 1.0448         0
                0    0.9522

  H     =  -3.4396    0.8485
            1.0000    0.5292

  H     =  -3.4396    1.6035
            1.0000    1.0000

  H     =   1.6035   -3.4396
            1.0000    1.0000

  Lambda = 0.9522         0
                0    1.0448
```

Task: Please verify that you have applied the Jordan decomposition above correctly, i.e. test[2] for $H\Lambda H^{-1} \neq A$ (use `help ne`). In the case of an existing solution, this should *not* hold. Hence, display an error message "Error in Jordan decomposition" (use `help error`) if it is true and a message "Successful application of Jordan decomposition" (use `help disp`) if it is not.[3] Use an `if`-condition (`help if`).

```
36  % Test
37  if H * Lambda * inv(H) ~= A
38      error('Error in Jordan decomposition')
39  else
40      disp('==> Successful application of Jordan decomposition')
41  end
```

```
  ==> Successful application of Jordan decomposition
```

Step 4 - Check whether the Blanchard-Kahn Condition is satisfied

Now it would be nice to see at a glance if the Blanchard-Kahn Condition is satisfied or not. Therefore, you should test whether the number of unstable eigenvalues equals the number of non-predetermined variables (use `help eq` and `help sum`) using an `if`-condition again.
First of all, define the eigenvalues on the principal diagonal of Λ (the eigenvalues of A) as a 2×1-matrix named "**eigenvalues**" (use `help diag`). Since we want information about the their stability, i.e. whether they are greater

[2]These small tests are very useful and are often used to avoid mistakes within the code and also for verification of the results, especially in larger programs.
[3]Of course, you are also allowed to conduct the test the other way round, i.e. test for $H\Lambda H^{-1} = A$ and display an error message if it is wrong.

or less than 1 in modulus, use only their *absolute values* (use `help abs`). Within the `if`-condition, check whether the computed eigenvalues are stable or unstable (use `help relop`). Since we want the condition to hold, a message "Blanchard-Kahn Condition is satisfied:" should be displayed if it is true. If it does not hold, display an appropriate error message "Blanchard-Kahn Condition is not satisfied:".

```
42    fprintf('Check Blanchard-Kahn Condition for AR_par = %0.2g:\n', AR_par)
43    eigenvalues = abs(diag(Lambda))
44    if sum(eigenvalues > 1) == 1
45        disp('==> Blanchard-Kahn Condition is satisfied:')
46    else
47        error('==> Blanchard-Kahn Condition is not satisfied:')
48    end
49    fprintf('Stable Eigenvalues:   %g\n', sum(eigenvalues < 1))
50    fprintf('Unstable Eigenvalues: %g\n', sum(eigenvalues > 1))
```

```
Check Blanchard-Kahn Condition for AR_par = 0.2:

eigenvalues = 0.9522
              1.0448

==> Blanchard-Kahn Condition is satisfied:
Stable Eigenvalues:   1
Unstable Eigenvalues: 1
```

Step 5 - Computing the solution Time Paths for k_t and c_t

Define the required expressions q, q_1, q_2 from (1.2.171), (1.2.179) and (1.2.180):

```
51  q  = H(1,1)/(Lambda(1,1) - AR_par) *  (di(1,:) - H(1,2) * di(2,:)) ...
52       + H(1,2)/(Lambda(2,2) - AR_par) * (-di(1,:) + H(1,1) * di(2,:))
53  q1 = 1/(Lambda(1,1) - AR_par) * (di(1,:) - H(1,2) * di(2,:))
54  q2 = 1/(Lambda(2,2) - AR_par) * (di(1,:) - H(1,1) * di(2,:))
```

```
q  =  0.1608
q1 =  0.0806
q2 =  0.0091
```

Define matrices which will save numerical solutions for $t = 0$ until $t = T$:

```
55  T = 61
56  k_solution = NaN(1,T)      % Defines k_solution as a 1-by-T vector without content
57  c_solution = NaN(1,T)      % Defines c_solution as a 1-by-T vector without content
```

The `NaN`-expression means "Not-a-Number" and serves as a placeholder for values to be calculated in the next steps (use `help NaN`). These vectors now look like

$$k_{\text{solution}} = \begin{pmatrix} \text{NaN}_1 & \dots & \text{NaN}_T \end{pmatrix} = \begin{pmatrix} k_0 & k_1 & \dots & k_{T-1} \end{pmatrix}$$
$$c_{\text{solution}} = \begin{pmatrix} \text{NaN}_1 & \dots & \text{NaN}_T \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & \dots & c_{T-1} \end{pmatrix}$$

Since we want to fill the above defined solution matrices with numerical values, we set the solution time paths $E_0 k_t$ and $E_0 c_t$ according to (1.2.172) and (1.2.184), i.e. depending on the time t. Additionally, we create a **for**-loop (use `help for`) which repeats a certain calculation for a specific number of times (here: from $t-1 = 0$ until $T-1 = 60$). Additionally, it replaces the NaN-placeholders in both matrices with the computed numerical solutions of the solution time paths.

```
58  for t = 1:T
59      k_solution(:,t) = 1/(H(1,1) - H(1,2)) * q * (Lambda(1,1)^(t-1) - AR_par^(t-1))
60      c_solution(:,t) = (1/(H(1,1)-H(1,2))) * (Lambda(1,1)/H(1,1) * (H(1,1)*q1 - H(1,2)*q2) ...
61                        * Lambda(1,1)^(t-2) - AR_par * (q1 - q2) * AR_par^(t-2))
62  end
```

```
k_solution = 0          0.0240    0.0276  ...  0.0019    0.0018    0.0017
c_solution = 0.0057     0.0161    0.0175  ...  0.0012    0.0011    0.0011
```

Note that since the variable of the loop-statement t denotes the column of the solution matrices as well as the time index in the formulas for the solution time paths, one has to be very careful not to unintentionally omit the value for $t = 0$. Why is that the case? (Hint: $NaN_t = k_{t-1}$) What can be done to avoid this difficulty?

Step 6 - Plots

The solution time paths are plotted for the first 60 periods:

```
63    t = 0:60;
```

Now we just have to specify how the plots have to look like (see: `help subplot` and `help plot`)

```
64    figure
65
66    subplot(1,2,1); hold on;
67    plot(t, k_solution(1,t+1), 'k-o', 'LineWidth',1)
68    xlabel('t'); ylabel('k'); title('capital stock')
69    hline = refline([0 0]);
70    set(hline,'Color','r')
71    box on
72
73    subplot(1,2,2); hold on;
74    plot(t, c_solution(1,t+1), 'k-o', 'LineWidth',1)
75    xlabel('t'); ylabel('c'); title('consumption')
76    hline = refline([0 0]);
77    set(hline,'Color','r')
78    box on
```

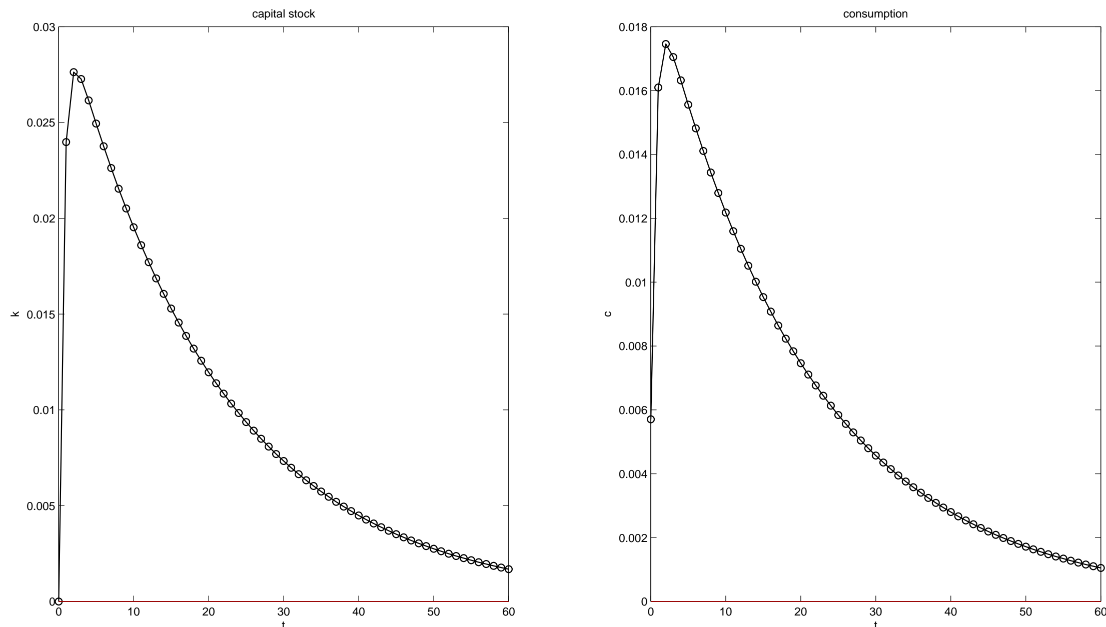

Figure 1: Adjustment time paths in response to a temporary technology shock with ($\rho_a = 0.2$).

Limitations of the Jordan Decomposition

Consider a dynamic equation system of the form

$$\begin{pmatrix} w_{t+1} \\ E_t v_{t+1} \end{pmatrix} = A \begin{pmatrix} w_t \\ v_t \end{pmatrix}.$$

In order to solve the dynamic equations, one is in need of the eigenvalues of the system matrix A. They can be obtained by the usage of the Jordan decomposition which decomposes A into the expression

$$H \Lambda H^{-1}.$$

This similarity transformation still contains the same information as A but now the diagonal matrix Λ reveals the wanted eigenvalues of the system matrix. The decomposition method solves the standard eigenvalue problem

$$\begin{aligned} Av &= \lambda v \\ (A - \lambda I)v &= 0 \end{aligned}$$

Unfortunately, there are several (theoretical and numerical) difficulties with the matrix decomposition of this form, in particular that *a solution does not always exist*. A brief examination of this difficulties will serve to highlight the limitations of the Jordan decomposition.

Theoretical Limitation

Since a solution of the Jordan decomposition only exists if H is non-singular (i.e. H^{-1} does exist), it is required that an $n \times n$ (square) matrix A has a full set of n linearly *in*dependent eigenvectors. Thus, the matrix H whose columns comprises the eigenvectors of A must have a full set of n linearly independent columns or put differently

$$H \text{ is non-singular} \quad \Longleftrightarrow \quad H^{-1} \text{ does exist} \quad \Longleftrightarrow \quad \text{rank}(H_{n\times n}) = n$$

(note that the rank of a matrix is equal to the amount of its linear independent vectors). Matrices which satisfy this condition are called *diagonalizable*. Matrices are typically diagonalizable if they have n *distinct* eigenvalues, i.e. all eigenvalues have a maximum multiplicity of 1. In the rare cases in which a matrix has *multiple* eigenvalues, the condition is not fulfilled and no solution of the Jordan decomposition exists since H is not invertible. Such non-diagonalizable matrices are called *defective*.

Example: Consider the following matrix

$$A = \begin{pmatrix} 0 & 4 & -2 \\ -1 & -4 & 1 \\ 0 & 0 & -2 \end{pmatrix}.$$

Show that A is defective! For this purpose compute the matrices Λ and H as well as the rank of H.

```
79  A = [0 4 -2; -1 -4 1; 0 0 -2];
80  [H,Lambda] = eig(A)
81  rankH = rank(H)
```

```
H =
    0.8944   -0.8944    0.8944
   -0.4472    0.4472   -0.4472
         0         0    0.0000

Lambda =
    -2     0     0
     0    -2     0
     0     0    -2

rankH = 1
```

The result shows that all columns of H are the same (except for the signs), i.e. aren't linearly independent. As a consequence, there exists only one distinct eigenvalue with multiplicity of 3. This fact can also be observed by looking at the principal diagonal of Λ. Lets see what happens if you now try to compute the inverse of H although you have already shown that A is defective:

```
82  invH = inv(H)
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate.
RCOND =  8.275114e-17.

invH =
   1.0e+15 *

    2.2518    4.5036   -6.4467
    2.2518    4.5036    2.1489
         0         0    8.5956
```

As you can see, MATLAB prints a warning that H is close to singular in this case. It means that the fundamental requirement of a full set of distinct eigenvectors of H for the existence of a solution of the Jordan Decomposition is not fulfilled.

Numerical Limitation

Decomposing the defective matrix A using the built-in `jordan`-function

```
83  disp('Jordan Decomposition')
84  [H, Lambda] = jordan(A)
85  rankH = rank(H)
86  invH  = inv(H)
```

would have led to the following output:

```
H =
     2     1     1
    -1     0     0
     0     0     1

Lambda =
    -2     1     0
     0    -2     0
     0     0    -2

rankH = 3

invH =
     0    -1     0
     1     2    -1
     0     0     1
```

Compare both results for the computed matrices H and Λ. The outcomes seem to depend on the used computational method. Additionally, What do you observe concerning the rank of H and its inverse H^{-1}?

Now the columns of H consists of ordinary eigenvectors and *generalized* eigenvectors. H is augmented with these always linear independent vectors satisfying

$$(A - \lambda I)^k v = 0 \quad \text{with } k > 1 \qquad \text{(multiplicity of EV)}$$

in order to achieve a full rank of H and, therefore, ensuring the existence of H^{-1}. The implementation of generalized eigenvectors serves to avoid the theoretical limitation of the non-existence of a solution but causes a numerical limitation. The numerical difficulties arise due to the fact that the obtained *Jordan canonical form* Λ now has ones

above its principal diagonal in positions corresponding to the columns of H which represent generalized eigenvectors, i.e. Λ is not a diagonal matrix anymore.

$$\Lambda_{\text{ordinary}} = \begin{pmatrix} \lambda_i & \textcircled{0} & 0 \\ 0 & \lambda_i & 0 \\ 0 & 0 & \lambda_i \end{pmatrix} \Longrightarrow \Lambda_{\text{generalized}} = \begin{pmatrix} \lambda_i & \textcircled{1} & 0 \\ 0 & \lambda_i & 0 \\ 0 & 0 & \lambda_i \end{pmatrix}$$

Λ is a *discontinuous* function of the matrix A and, therefore, almost any perturbation of a defective matrix can cause a multiple eigenvalue to separate into distinct values which would lead to the elimination of the ones above the diagonal. Thus, a (even small) change in the parameterization typically leads to a different system matrix A. For instance, see what happens if we would change the parameters in such a way that only element a_{32} flips from 0 to 1, i.e. matrix A just changes slightly to

$$A = \begin{pmatrix} 0 & 4 & -2 \\ -1 & -4 & 1 \\ 0 & 1 & -2 \end{pmatrix}.$$

```
87  A = [0 4 -2; -1 -4 1; 0 1 -2];
88  [H,Lambda] = eig(A)
89  rankH = rank(H)
```

```
H =
   -0.8165    0.8165   -0.7071
    0.4082   -0.4082    0.0000
   -0.4082   -0.4082   -0.7071

Lambda =
   -3.0000         0         0
         0   -1.0000         0
         0         0   -2.0000

rankH = 3
```

Now, due to this small change, we again have a set of linearly independent eigenvectors (full rank of H) and, thus, completely distinct set of eigenvalues. Moreover, this example shows that matrices which are nearly defective have badly conditioned sets of eigenvectors, and even if the inverse of H exists, the resulting similarity transformations cannot be used for reliable numerical computation.

A numerically satisfactory alternative to the Jordan decomposition is provided by the *Schur decomposition*. This decomposition method is able to transform *any* matrix into its upper triangular form by a unitary similarity transformation

$$A = Z \cdot T \cdot \overline{Z}'$$

The unitary matrix Z with

$$Z \cdot \overline{Z}' = \overline{Z}' \cdot Z = I_{n\times n}$$

provides a basis with much better numerical properties than a set of eigenvectors like the matrix H of the Jordan decomposition because a unitary matrix is always diagonalizable, i.e. is never defective!

Decomposition Method	Form of the System	Decomposition	Corresponding EV Problem	Preconditions
Jordan	$\begin{pmatrix} w_{t+1} \\ E_t v_{t+1} \end{pmatrix} = C \begin{pmatrix} w_t \\ v_t \end{pmatrix}$ with $C = A^{-1}B$	$C = H \Lambda H^{-1}$	$Cv = \lambda v$ (standard)	• A has to be non-singular • H has to be non-singular
Schur	$\begin{pmatrix} w_{t+1} \\ E_t v_{t+1} \end{pmatrix} = C \begin{pmatrix} w_t \\ v_t \end{pmatrix}$ with $C = A^{-1}B$	$C = ZT\overline{Z}'$	??? ???	???
QZ Factorization (Generalized Schur)	???	??? ???	???	???

Table 1: Matrix Decomposition Methods in Macroeconomic Dynamics: An Overview