DIY Macroeconomic Model Simulation

Table of contents

Welcome

> ⚠ Warning
>
> This website is under construction and will be regularly updated and extended.

This platform provides an open source code repository and online script for macroeconomic model simulation. It follows a "do-it-yourself" (DIY) approach, empowering users to numerically simulate key macroeconomic models on their own using the open-source programming languages *R* and *Python*. Whether you are a university teacher, student, researcher, or an economics enthusiast, our platform offers resources to deepen your understanding of both macroeconomic modelling and coding.

The platform covers an array of macroeconomic models, including canonical textbook models, models from different economic paradigms, and seminal models from the history of economic thought. It bridges a gap between intermediate and advanced level macroeconomics by providing detailed yet accessible treatments of seminal macroeconomic models. Most intermediate macroeconomics textbooks focus on graphical analysis, while advanced level materials are often more mathematical and less accessible. Our platform offers a hands-on and approachable resource for users to build both a solid foundation in modelling and macroeconomic intuition.

The platform's DIY-approach aims to foster reproducibility and open-source principles in macroeconomic research and education by providing learning materials that are freely available and modifiable by everyone. The platform's content will expand over time through new entries added by the project team.

Project team

Contact

We welcome any feedback. If you encounter any issues, find typos or mistakes, or have questions/thoughts on the content, please do get in touch.

1. If you are Github user, you can report an issue in our repository
2. You can also email us directly: `franz.prante@wiwi.tu.chemnitz.de` and `k.kohler@leeds.ac.uk`

(a) Franz Prante

(b) Karsten Kohler

Franz is a research associate at Chemnitz University of Technology, where he is currently working on the macroeconomic effects of monetary policy and price effects on energy demand. He is also a PhD student at Université Sorbonne Paris Nord.

Karsten is an Associate Professor in Economics at Leeds University Business School, where he does research on the interaction between finance and the real economy, especially sources of cyclical dynamics, instability, and rising inequality.

License

Acknowledgements

We are grateful for helpful comments from Adam Aboobaker, Chandni Dwarkasing, Giuseppe Fontana, Alex Guschanski, Eckhard Hein, Grégoire Noël, and Rafael Wildauer on various sections of this website. All errors are ours.

1 Getting Started

1.1 Structure of platform

The platform starts off with a general introduction the numerical simulation of economic models (Chapter **??**). After that, it jumps right into a series of macroeconomic models. These models are grouped into static and dynamic models. In static models, time plays no role and all variables adjust instantaneously. By contrast, dynamic models characterise the adjustment of variables over time.

The model entries are largely self-contained and can be read independently of each other. For each model, the chapters provide three main components:

1. **Model descriptions** that concisely explain the key ideas, assumptions, and equations of each model. This helps users grasp the underlying concepts and intuition behind the models.
2. **Annotated code** that allows users to numerically simulate the models, examine their results under different scenarios, and produce visualisations to better understand the models' structure and output. This hands-on approach enables users to gain practical coding skills while exploring different macroeconomic theories.
3. **Analytical discussions** for users who are interested in delving deeper into the mathematical properties of the models.

To further facilitate the understanding of dynamic models, Chapter **??** of the section on dynamic models begins with a general introduction into the mathematical analysis of dynamic models (this is mostly relevant for the analytical discussions of dynamic models).

All simulation codes are written in the open-source programming languages *R* and *Python*. *R* codes are presented in the main text (and images and results in the text are the output of the shown *R* codes), while the corresponding *Python* codes are available by clicking on the callout blocks underneath the *R* codes.

1.2 Access and introduction to *R* and *Python*

To be able to manipulate the codes on this platform on your own machine, you first need to download and install *R* and *RStudio*. For Python, there are different options. One of them

is to download and install *Spyder* via the Anaconda Python distribution. *Spyder* provides an interface for *Python* (like *RStudio* for *R*). If you install it via *Anaconda*, it will install *Python* automatically.

Besides being free, a key advantage of both *R* and *Python* is their huge and growing functionality due to new user-written libraries and packages continuously being added. In addition, a large amount of learning material is freely available on the web, e.g. here and here for *R* and here, and here for *Python*. However, to get started in can be best to directly delve into some of the codes on this platform and learn by doing. To this end, the following "cheatsheets" that provide a concise overview of key functions are useful:

- R Studio Cheatsheet
- Base R Cheatsheet
- more R cheatsheets here
- Python Basics Cheatsheet
- Python for Data Science Cheatsheet
- Python Cheatsheet for NumPy library

Once you have installed *R* or *Python*, you can play with the codes on this platform yourself by copy and pasting them into the script panel of your local interface (IDE) and hitting CTRL + Enter to execute them. Don't forget to always comment your code using the "#" symbol and to save your scripts to make sure your future self can seamlessness continue working on it.

The codes below covers some basic operations.[1] If you are new to *R* or *Python* and keen to get started, do the following:

- copy the codes below into the script panel of *RStudio* or *Spyder* on your machine
- adjust the working directory to your personal folder
- then run the code
- make sure you understand what it does

```
################################### R Basics ############################

#In the R script you write code and comments
  #any line starting with # is a comment and it is NOT executed

## First things first: set the working directory to the the folder in which your R files are
# note that you need to separate folders by slashes /
```

[1]We are grateful to Rafael Wildauer for permission to reproduce a slightly modified version of his learning materials. In addition, some of the material below is taken from here.

```r
#Let's define some variables
s = 0.05
Y = 10

#We can also assign several variables the same value at once
C = I = R = 2

#For displaying them we simply call the name of the object and execute the relevant line
Y
C

#We can define new variables using existing ones
W = Y - R
W

#R has a vast amount of built in functions, for example
max(10,2,100,-3)
sqrt(9)
abs(-13)

#you can find out more about these by using the help function, e.g.:?max()

#How can you delete stuff? Use the remove function rm()
#for individual objects
rm(Y)
#if you want to remove everything
rm(list=ls(all=TRUE))

#You can also assign text (a string) to a variable
text1 = 'Reggaeton'
text2 = "Bad Bunny"
#note "text" is the same as 'text' and a string can contain spaces
#You use the paste function to combine strings
paste(text1, text2)

#R uses standard operators like +, -, *
#for exponents use ^
3^2


############################ if statements and loops ######################
```

```r
#sometimes we need to introduce if conditions into our code
#The syntax is
#if condition { do something }
a = 10
b = 13
if (b>a) {
  print("b is bigger")
}
#we can also tell R what to do in case the condition is not fulfilled
b=10
if (b>a) {
  print("b is bigger")
} else {
  print("b not bigger")
}


#Next we will look at loops which are a key tool to repeat tasks such as solving
#a theoretical model again and again to find its equilibrium.
#The basic structure is:
for (i in 1:5){
  print(i)
}

#Let's use it to solve a simple Keynesian cross model of the form
#Y=C+I
#C=c0+c1Y
#Define exogenous parameters
c0=2
c1=0.8
I=10
#set initial values for two endogenous variable
Y=C=1
#Use a for loop to solve it
C
Y
for (i in 1:100){
  Y = C+I
  C = c0+c1*Y
}
C
Y
```

```r
#solution is Y=(I+c0)/(1-c1)=60

#What is special about this loop is that it uses the values from the previous iteration
#to define the values of the next, because it starts with assigning a value to Y
#and then uses that value to assign a new value to C and in the next iteration it
#uses this new value for C to define a new value for Y etc.


############################ Data structures ############################
#In most applications we produce outputs which do not consist of a single number.
#Often we have an entire stream of results, or we want to analyse data and have to store larg
#R has a variety of data structures for this purpose.
#let's clean up first
rm(list=ls(all=TRUE))

####Vectors
#We can create an empty vector and fill it later (with results of our model for example)
vec1 = vector(length=3)
#we can define vectors explicitly using the c() function (c for column?)
vec2 = c(1,2,3)
vec3 = c(6,7,8)
#we can also use the sequence operator
vec4 = 1:10
#and we can define the step size
vec5 = seq(1,2,0.1)
vec5
#we can call specific entries using square brackets
vec5[4]
#if we want to access more elements at once
vec5[c(4,1)]

####Matrices
#define a matrix: 3 rows and 2 columns, all elements equal to 0
mat1 = matrix(0, nrow=3, ncol=2)
mat1
#we can also fill it with specific values
mat2 = matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)
mat2
#access specific elements (columns,rows)
mat2[3,1]
#access entire rows or columns
mat2[,1]
```

```r
mat2[1,]
#access sub matrices
mat2[c(1,2),]
## Combine two column vectors in a matrix
mat3=cbind(vec1, vec2)
mat3
## Combine two row vectors in a matrix
mat4=rbind(vec1, vec2)
mat4
```

Python code

1.3 Simple exercise

1.

Solution

```
 [1] 1.000000 1.250000 1.361111 1.423611 1.463611 1.491389 1.511797 1.527422
 [9] 1.539768 1.549768
```

2.

Solution

```
              x        y
 [1,] 1.000000 1.000000
 [2,] 1.250000 1.125000
 [3,] 1.361111 1.162037
 [4,] 1.423611 1.177662
 [5,] 1.463611 1.185662
 [6,] 1.491389 1.190292
 [7,] 1.511797 1.193207
 [8,] 1.527422 1.195160
 [9,] 1.539768 1.196532
[10,] 1.549768 1.197532
```

2 How to Simulate Economic Models

2.1 Introduction: economic models

1.
2.
3.

-
-
-
-

-
-

[1]

-
-
-

[1]That raises the question of whether an equilibrium is stable or unstable, which is discussed in Chapter **??**.

2.2 Solving economic models numerically

the Gauss-Seidel method

-
-
-

2.2.1 Solving economic models numerically: examples

2.2.1.1 A static model

```
### Simulate Keynesian goods market model via iteration

#Clear the environment
rm(list=ls(all=TRUE))

# Set number of parameterisations that will be considered
S=2

# Set fixed parameter values
c0=3
c1=0.8

#Create vector in which equilibrium solutions from different parameterisations will be stored
Y_eq=vector(length=S)
C_eq=vector(length=S)

#Create vector with parameter that will change
I0=vector(length=S)
I0[1]=5
I0[2]=6

# Initialise endogenous variables at arbitrary positive value
Y=C=1

#Solve this system numerically through 1000 iterations based on the initialisation
for (i in 1:S){
  for (iteration in 1:1000){
  Y = C + I0[i]
  C = c0 + c1*Y
  } # close iterations loop

#Save results for different parameterisations in vector
```

```
Y_eq[i]=Y
C_eq[i]=C
} # close parameterisations loop

# Display solutions
Y_eq
```

```
C_eq
```

```
# Verify solutions for Y
(c0+I0[])/(1-c1)
```

Python code

```
### Simulate Keynesian goods market model via iteration

# Load NumPy library
import numpy as np

# Set the number of parameterisations that will be considered
S = 2

# Set fixed parameter values
c0 = 3
c1 = 0.8

# Create numpy arrays in which equilibrium solutions from different parameterisations will
Y_eq = np.zeros(S)
C_eq = np.zeros(S)

# Create a numpy array with the parameter that will change
I0 = np.zeros(S)
I0[0] = 5
I0[1] = 6

# Initialize endogenous variables at an arbitrary positive value
Y = C = 1

# Solve this system numerically through 1000 iterations based on the initialization
for i in range(S):
    for iteration in range(1000):
        Y = C + I0[i]
        C = c0 + c1 * Y

    # Save results for different parameterisations in the numpy arrays
    Y_eq[i] = Y
    C_eq[i] = C

# Display solutions
Y_eq
C_eq

# Verify solutions for Y
(c0+I0)/(1-c1)
```

-
-
-
-
-
  -
-
-

```
### Parameterisation for which method of iteration fails

#Clear the environment
rm(list=ls(all=TRUE))

# Set number of parameterisations that will be considered
S=2

# Set fixed parameter values
c0=3
c1=1.2

#Create vector in which equilibrium solutions from different parameterisations will be stored
```

```
Y_eq=vector(length=S)
C_eq=vector(length=S)

#Create vector with parameter that will change
I0=vector(length=S)
I0[1]=5
I0[2]=6

# Initialise endogenous variables at arbitrary positive value
Y=C=1

#Solve this system numerically through 1000 iterations based on the initialisation
for (i in 1:S){
  for (iteration in 1:1000){
  Y = C + I0[i]
  C = c0 + c1*Y
  } # close iterations loop

#Save results for different parameterisations in vector
Y_eq[i]=Y
C_eq[i]=C
} # close parameterisations loop

# Display solutions
Y_eq
```

```
# Verify solutions for Y
(c0+I0[])/(1-c1)
```

Python code

```
### Parameterisation for which method of iteration fails
c1 = 1.2

# Initialize endogenous variables at an arbitrary positive value
Y = C = 1

# Solve this system numerically through 1000 iterations based on the initialization
for i in range(S):
    for iteration in range(1000):
        Y = C + I0[i]
        C = c0 + c1 * Y

    # Save results for different parameterisations in the numpy arrays
    Y_eq[i] = Y
    C_eq[i] = C

# Display solutions for Y_eq
Y_eq

# Verify solutions for Y
(c0+I0)/(1-c1)
```

2.2.1.2 A dynamic model (in discrete time)

```
### Simulate Samuelson 1939

#Clear the environment
rm(list=ls(all=TRUE))

# Set number of periods for which you want to simulate
Q=100

# Set number of parameterisations that will be considered
S=2

# Set period in which shock or shift in an will occur
s=15

# Set fixed parameter values
c1=0.8
beta=0.6

# Construct (S x Q) matrices in which values for different periods will be stored; initialise
C=matrix(data=1, nrow=S, ncol=Q)
I=matrix(data=1, nrow=S, ncol=Q)

#Construct matrices for exogenous variable or parameter that will change over time to capture
G0=matrix(data=5, nrow=S, ncol=Q)

# Set parameter values for different scenarios
G0[2,s:Q]=6     # scenario: permanent increase in government spending from I0=5 to I0=6 from

#Solve this system recursively based on the initialisation
for (i in 1:S){
  for (t in 2:Q){
    C[i,t] = c1*(C[i,t-1] + I[i,t-1] + G0[i,t])
```

```
    I[i,t] = beta*(c1*(C[i,t-1] + I[i,t-1] + G0[i,t]) - C[i,t-1])
  } # close time loop
}   # close scenarios loop

# Calculate output
Y=C+G0+I

# Display solution
Y[,Q]
```

```
# Verify solutions for Y
(G0[,Q])/(1-c1)
```

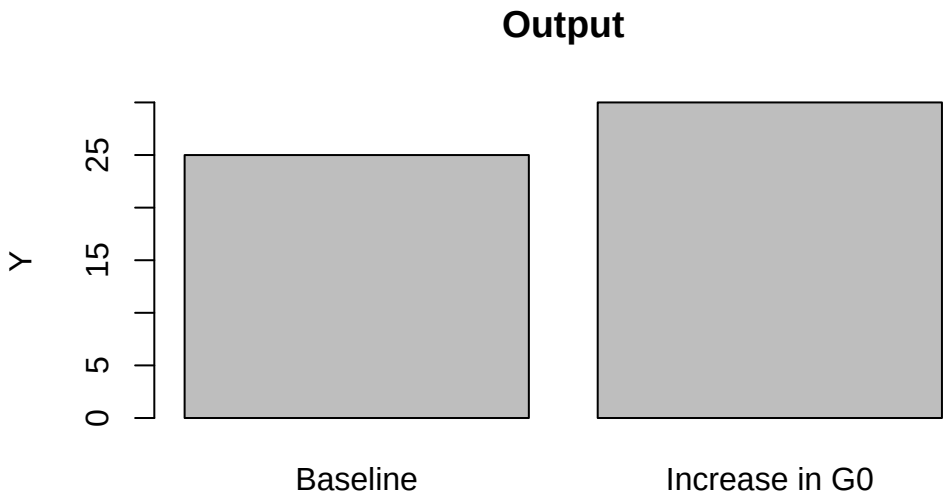
Python code

```
### Simulate Samuelson 1939

# Set the number of periods for which you want to simulate
Q = 100

# Set the number of parameterisations that will be considered
S = 2

# Set the period in which a shock or shift in 'an' will occur
s = 15

# Set fixed parameter values
c1 = 0.8
beta = 0.6

# Construct (S x Q) matrices in which values for different periods will be stored; initiali
C = np.ones((S, Q))
I = np.ones((S, Q))

# Construct matrices for exogenous variables or parameters that will change
# over time to capture different scenarios, initialise at 5
G0 = np.ones((S, Q))*5

# Set parameter values for different scenarios
G0[1, s:Q] = 6  # scenario: permanent increase in government spending from I0=5 to I0=6 fro

# Solve this system recursively based on the initialization
for i in range(S):
    for t in range(1, Q):
        C[i, t] = c1 * (C[i, t - 1] + I[i, t - 1] + G0[i, t])
        I[i, t] = beta * (c1 * (C[i, t - 1] + I[i, t - 1] + G0[i, t]) -
                          C[i, t - 1])

# Calculate output
Y = C + G0 + I

# Display the solutions at time Q
Y[:, Q - 1]

# Verify solutions for Y
(G0[:,Q - 1])/(1-c1)
```

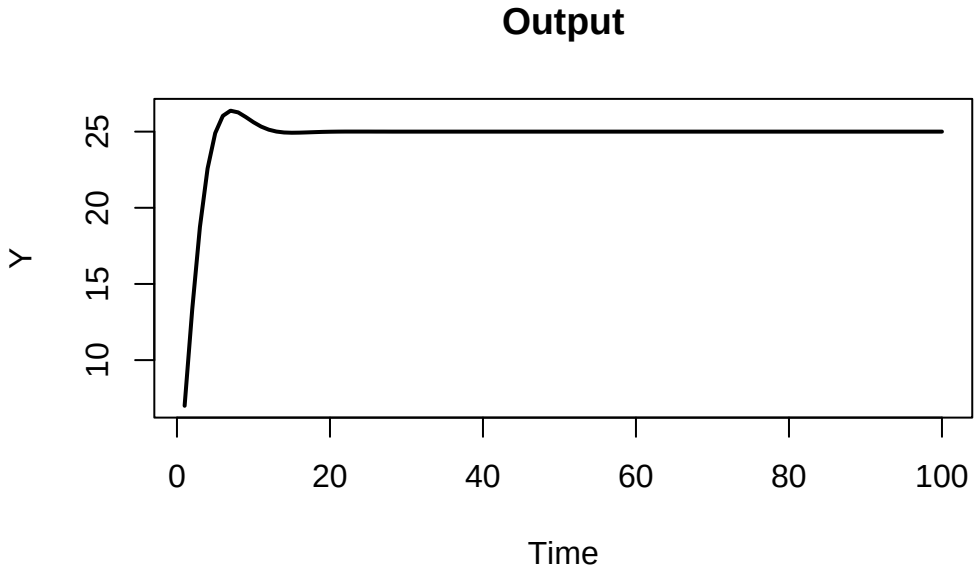
-
-
-
-
-
  –
-

2.3 How to plot the results of a model

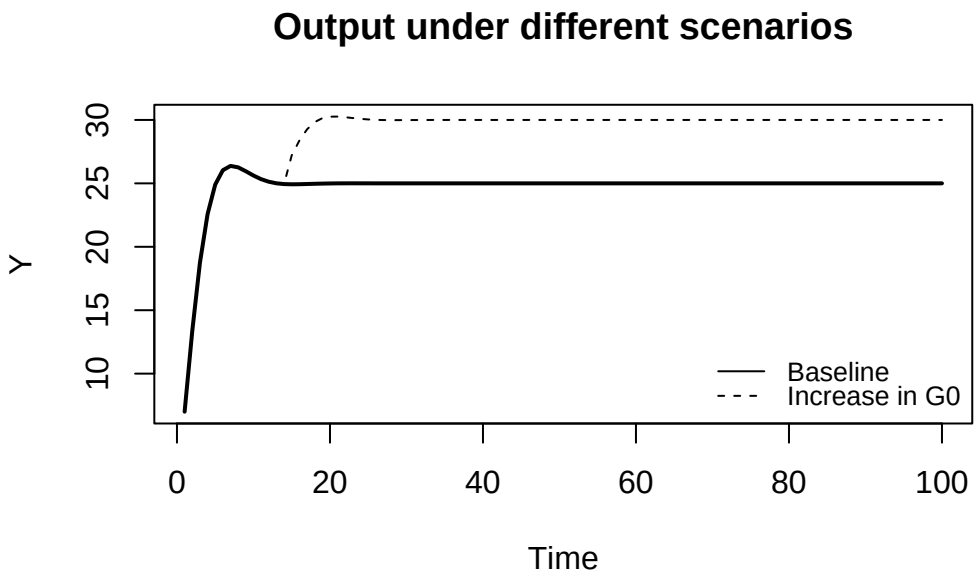
```
# Bar chart of different equilibrium solutions of Samuelson (1939) model
barplot(Y[,Q], ylab="Y", main="Output", names.arg=c("Baseline", "Increase in G0"))
```



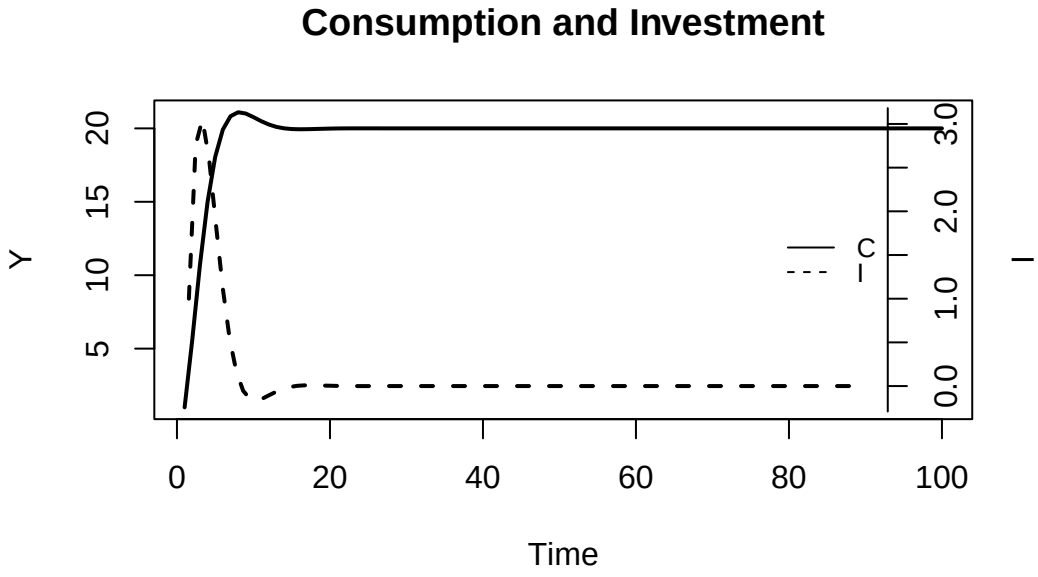

```
# Time series chart of output dynamics in Samuelson (1939) model
plot(Y[1, 1:100],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="Y")
title(main="Output", cex=0.8)
```



```r
# Time series chart of output dynamics for different scenarios in Samuelson (1939) model
plot(Y[1, 1:100],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="Y", ylim=range(min(Y[1,],
title(main="Output under different scenarios", cex=0.8)
lines(Y[2, 1:100],lty=2)
legend("bottomright", legend=c("Baseline", "Increase in G0"),
       lty=1:2, cex=0.8, bty = "n", y.intersp=0.8)
```




```r
# Time series chart of Samuelson (1939) model with separate axes for consumption and investme
plot(C[1, 1:100], type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="Y")
title(main="Consumption and Investment", cex=0.8)
par(mar = c(5, 4, 4, 4) + 0.3)
par(new = TRUE)
plot(I[1, 1:100], type="l", col=1, lwd=2, lty=2, font.main=1, cex.main=1,ylab = '', axes=FALS
     xlab = '', ylim = range(I[1, 1:100]), cex=0.8)
axis(side = 4, at=pretty(range(I[1, 1:100])))
mtext("I", side = 4, line = 3)
legend("right", legend=c("C", "I"),
       lty=1:2, cex=0.8, bty = "n", y.intersp=0.8)
```


Consumption and Investment
Y
5
10
15
20
C
I
0.0
1.0
2.0
3.0
0
20
40
60
80
100
Time

Python code

```
#### Plots

# Load matplot library
import matplotlib.pyplot as plt

# Bar chart of different equilibrium solutions of Samuelson (1939) model
scenario_labels = ["Baseline", "Increase in G0"]
plt.bar(scenario_labels, Y[:, Q - 1])
plt.xlabel("Scenario")
plt.ylabel("Y")
plt.title("Output")
plt.show()

# Time series chart of output dynamics in Samuelson (1939) model
plt.plot(range(1, Q), Y[0, 0:Q - 1], color='black', linewidth=2, linestyle='-')
plt.xlabel("Time")
plt.ylabel("Y")
plt.title("Output", fontsize=10)
plt.show()

# Time series chart of output dynamics for different scenarios in Samuelson
#(1939) model
plt.plot(range(1, Q), Y[0, 0:Q - 1], color='black', linewidth=1, linestyle='-')
plt.plot(range(1, Q), Y[1, 0:Q - 1], color='black', linewidth=1, linestyle='--')
plt.xlabel("Time")
plt.ylabel("Y")
plt.title("Output under different scenarios", fontsize=10)
plt.legend(["Baseline", "Increase in G0"], loc='lower right')
plt.show()

# Time series chart of Samuelson (1939) model with separate axes for consumption
# and investment
fig, ax1 = plt.subplots()
ax1.plot(range(1, Q), C[0, 0:Q - 1], color='black', linewidth=2, linestyle='-',
         label='C')
ax1.set_xlabel("Time")
ax1.set_ylabel("C", color='black')
ax1.tick_params(axis='y', labelcolor='black')
ax2 = ax1.twinx()
ax2.plot(range(1, Q), I[0, 0:Q - 1], color='black', linewidth=2, linestyle='--',
         label='I')
ax2.set_ylabel("I", color='black')
ax2.tick_params(axis='y', labelcolor='black')
plt.title("Consumption and Investment", fontsize=10)
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='right')
plt.show()
```

2.4 How to create a directed graph of a model

[2]

[3]

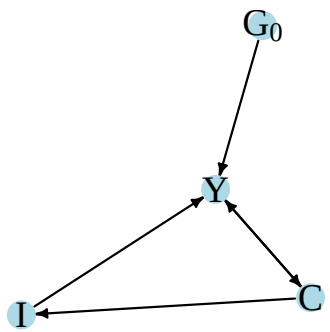
```
## Create directed graph
# Construct auxiliary Jacobian matrix for 4 variables:
  # endogenous: (1) Y, (2) C, (3) I
  # exogenous: (4) G0
# where non-zero elements in regular Jacobian are set to 1 and zero elements are unchanged

              #1 2 3 4
M_mat=matrix(c(0,1,1,1, #1
               1,0,0,0, #2
               0,1,0,0, #3
               0,0,0,0), #4
               4, 4, byrow=TRUE)
```

[2] See Fennell et al. (2015) for a neat exposition.

[3] To create the directed graph, we rely on external libraries which you may have to install first. In *R*, this can be accomplished with `install.packages("igraph")` and in Python with `pip install networkx`. Once that library is installed, you only need to activate it in each session before you use it. In *R*, you can do this by executing `library(igraph)` and in *Python* through `import networkx`.

```r
# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg=graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c("Y", "C", "I", expression(G[0]))

# Plot directed graph matrix
plot(dg, main="Directed graph of Samuelson model", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

Directed graph of Samuelson model



Python code

```
#Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# Construct auxiliary Jacobian matrix for 4 variables
  # endogenous: (1) Y, (2) C, (3) I
  # exogenous: (4) G0
# where non-zero elements in regular Jacobian are set to 1 and zero elements are
# unchanged
M_mat = np.array([[0, 1, 1, 1],
          [1, 0, 0, 0],
          [0, 1, 0, 0],
          [0, 0, 0, 0]])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {0: "Y", 1: "C", 2: "I", 3: "$G_0$"}

# Plot the directed graph
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=500, node_color='lightblue', f
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.title("Directed graph of Samuelson model", fontsize=12)
plt.axis('off')
plt.show()
```

1.
2.

3.

2.5 Appendix: How to simulate dynamic model in continuous time

```
### Simulate continuous time version of Keynesian goods market model

#Clear the environment
rm(list=ls(all=TRUE))

# Set number of periods for which you want to simulate
Q=800

# Set number of parameterisations that will be considered
S=1

# Set fixed parameter values
c0=3
c1=0.8
```

```
k=8
delta=0.01

# Construct matrices in which values for different periods will be stored; initialise at 1
Y=matrix(data=1, nrow=S, ncol=Q)
C=matrix(data=1, nrow=S, ncol=Q)

#Construct matrices for exogenous variable
I0=matrix(data=5, nrow=S, ncol=Q)

#Solve this system recursively based on the initialisation
  for (t in 2:Q){
    for (iterations in 1:1000){
    Y[1,t] = Y[1,t-1] + delta*(k*(C[1,t-1] + I0[1,t-1] - Y[1,t-1]))
    C[1,t] = c0 + c1*Y[1,t]
    } # close within-period loop
  } # close time loop


# Verify solutions for Y
(c0+I0[1,Q])/(1-c1)
```

```
Y[1,Q]
```

Python code

```
### Simulate continuous time version of Keynesian goods market model

# Load NumPy
import numpy as np

# Set number of periods for which you want to simulate
Q = 800

# Set number of parameterizations that will be considered
S = 1

# Set fixed parameter values
c0 = 3
c1 = 0.8
k = 8
delta = 0.01

# Initialize matrices to store values for different periods
Y = np.ones((S, Q))
C = np.ones((S, Q))

# Initialize the matrix for the exogenous variable
I0 = np.full((S, Q), 5)

# Solve this system recursively based on the initialization
for t in range(1, Q):
    for iterations in range(1000):
        Y[0, t] = Y[0, t - 1] + delta * (k * (C[0, t - 1] + I0[0, t - 1] - Y[0, t - 1]))
        C[0, t] = c0 + c1 * Y[0, t]

# Verify the solution for Y at time Q
(c0 + I0[0, Q - 1]) / (1 - c1)

Y[0,Q - 1]
```

2.6 References

Part I

Static Models

3 A Neoclassical Macro Model

3.1 Overview

3.2 The Model

[1]

[1] See the analytical discussion below for a derivation of equations Equation ?? -Equation ?? and Equation ?? from optimising microfoundations.

[2]

[3]

3.3 Simulation

3.3.1 Parameterisation

[2]Households are assumed to form rational expectations. In a deterministic setting, this implies perfect foresight so that expected and actual future income coincide.

[3]See the analytical discussion below for more details on Ricardian Equivalence.

3.3.2 Simulation code

```
# Clear the environment
rm(list=ls(all=TRUE))
```

```
# Set number of scenarios (including baseline)
S=6

#Create vector in which equilibrium solutions from different parameterisations will be stored
Y_star=vector(length=S) # Income/output
w_star=vector(length=S) # Real wage
C_star=vector(length=S) # Consumption
I_star=vector(length=S) # Investment
r_star=vector(length=S) # Real interest rate
rn_star=vector(length=S) # Nominal interest rate
N_star=vector(length=S) # Employment
P_star=vector(length=S) # Price level

# Create and parameterise exogenous variables/parameters that will be shifted
M0=vector(length=S) # money supply
G0=vector(length=S) # government expenditures
A=vector(length=S)  # productivity
Yf=vector(length=S) # expected future income
b1=vector(length=S) # household preference for leisure
M0[]=5
G0[]=1
A[]=2
Yf[]=1
b1[]=0.4

# Set parameter values for different scenarios
M0[2]=6   # scenario 2: monetary expansion
G0[3]=2   # scenario 3: fiscal expansion
A[4]=2.5  # scenario 4: productivity boost
Yf[5]=0.2 # scenario 5: lower expected future income
b1[6]=0.8 # scenario 6: increased preference for leisure

#Set constant parameter values
a=0.3  # Capital elasticity of output
b2=0.9 # discount rate
b3=0.6 # household preference for money
K=5    # Exogenous capital stock
pe=0.02 # Expected rate of inflation
Gf=1  # Future government spending

# Initialise endogenous variables at arbitrary positive value
w = C = I = Y = r = N = P = 1
```

```
#Solve this system numerically through 1000 iterations based on the initialisation
for (i in 1:S){

for (iterations in 1:1000){

  #Model equations

  #(1) Cobb-Douglass production function
  Y = A[i]*(K^a)*N^(1-a)

  #(2) Labour demand
  w = A[i]*(1-a)*(K^a)*N^(-a)

  #(3) Labour supply
  N = 1 - (b1[i])/w

  #(4) Consumption demand
  C = (1/(1+b2+b3))*(Y - G0[i] + (Yf[i]-Gf)/(1+r) -b1[i]*(b2+b3)*log(b1[i]/w))

  #(5) Investment demand, solved for r
  r=(I^(a-1))*a*A[i]*N^(1-a)

  #(6) Goods market equilibrium condition, solved for I
  I = Y - C - G0[i]

  #(7) Nominal interest rate
  rn = r + pe

  #(8) Price level
  P = (M0[i]*rn)/((1+rn)*b3*C)

}

#Save results for different parameterisations in vector
Y_star[i]=Y
w_star[i]=w
C_star[i]=C
I_star[i]=I
r_star[i]=r
N_star[i]=N
P_star[i]=P
rn_star[i]=rn
```

```
}
```

Python code

```
import numpy as np

# Set the number of scenarios (including baseline)
S = 6

# Create arrays to store equilibrium solutions from different parameterizations
Y_star = np.empty(S)  # Income/output
w_star = np.empty(S)  # Real wage
C_star = np.empty(S)  # Consumption
I_star = np.empty(S)  # Investment
r_star = np.empty(S)  # Real interest rate
rn_star = np.empty(S)  # Nominal interest rate
N_star = np.empty(S)  # Employment
P_star = np.empty(S)  # Price level

# Create and parameterize exogenous variables/parameters that will be shifted
M0 = np.zeros(S)  # Money supply
G0 = np.zeros(S)  # Government expenditures
A = np.zeros(S)   # Productivity
Yf = np.zeros(S)  # Expected future income
b1 = np.zeros(S)  # Household preference for leisure

# Baseline parameterisation
M0[:] = 5
G0[:] = 1
A[:] = 2
Yf[:] = 1
b1[:] = 0.4

# Set parameter values for different scenarios
M0[1] = 6   # Scenario 2: monetary expansion
G0[2] = 2   # Scenario 3: fiscal expansion
A[3] = 2.5  # Scenario 4: productivity boost
Yf[4] = 0.2  # Scenario 5: lower expected future income
b1[5] = 0.8  # Scenario 6: increased preference for leisure

# Set constant parameter values
a = 0.3  # Capital elasticity of output
b2 = 0.9  # Discount rate
b3 = 0.6  # Household preference for money
K = 5  # Exogenous capital stock
pe = 0.02  # Expected rate of inflation
Gf = 1  # Future government spending

# Initialize endogenous variables at arbitrary positive values
w = C = I = Y = r = N = P = 1

# Solve this system numerically through 1000 iterations based on the initialization
for i in range(S):
    for iterations in range(1000):
        # Model equations
```

3.3.3 Plots

```
barplot(Y_star, ylab="Y", names.arg=c("1: Baseline", "2: Increase in M0", "3: Increase in G0
```

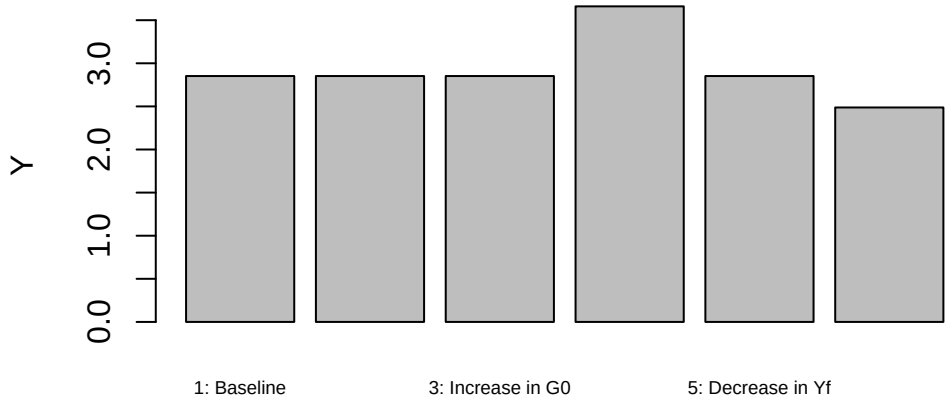


Figure 3.1: Output

```
barplot(P_star, ylab="P",  names.arg=c("1: Baseline", "2: Increase in M0", "3: Increase in G
```

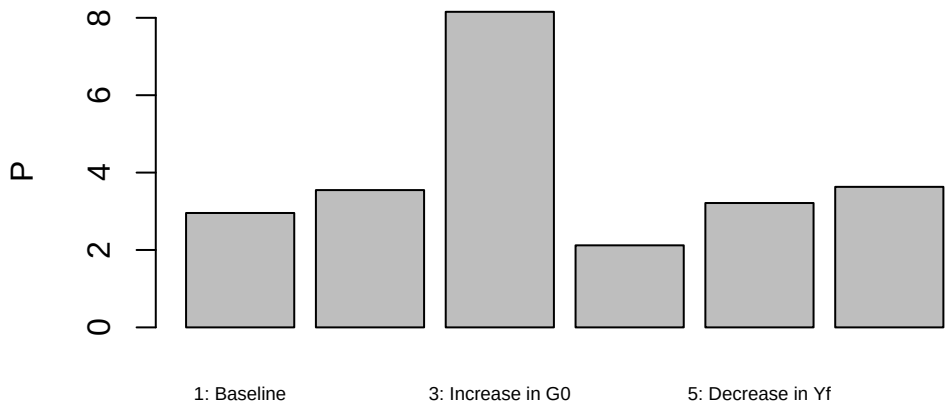


Figure 3.2: Price level

[4]

```
barplot(N_star, ylab="N",  names.arg=c("1: Baseline", "2: Increase in M0", "3: Increase in G
```

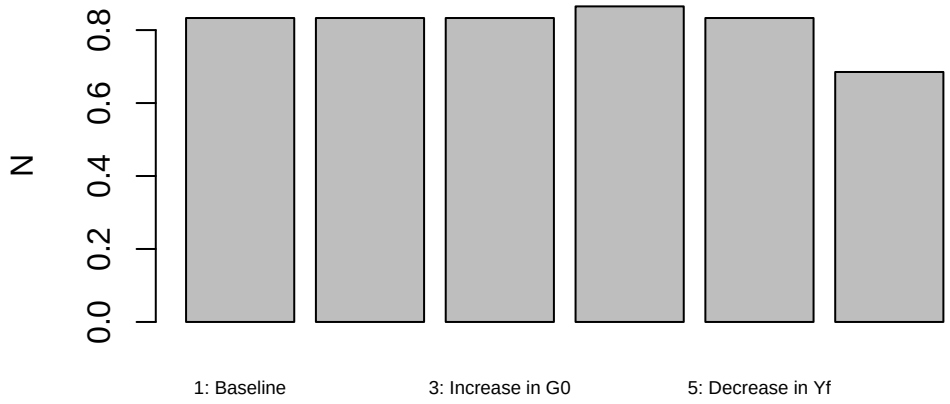


Figure 3.3: Employment

```
barplot(C_star, ylab="C",  names.arg=c("1: Baseline", "2: Increase in M0", "3: Increase in G
```

[4]This result is partly driven by the use of a non-separable utility function, see analytical discussion below. With a separable utility function, the increase in government spending would increase labour supply and thereby have effects on employment and output.

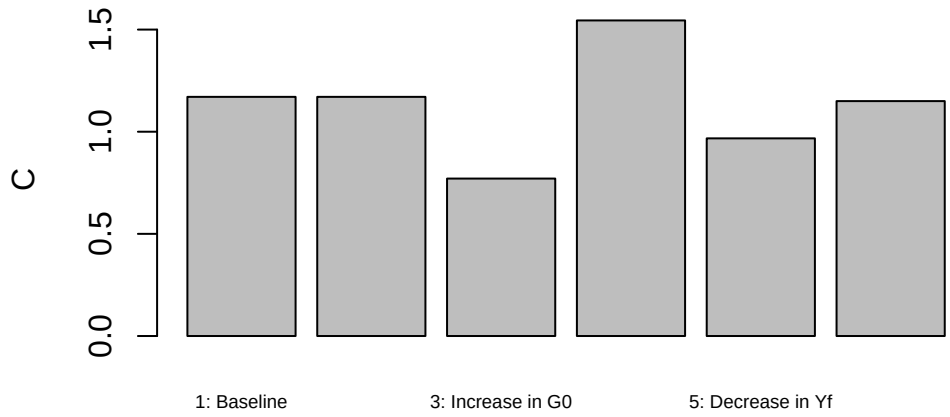


Figure 3.4: Consumption

```
barplot(I_star, ylab="I",  names.arg=c("1: Baseline", "2: Increase in M0", "3: Increase in G
```

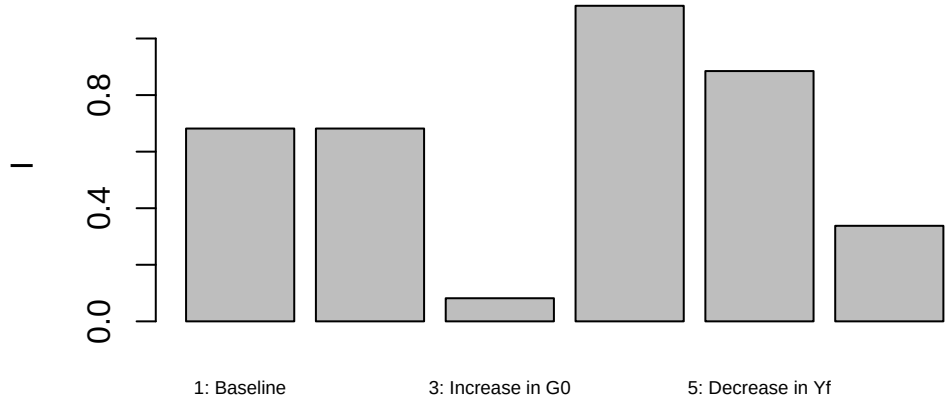


Figure 3.5: Investment

```
barplot(r_star, ylab="r", names.arg=c("1: Baseline", "2: Increase in M0", "3: Increase in G0
```

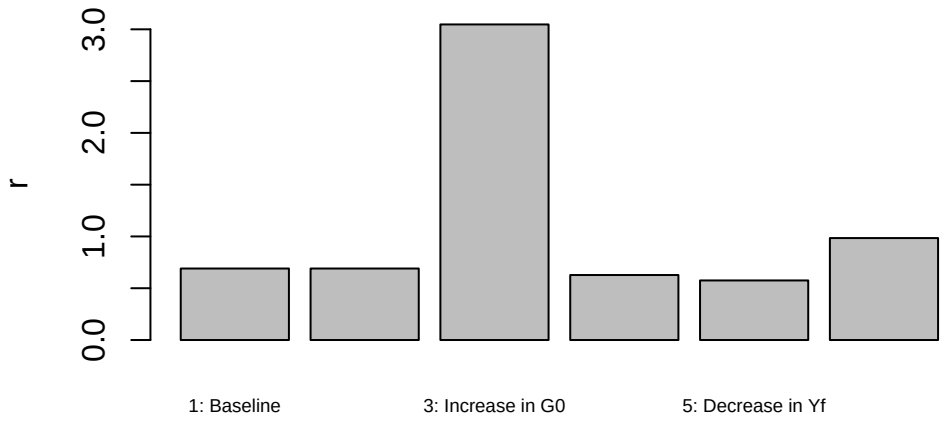


Figure 3.6: Interest rate

Python code

```
# Plot results (here for output only)
import matplotlib.pyplot as plt

scenario_names = ["1: Baseline", "2: Increase in M0", "3: Increase in G0",
                  "4: Increase in A", "5: Decrease in Yf", "6: Increase in b1"]

# Output
plt.bar(scenario_names , Y_star)
plt.ylabel('Y')
plt.xticks(rotation=45, ha="right")  # Rotate x-axis labels for better readability
plt.tight_layout()  # Ensure the labels fit within the plot area
plt.show()
```

3.4 Directed graph

```
## Create directed graph
# Construct auxiliary Jacobian matrix for 13 variables:
# Y w   N   C   I   r   P   rn MO   GO A Yf Md
# where non-zero elements in regular Jacobian are set to 1 and zero elements are unchanged

M_mat=matrix(c(0,0,1,0,0,0,0,0,0,0,1,0,0,
               0,0,1,0,0,0,0,0,0,0,1,0,0,
               0,1,0,0,0,0,0,0,0,0,0,0,0,
               1,1,0,0,0,1,0,0,0,1,0,1,0,
               0,0,1,0,0,1,0,0,0,0,0,0,0,
               1,0,0,1,1,0,0,0,0,1,0,0,0,
               0,0,0,0,0,0,0,0,1,0,0,0,1,
               0,0,0,0,0,1,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,1,0,0,1,1,0,0,0,0,0), 13, 13, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg= graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c("Y","w","N","C","I","r","P", expression(r[n]), expression(M[0]),expression(G[0])

# Plot directed graph
plot(dg, main="", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

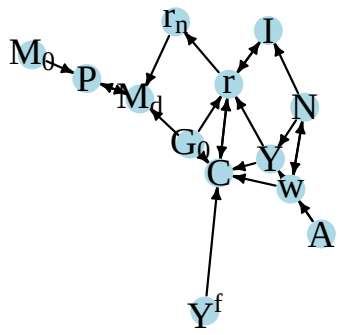


Figure 3.7: Directed graph

Python code

```
# Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# Construct the auxiliary Jacobian matrix
M_mat = np.array([
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]
])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {
    0: "Y",
    1: "w",
    2: "N",
    3: "C",
    4: "I",
    5: "r",
    6: "P",
    7: r"$r_n$",
    8: r"$M_0$",
    9: r"$G_0$",
    10: "A",
    11: r"$Y^f$",
    12: r"$M_d$"
}

# Plot the directed graph
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=300, node_color='lightblue',
        font_size=10)
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.axis('off')
plt.show()
```

3.5 Analytical discussion: derivation of behavioural functions

3.5.1 The firm's problem: profit maximisation

[5]

[5]See Garín, Lester, and Sims (2021, chap. 12) for a more elaborate version where the firm maximises its lifetime value. The resulting investment function is very similar.

3.5.2 The government's budget constraint

[6]

3.5.3 The household's problem: intertemporal utility maximisation and Ricardian Equivalence

[7]

[6] See Garín, Lester, and Sims (2021, chap. 13) for a more detailed discussion of the government's budget constraints.

[7] See Garín, Lester, and Sims (2021), pp.280-289), on the differences between a separable and a non-separable preference specification. Note also that for simplicity, the utility function omits the utility from future leisure.

8

[8]See Garín, Lester, and Sims (2021, chap. 14) for a more detailed discussion of the household's budget constraints.

References

4 An IS-LM Model

4.1 Overview

The General Theory of Employment, Interest and Money

4.2 The Model

4.3 Simulation

4.3.1 Parameterisation

4.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

# Set number of scenarios (including baseline)
S=6

#Create vector in which equilibrium solutions from different parameterisations will be stored
Y_star=vector(length=S) # Income/output
C_star=vector(length=S) # Consumption
I_star=vector(length=S) # Investment
r_star=vector(length=S) # Real interest rate
N_star=vector(length=S) # Employment
U_star=vector(length=S) # Unemployment rate

# Set exogenous variables that will be shifted
i0=vector(length=S) # autonomous investment
m0=vector(length=S) # Autonomous demand for money
M0=vector(length=S) # money supply
G0=vector(length=S) # government spending
T0=vector(length=S) # taxes
i0[]=2
m0[]=6
M0[]=5
G0[]=1
T0[]=1
```

```
## Construct scenarios
# scenario 2: fall in animal spirits
i0[2]=1

#scenario 3: increase in liquidity preference
m0[3]=7

# scenario 4: monetary expansion
M0[4]=6

# scenario 5: reduction in tax rate
T0[5]=0

# scenario 6: fiscal expansion
G0[6]=2

#Set constant parameter values
c0=2   # Autonomous consumption
c1=0.6 # Sensitivity of consumption with respect to the income (marginal propensity to consu
i1=0.1 # Sensitivity of investment with respect to the interest rate
m1=0.2 # Sensitivity of money demand with respect to income
m2=0.4 # Sensitivity of money demand with respect to interest rate
a=1.5  # labour coefficient
Nf=18  # Full employment/labour force

# Initialise endogenous variables at some arbitrary positive value
Y = C = I =r = N = U = 1

#Solve this system numerically through 1000 iterations based on the initialisation

for (i in 1:S){

  for (iterations in 1:1000){

    #Model equations

    # Goods market equilibrium
    Y = C + I + G0[i]

    # Consumption demand
    C = c0 + c1*(Y-T0[i])
```

```
    # Investment demand
    I = i0[i] - i1*r

    # Money market, solved for interest rate
    r = (m0[i] - M0[i])/m2 + m1*Y/m2

    # Employment
    N = a*Y

    #Unemployment rate
    U = (1 - N/Nf)

  }
  #Save results for different parameterisations in vector
  Y_star[i]=Y
  C_star[i]=C
  I_star[i]=I
  r_star[i]=r
  N_star[i]=N
  U_star[i]=U
}
```

Python code

```
import numpy as np

# Set the number of scenarios (including baseline)
S = 6

# Create arrays to store equilibrium solutions from different parameterizations
Y_star = np.empty(S)  # Income/output
C_star = np.empty(S)  # Consumption
I_star = np.empty(S)  # Investment
r_star = np.empty(S)  # Real interest rate
N_star = np.empty(S)  # Employment
U_star = np.empty(S)  # Unemployment rate

# Set exogenous variables that will be shifted
i0 = np.zeros(S)  # Autonomous investment
m0 = np.zeros(S)  # Autonomous demand for money
M0 = np.zeros(S)  # Money supply
G0 = np.zeros(S)  # Government spending
T0 = np.zeros(S)  # Taxes

# Baseline parameterisation
i0[:] = 2
m0[:] = 6
M0[:] = 5
G0[:] = 1
T0[:] = 1

# Construct scenarios
# scenario 2: fall in animal spirits
i0[1] = 1

# scenario 3: increase in liquidity preference
m0[2] = 7

# scenario 4: monetary expansion
M0[3] = 6

# scenario 5: reduction in tax rate
T0[4] = 0

# scenario 6: fiscal expansion
G0[5] = 2
```

```
# Set constant parameter values
c0 = 2  # Autonomous consumption
c1 = 0.6  # Sensitivity of consumption with respect to income (marginal propensity to consu
i1 = 0.1  # Sensitivity of investment with respect to the interest rate
m1 = 0.2  # Sensitivity of money demand with respect to income
m2 = 0.4  # Sensitivity of money demand with respect to the interest rate
a = 1.5  # labor coefficient
Nf = 18  # Full employment/labor force
```

4.3.3 Plots

[1]

```
barplot(Y_star, ylab="Y", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise liq. pre
                                      "4:Monetary exp.", "5:Tax cut", "6:Fiscal exp."), cex.n
```

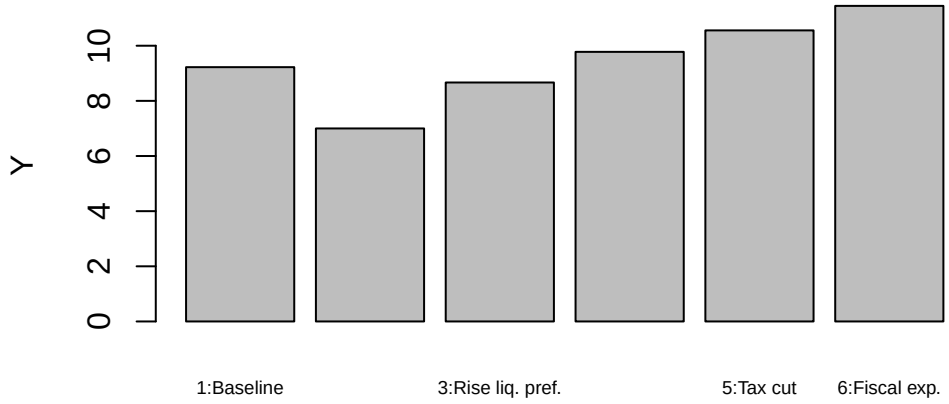


Figure 4.1: Output

```
barplot(r_star, ylab="r", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise liq. pre
                                      "4:Monetary exp.", "5:Tax cut", "6:Fiscal exp."), cex.n
```

[1]The analytical discussion below shows formally that fiscal policy is more effective than monetary policy if $m_2 > i_2$.

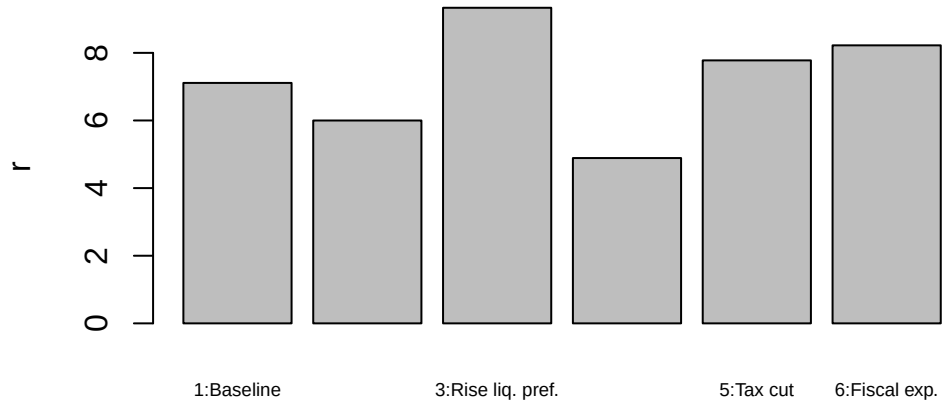


Figure 4.2: Interest rate

```
barplot(U_star*100, ylab="U (%)", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise
                                              "4:Monetary exp.", "5:Tax cut", "6:Fiscal exp.
```

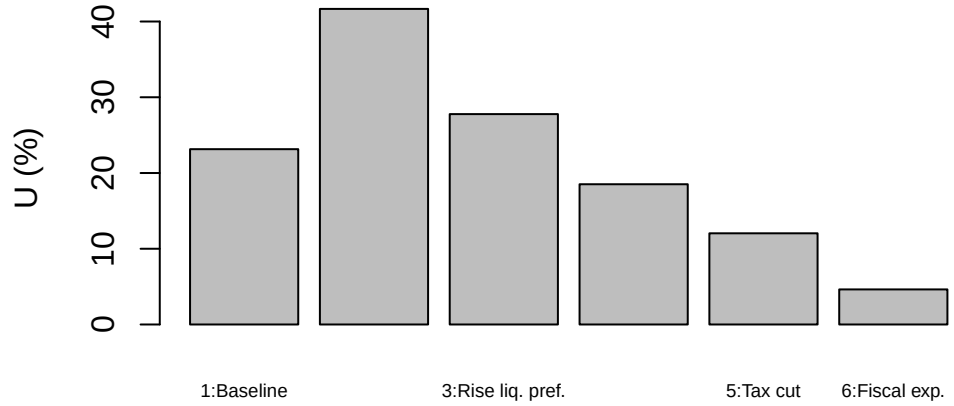


Figure 4.3: Unemployment

```
barplot(I_star, ylab="I", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise liq. pre
                                      "4:Monetary exp.", "5:Tax cut", "6:Fiscal exp."), cex.n
```

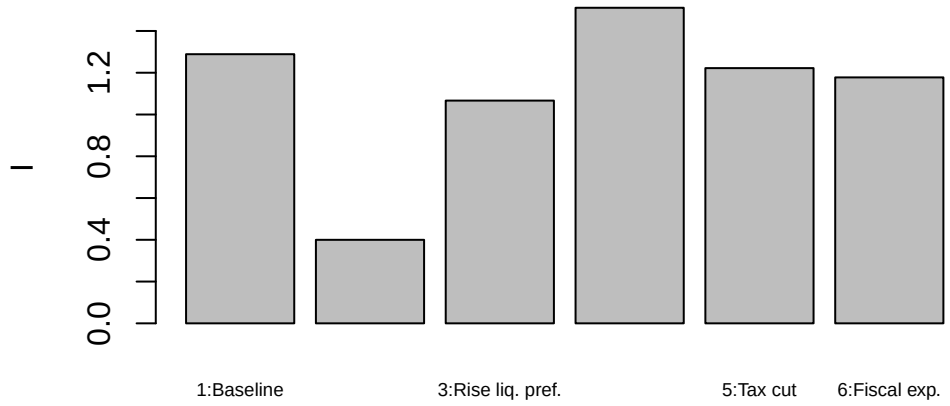


Figure 4.4: Investment

```
barplot(C_star, ylab="C", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise liq. pre
                                      "4:Monetary exp.", "5:Tax cut", "6:Fiscal exp."), cex.n
```

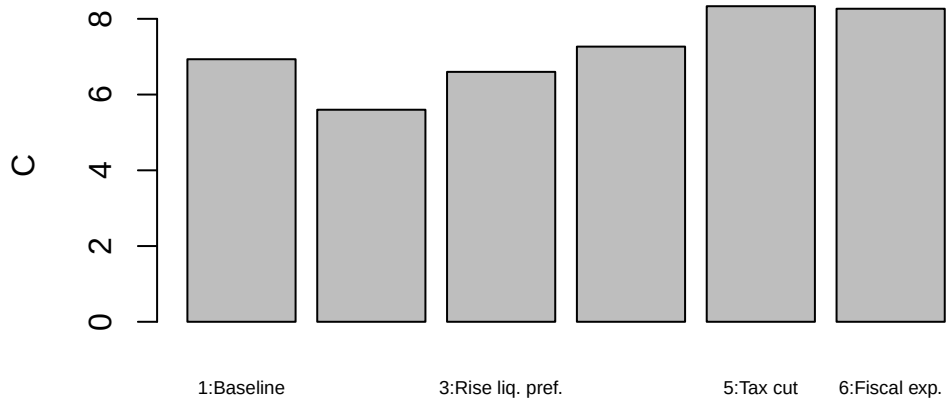


Figure 4.5: Consumption

```
barplot(N_star, ylab="N", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise liq. pre
```

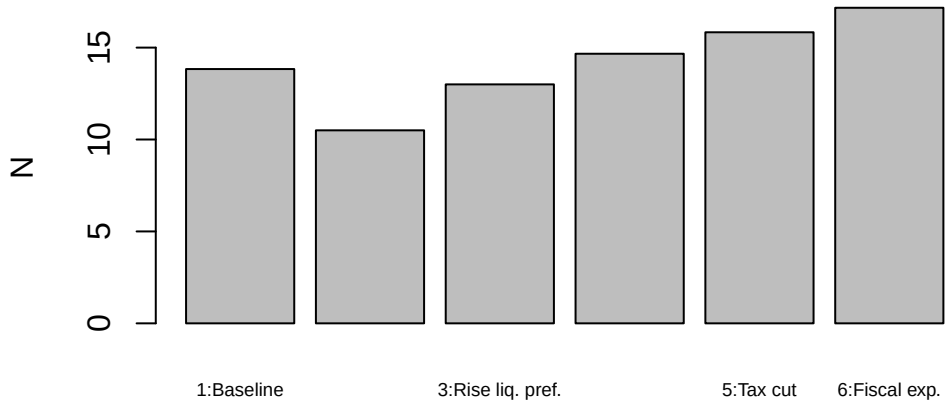


Figure 4.6: Employment

Python code

```
# Plot results (here only for output)
import matplotlib.pyplot as plt

scenario_names = ["1:Baseline", "2:Fall animal spirits", "3:Rise liq. pref.",
                  "4:Monetary exp.", "5:Tax cut", "6:Fiscal exp."]

plt.bar(scenario_names, Y_star)
plt.ylabel('Y')
plt.xticks( scenario_names, rotation=45, fontsize=6)
plt.show()
```

4.4 Directed graph

```
# Construct auxiliary Jacobian matrix for 11 variables: Y, C, I, G, T, r, M0, N, i0, m0, Md
# where non-zero elements in regular Jacobian are set to 1 and zero elements are unchanged

M_mat=matrix(c(0,1,1,1,0,0,0,0,0,0,0,
               1,0,0,0,1,0,0,0,0,0,0,
               0,0,0,0,0,1,0,0,1,0,0,
```

```
              0,0,0,0,0,0,0,0,0,0,0,
              0,0,0,0,0,0,0,0,0,0,0,
              0,0,0,0,0,0,1,0,0,0,1,
              0,0,0,0,0,0,0,0,0,0,0,
              1,0,0,0,0,0,0,0,0,0,0,
              0,0,0,0,0,0,0,0,0,0,0,
              0,0,0,0,0,0,0,0,0,0,0,
              1,0,0,0,0,1,0,0,0,1,0), 11, 11, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg= graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c("Y", "C", "I", expression(G[0]), expression(T[0]), "r", expression(M[0]), "N", e

# Plot directed graph
plot(dg, main="", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

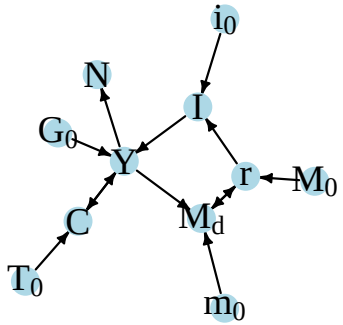


Figure 4.7: Directed graph of IS-LM model

Python code

```
# Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np


# Define the auxiliary Jacobian matrix
M_mat = np.array([
    [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)


# Define node labels
nodelabs = {
    0: "Y",
    1: "C",
    2: "I",
    3: r"$G_0$",
    4: r"$T_0$",
    5: "r",
    6: r"$M_0$",
    7: "N",
    8: r"$i_0$",
    9: r"$m_0$",
    10: r"$M_d$"
}

# Plot the directed graph
pos = nx.spring_layout(G, seed=43)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=300, node_color='lightblue',
        font_size=10)
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.axis('off')
plt.show()
```

4.5 Analytical discussion

-
-
-

4.5.1 Calculate equilibrium fiscal multiplier

```
Y_star[6]-Y_star[1] # numerical approach
```

```
m2/((1-c1)*m2+i1*m1) # analytical approach
```

Python code

```
Y_star[5]-Y_star[0] # numerical approach

m2/((1-c1)*m2+i1*m1) # analytical approach
```

References

5 A Neoclassical Synthesis Model (IS-LM-AS-AD)

5.1 Overview

The General Theory of Employment, Interest and Money

5.2 The Model

[1]

5.3 Simulation

5.3.1 Parameterisation

[1] See the notes on the Classical Model (here) for a formal derivation of the labour demand and supply curves from optimisation. A minor modification is that here we work with a normalisation of the term for leisure in the household's log-utility function, $\ln(1 - \frac{N}{N^{\bar{f}}})$, to allow N to be larger than unity.

5.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

# Set number of scenarios (including baseline)
S=6

#Create vector in which equilibrium solutions from different parameterisations will be stored
Y_star=vector(length=S) # Income/output
C_star=vector(length=S) # Consumption
I_star=vector(length=S) # Investment
r_star=vector(length=S) # Real interest rate
N_star=vector(length=S) # Employment
U_star=vector(length=S) # Unemployment rate
P_star=vector(length=S) # Price level
w_star=vector(length=S) # Real wage
W_star=vector(length=S) # Nominal wage

# Set exogenous variables that will be shifted
i0=vector(length=S) # autonomous investment (animal spirits)
M0=vector(length=S) # money supply
G0=vector(length=S) # government spending
```

```
P0=vector(length=S) # expected price level
A=vector(length=S)  # Exogenous productivity

### Construct different scenarios
# baseline
A[]=2
i0[]=2
M0[]=5
G0[]=1
P0[]=1

# scenario 2: fall in animal spirits
i0[2]=1.5

# scenario 3: increase in productivity
A[3]=3

# scenario 4: increase in expected price level
P0[4]=1.5

# scenario 5: monetary expansion
M0[5]=6

# scenario 6: fiscal expansion
G0[6]=2

#Set constant parameter values
c0=2   # Autonomous consumption
c1=0.6 # Sensitivity of consumption with respect to the income (marginal propensity to consu
i1=0.1 # Sensitivity of investment with respect to the interest rate
m1=0.2 # Sensitivity of money demand with respect to income
m2=0.4 # Sensitivity of money demand with respect to interest rate
Nf=5   # Full employment/labour force
K=4    # Exogenous capital stock
a=0.3  # Capital elasticity of output
b=0.4  # household preference for leisure
T0=1   # tax revenues
m0=6   # liquidity preference

# Initialise endogenous variables at some arbitrary positive value
Y = C = I = r = P = w = N = W = 1
```

```
#Solve this system numerically through 1000 iterations based on the initialisation

for (i in 1:S){

  for (iterations in 1:1000){

    #Model equations

    # Goods market equilibrium
    Y = C + I + G0[i]

    # Consumption demand
    C = c0 + c1*(Y-T0)

    # Investment demand
    I = i0[i] - i1*r

    # Money market, solved for interest rate
    r = (m0 - (M0[i]/P))/m2 + m1*Y/m2

    #Unemployment rate
    U = (1 - N/Nf)

    #Real wage
    w = A[i]*(1-a)*(K^a)*N^(-a)

    #Nominal wage
    W= (P0[i]*b*C)/(1- (N/Nf))

    #Price level
    P = W/w

    #Employment
    N = (Y/(A[i]*(K^a)))^(1/(1-a))

}

  #Save results for different parameterisations in vector
  Y_star[i]=Y
  C_star[i]=C
  I_star[i]=I
  r_star[i]=r
```

```
  N_star[i]=N
  U_star[i]=U
  P_star[i]=P
  w_star[i]=w
  W_star[i]=W

}
```

Python code

```
import numpy as np

# Set the number of scenarios (including baseline)
S = 6

# Create arrays to store equilibrium solutions from different parameterizations
Y_star = np.empty(S)  # Income/output
C_star = np.empty(S)  # Consumption
I_star = np.empty(S)  # Investment
r_star = np.empty(S)  # Real interest rate
N_star = np.empty(S)  # Employment
U_star = np.empty(S)  # Unemployment rate
P_star = np.empty(S)  # Price level
w_star = np.empty(S)  # Real wage
W_star = np.empty(S)  # Nominal wage

# Set exogenous variables that will be shifted
i0 = np.zeros(S)  # Autonomous investment (animal spirits)
M0 = np.zeros(S)  # Money supply
G0 = np.zeros(S)  # Government spending
P0 = np.zeros(S)  # Expected price level
A = np.empty(S)  # Exogenous productivity

# Construct different scenarios
# baseline
A[:] = 2
i0[:] = 2
M0[:] = 5
G0[:] = 1
P0[:] = 1

# scenario 2: fall in animal spirits
i0[1] = 1.5

# scenario 3: increase in productivity
A[2] = 3

# scenario 4: increase in expected price level
P0[3] = 1.5
```

5.3.3 Plots

```
barplot(Y_star, ylab="Y", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise product
                                      "4:Rise exp. price", "5:Monetary expan.", "6:Fiscal ex
```

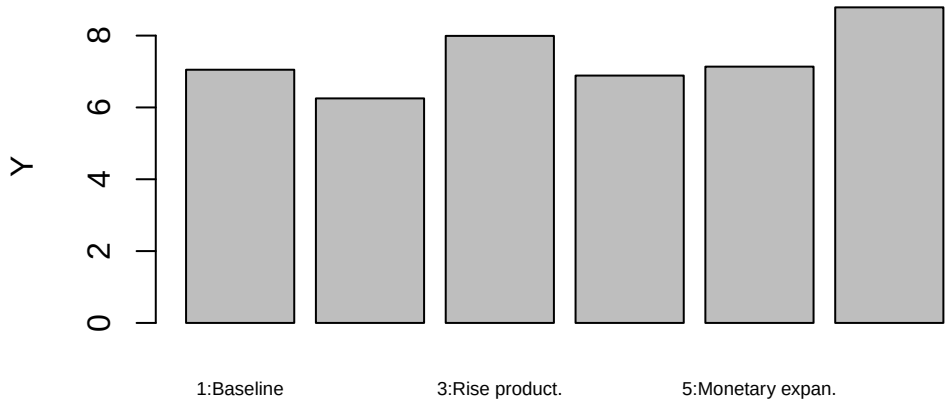


Figure 5.1: Output

```
barplot(P_star, ylab="P", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise product
                                      "4:Rise exp. price", "5:Monetary expan.", "6:Fiscal ex
```

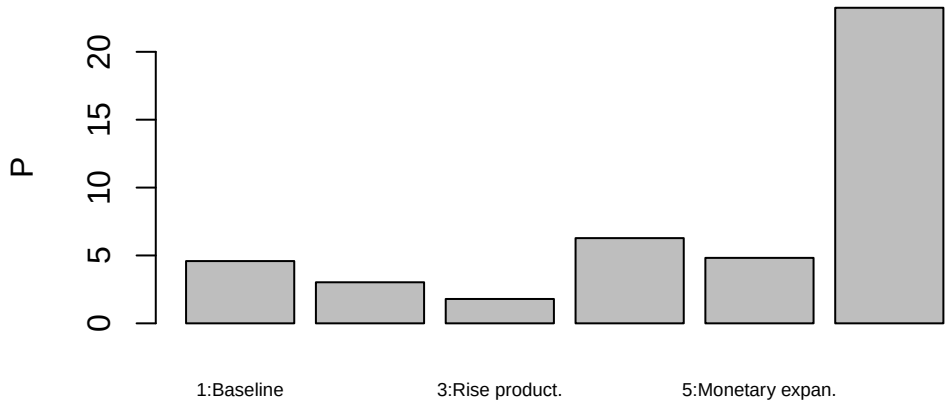


Figure 5.2: Price level

```
barplot(r_star, ylab="r", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise product
                                      "4:Rise exp. price", "5:Monetary expan.", "6:Fiscal ex
```

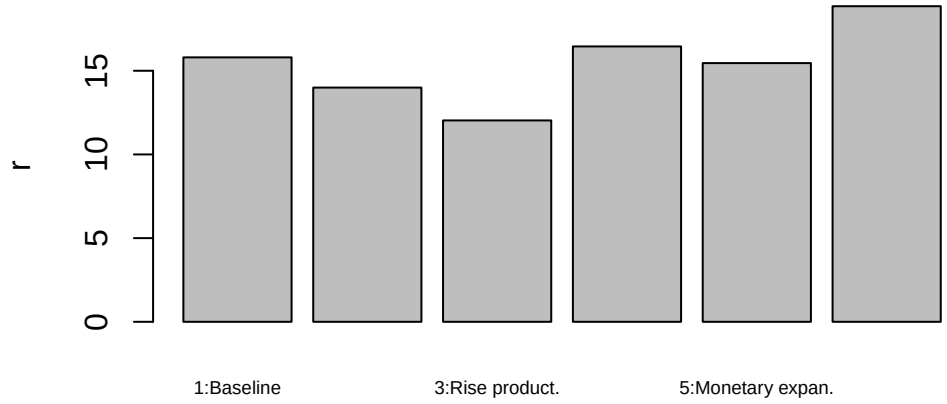


Figure 5.3: Interest rate

```
barplot(U_star*100, ylab="U (%)", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise
                                              "4:Rise exp. price", "5:Monetary expan.", "6:F
```

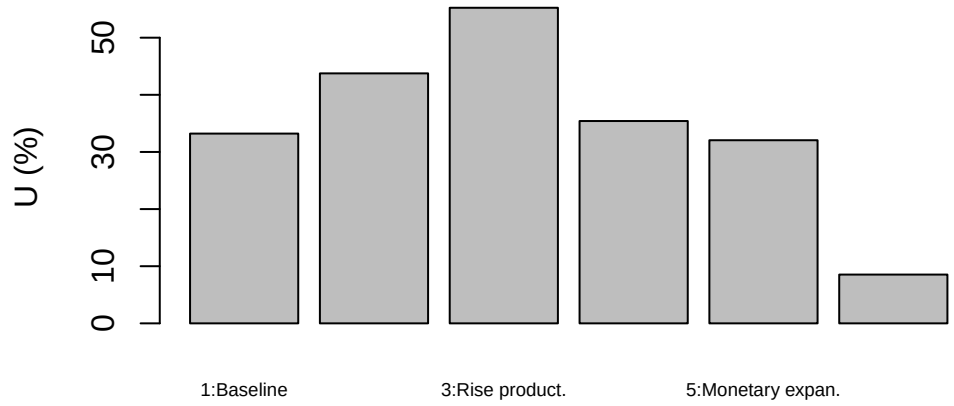


Figure 5.4: Unemployment rate

```
barplot(W_star, ylab="W", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise product
                                      "4:Rise exp. price", "5:Monetary expan.", "6:Fiscal ex
```

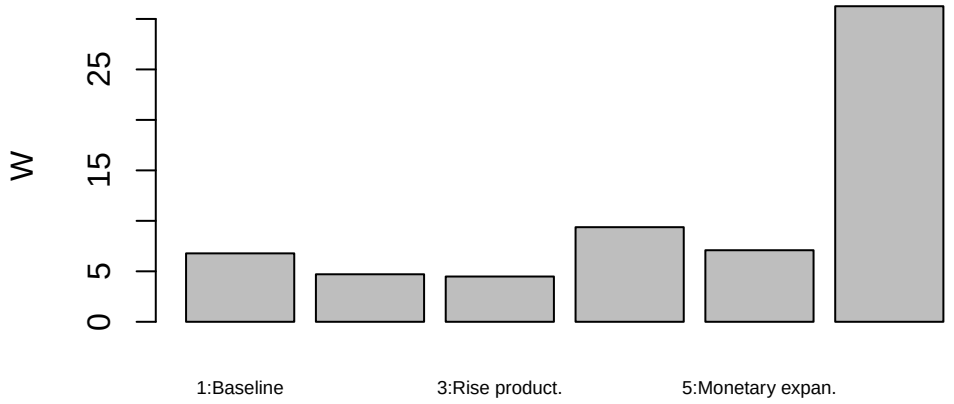


Figure 5.5: Nominal wage

```
barplot(w_star, ylab="w", names.arg=c("1:Baseline", "2:Fall animal spirits", "3:Rise product
                                      "4:Rise exp. price", "5:Monetary expan.", "6:Fiscal ex
```

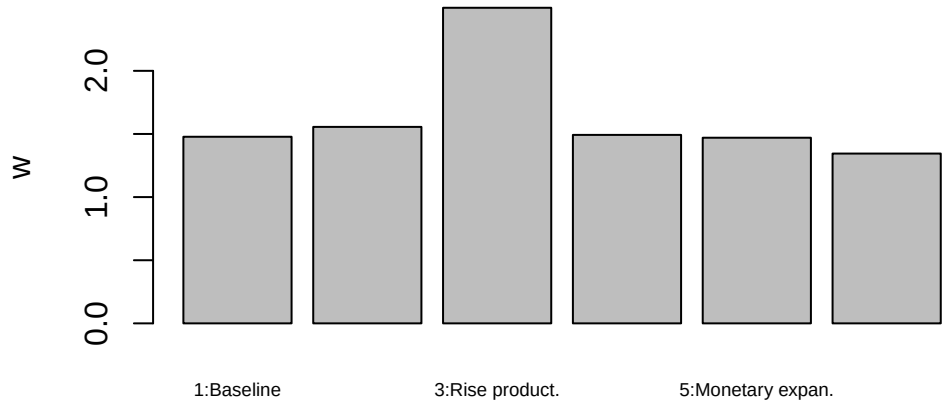


Figure 5.6: Real wage

Python code

```
# Plot results (here only for output)
import matplotlib.pyplot as plt

scenario_names = ["1:Baseline", "2:Fall animal spirits", "3:Rise product.",
                  "4:Rise exp. price", "5:Monetary expan.", "6:Fiscal expan."]

plt.bar(scenario_names, Y_star)
plt.ylabel('Y')
plt.xticks(scenario_names, rotation=45, fontsize=6)
plt.show()
```

5.4 Directed graph

```
## Create directed graph
# Construct auxiliary Jacobian matrix for 15 variables: Y, C, I, G, T, r, w, W, P, M0, N, i0

M_mat=matrix(c(0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,
               1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,
```

```
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                1,0,0,0,0,0,0,1,1,0,0,0,0,1,
                0,0,0,0,0,0,0,0,0,1,0,0,0,0,
                0,1,0,0,0,0,0,0,0,1,0,0,1,0,
                0,0,0,0,0,0,1,1,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                1,0,0,0,0,0,0,0,0,0,0,1,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                1,0,0,0,0,1,0,0,1,0,0,0,0,0,0), 15, 15, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg= graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c("Y", "C", "I", expression(G[0]), expression(T[0]), "r", "w", "W", "P", expressi

# Plot directed graph
plot(dg, main="", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

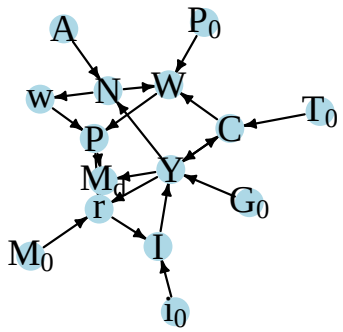


Figure 5.7: Directed graph of Neoclassical Synthesis model

Python code

```python
# Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np


# Define the Jacobian matrix
M_mat = np.array([
    [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)


# Define node labels
nodelabs = {
    0: "Y",
    1: "C",
    2: "I",
    3: r"$G_0$",
    4: r"$T_0$",
    5: "r",
    6: "w",
    7: "W",
    8: "P",
    9: r"$M_0$",
    10: "N",
    11: r"$i_0$",
    12: "A",
    13: r"$P_0$",
    14: r"$M_d$",
}

# Plot the directed graph
pos = nx.spring_layout(G, seed=43)
```

5.5 Analytical discussion

References

6 A Post-Keynesian Macro Model with Endogenous Money

6.1 Overview

[1]

Neoclassical Synthesis

[1] See Lavoie (2006), chap.1 and Exploring Economics for introductions. Lavoie (2014) and Hein (2014) provide more advanced treatments.

6.2 The Model

[2]

[2]For simplicity, it is assumed that all autonomous demand is debt-financed, i.e. there is no spending out of wealth.

6.3 Simulation

6.3.1 Parameterisation

6.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

# Set number of scenarios (including baseline)
S=6

#Create vector in which equilibrium solutions from different parameterisations will be stored
Y_star=vector(length=S)  # income/output
D_star=vector(length=S)  # (notional) credit-financed aggregate demand
ND_star=vector(length=S) # income-financed aggregate demand
r_star=vector(length=S)  # lending rate
```

```
N_star=vector(length=S)  # employment
U_star=vector(length=S)  # unemployment
P_star=vector(length=S)  # price level
w_star=vector(length=S)  # real wage
W_star=vector(length=S)  # nominal wage
i_star=vector(length=S)  # central bank rate
dL_star=vector(length=S) # change in loans
dM_star=vector(length=S) # change in bank deposits
dR_star=vector(length=S) # change in bank reserves

# Set exogenous variables that will be shifted
c=vector(length=S) # share of credit demand that is accommodated
d0=vector(length=S)# autonomous component of debt-financed aggregate demand
m=vector(length=S) # mark-up on lending rate
n=vector(length=S) # mark-up on prices
a=vector(length=S) # productivity

# Baseline parameterisation
c[]=0.8
d0[]=5
m[]=0.15
n[]=0.15
a[]=0.8

## Construct scenarios

# scenario 2: increase in credit rationing
c[2]=0.4

# scenario 3: increase in autonomous demand
d0[3]=10

# scenario 4: increase in interest rate mark-up
m[4]=0.3

# scenario 5: increase in price mark-up
n[5]=0.3

# scenario 6: increase in productivity
a[6]=0.4
```

```
#Set constant parameter values
b=0.5   # propensity to spend out of income
d1=0.8  # sensitivity of demand with respect to the interest rate
i0=0.01 # discretionary component of central bank rate
i1=0.5  # sensitivity of central bank rate with respect to price level
Nf=12   # full employment/labour force
h=0.8   # sensitivity of nominal wage with respect to unemployment
k=0.3   # desired reserve ratio
W0=2    # exogenous component of nominal wage

# Initialise endogenous variables at some arbitrary positive value
Y = D = ND = r = N = U = P = w = W = i = dL = dR = dM = 1

#Solve this system numerically through 1000 iterations based on the initialisation
for (j in 1:S){

  for (iterations in 1:1000){

    #Model equations

    # (1) Goods market
    Y = ND + c[j]*D

    # (2) Not-debt financed component of aggregate demand
    ND = b*Y

    # (3) Debt-financed component of aggregate demand
    D= d0[j] - d1*r

    # (4) Policy rate
    i = i0 + i1*P

    # (5) Lending rate
    r = (1+m[j])*i

    # (6) Change in loans
    dL = c[j]*D

    # (7) Change in deposits
    dM = dL

    # (8) Change in reserves
```

```
    dR = k*dM

    # (9) Price level
    P = (1+n[j])*a[j]*W

    # (10) Nominal wage
    W = W0 - h*(U)

    # (11) Real wage
    w = 1/((1+n[j])*a[j])

    # (12) Employment
    N = a[j]*Y

    # (13) Unemployment rate
    U = (Nf - N)/Nf

  }

  #Save results for different parameterisations in vector
  Y_star[j]=Y
  D_star[j]=D
  ND_star[j]=ND
  r_star[j]=r
  N_star[j]=N
  U_star[j]=U
  P_star[j]=P
  w_star[j]=w
  W_star[j]=W
  i_star[j]=i
  dL_star[j]=dL
  dM_star[j]=dM
  dR_star[j]=dR
}
```

Python code

```
# Load NumPy library
import numpy as np

# Set the number of scenarios (including baseline)
S = 6

# Create arrays to store equilibrium solutions
Y_star = np.zeros(S)  # income/output
D_star = np.zeros(S)  # (notional) credit-financed aggregate demand
ND_star = np.zeros(S)  # income-financed aggregate demand
r_star = np.zeros(S)  # lending rate
N_star = np.zeros(S)  # employment
U_star = np.zeros(S)  # unemployment
P_star = np.zeros(S)  # price level
w_star = np.zeros(S)  # real wage
W_star = np.zeros(S)  # nominal wage
i_star = np.zeros(S)  # central bank rate
dL_star = np.zeros(S)  # change in loans
dM_star = np.zeros(S)  # change in bank deposits
dR_star = np.zeros(S)  # change in bank reserves

# Set exogenous variables that will be shifted
c = np.zeros(S)  # share of credit demand that is accommodated
d0 = np.zeros(S)  # autonomous component of debt-financed aggregate demand
m = np.zeros(S)  # mark-up on lending rate
n = np.zeros(S)  # mark-up on prices
a = np.zeros(S)  # productivity

# Baseline parameterisation
c[:] = 0.8
d0[:] = 5
m[:] = 0.15
n[:] = 0.15
a[:] = 0.8

# Construct scenarios
# Scenario 2: increase in credit rationing
c[1] = 0.4

# Scenario 3: increase in autonomous demand
d0[2] = 10
```

6.3.3 Plots

```
barplot(Y_star, ylab="Y", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD",
                                      "4:rise bank markup","5:rise firm markup", "6:rise proc
```

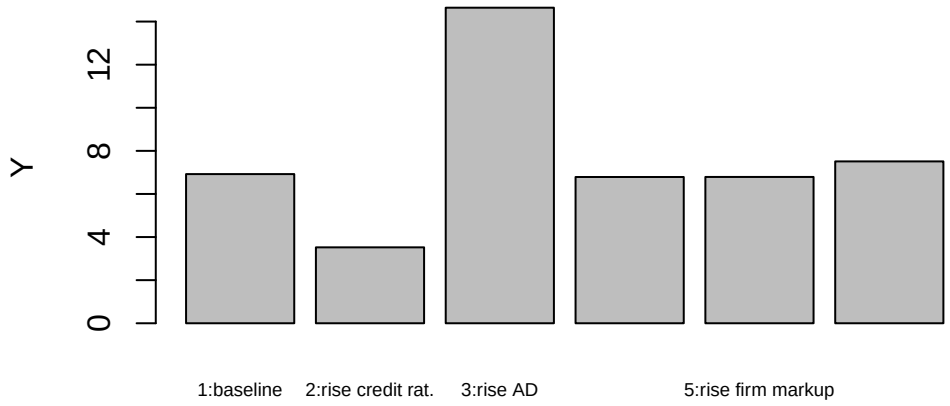


Figure 6.1: Output

```
barplot(P_star, ylab="P", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD",
                                      "4:rise bank markup","5:rise firm markup", "6:rise proc
```

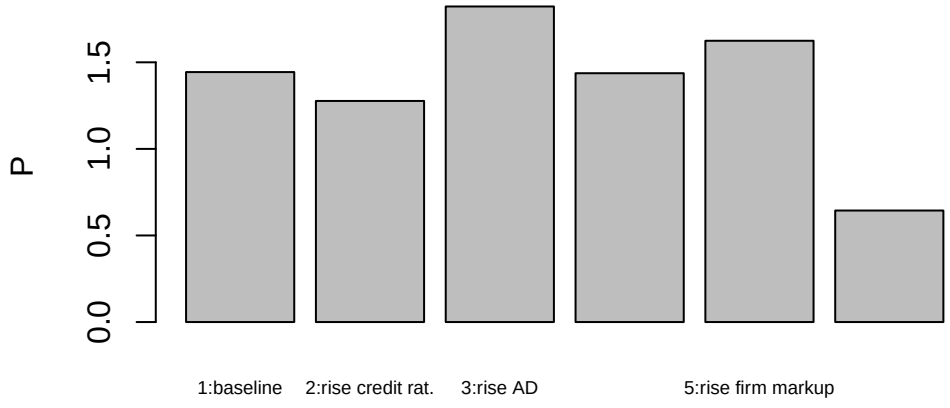


Figure 6.2: Price level

```
barplot(r_star, ylab="r", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD",
                                      "4:rise bank markup","5:rise firm markup", "6:rise proc
```

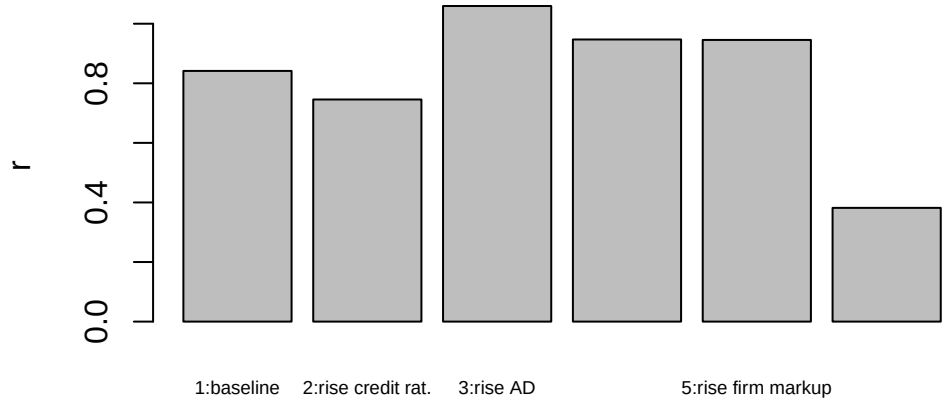


Figure 6.3: Lending rate

```
barplot(dM_star, ylab="dM", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD",
                                        "4:rise bank markup","5:rise firm markup", "6:rise pr
```

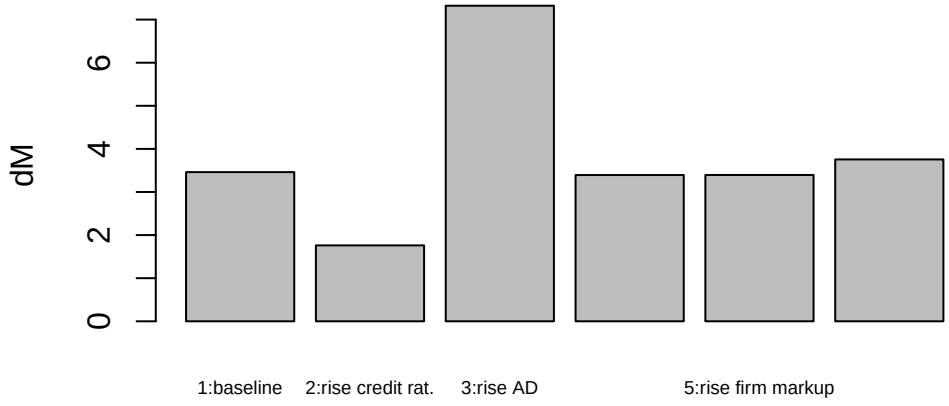


Figure 6.4: Deposit money creation

```
barplot(U_star*100, ylab="U (%)", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD"
                                              "4:rise bank markup","5:rise firm markup", "6:r
```

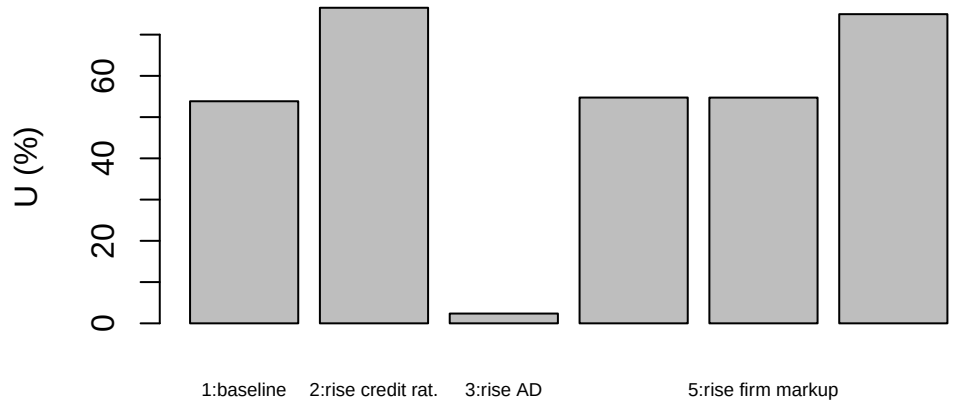


Figure 6.5: Unemployment rate

```
barplot(W_star, ylab="W", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD",
                                      "4:rise bank markup","5:rise firm markup", "6:rise proc
```

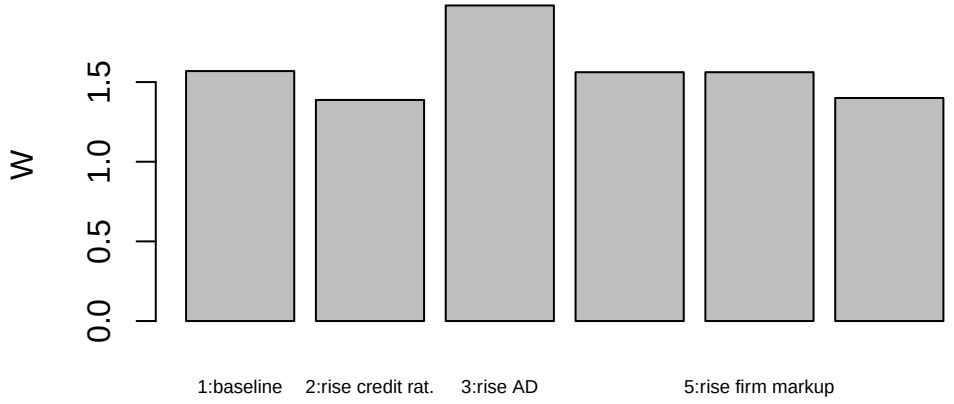


Figure 6.6: Nominal wage

```r
barplot(w_star, ylab="w", names.arg=c("1:baseline", "2:rise credit rat.", "3:rise AD",
                                      "4:rise bank markup","5:rise firm markup", "6:rise proc
```

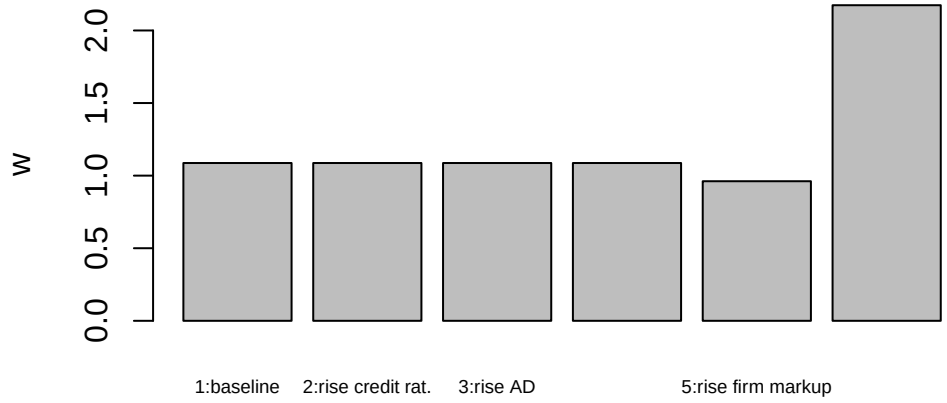


Figure 6.7: Real wage

Python code

```python
# Plot results (here for output only)
import matplotlib.pyplot as plt

scenario_names = ["1:baseline", "2:rise credit rat.", "3:rise AD", "4:rise bank markup", "5

# Create a bar plot
plt.bar(scenario_names, Y_star)
plt.ylabel("Y")
plt.xticks(rotation=45, ha="right")  # Rotate x-axis labels for better readability
plt.tight_layout()  # Ensure the labels fit within the plot area

# Show the plot
plt.show()
```

6.4 Directed graph

```
## Create directed graph
# Construct auxiliary Jacobian matrix for 18 variables:
# r, Y, ND, D, i, P, W, w, N, U, dl, dM, dR, dO, c, m, a, n

M_mat=matrix(c(0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,
               0,0,1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,
               0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,
               0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,
               0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,
               0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,
               0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,
               0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0), 18, 18, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg= graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c("r", "Y", "ND", "D", "i", "P", "W", "w", "N", "U", "dL", "dM", "dR", expression

# Plot directed graph
plot(dg, main="", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

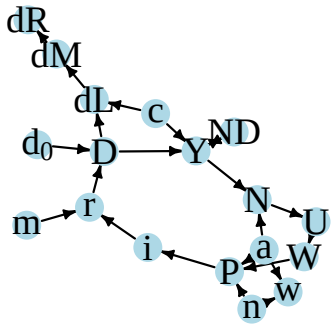


Figure 6.8: Directed graph of post-Keynesian endogenous money model

Python code

```
#Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# Construct auxiliary Jacobian matrix for 18 variables
#    r  Y ND  D  i  P  W  w  N  U  dL dM dR d0 c  m  a  n

M_mat = np.array([[0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0],
                  [0,0,1,1,0,0,0,0,0,0,0,0,0,1,0,0,0],
                  [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                  [1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0],
                  [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1],
                  [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                  [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],
                  [0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],
                  [0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]])


# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {0: "r", 1: "Y", 2: "ND", 3: "D", 4: "i", 5:"P", 6: "W", 7: "w",
            8: "N", 9: "U", 10: "dL", 11: "dM", 12: "dR", 13: "d0", 14: "c",
            15: "m", 16: "a", 17: "n"}

# Plot the directed graph
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=300, node_color='lightblue',
        font_size=10)
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.axis('off')
plt.show()
```

6.5 Analytical discussion

6.5.1 Confirm analytical solutions numerically

```
### Confirm equilibrium solution for Y*  (baseline)
# Analytical solution
(c[1]*(d0[1] -d1*(1+m[1])*(i0+i1*(1+n[1])*a[1]*(W0-h))))/(1-b +c[1]*d1*(1+m[1])*i1*(1+n[1])*
```

```
# Numerical solution
Y_star[1]
```

```
### Confirm equilibrium solution for P*  (baseline)
# Analytical solution
((1+n[1])*a[1]*((W0-h)*(1-b)+(Nf^-1)*h*a[1]*c[1]*(d0[1]-d1*(1+m[1])*i0[1])))/(1-b +c[1]*d1*(
```

```
# Numerical solution
P_star[1]
```

References

7 A Kaldor-Robinson Distribution and Growth Model

7.1 Overview

[1]

[2]

7.2 The Model

[1] See Hein (2014), chap. 4 for a detailed treatment.
[2] All variables are normalised by the capital stock and thus rendered stationary.

7.3 Simulation

7.3.1 Parameterisation

7.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

# Set number of scenarios (including baselines)
S=3

#Create vector in which equilibrium solutions from different parameterisations will be stored
h_star=vector(length=S) # profit share
g_star=vector(length=S) # growth rate of capital stock
s_star=vector(length=S) # saving rate
c_star=vector(length=S) # consumption rate
r_star=vector(length=S) # profit rate

# Set constant parameter values
v=3    # capital-to-potential output ratio
g1=0.3 # sensitivity of investment with respect to profit rate
un=0.9 # normal rate of capacity utilisation

# Set exogenous variables whose parameterisation changes across regimes
g0=vector(length=S) # animal spirits
sp=vector(length=S) # propensity to save out of profits

### Construct different scenarios

# scenario 1: baseline
g0[]=0.02
sp[]=0.6

#scenario 2: increase in animal spirits
g0[2]=0.04

# scenario 3: increase in propensity to save out of profits
sp[3]=0.9
```

```
#Check  stability condition for all scenarios
for (i in 1:S){
  print(sp[i]>g1)
}
```

```
# Initialise endogenous variables at some arbitrary positive value
g = r = s = c = h = 1

#Solve this system numerically through 1000 iterations based on the initialisation
for (i in 1:S){

  for (iterations in 1:1000){

    #(1) Profit rate
    r=(h*un)/v

    #(2) Saving
    s = sp[i]*r

    #(3) Consumption
    c= un/v - s

    #(4) Investment
    g = g0[i]+g1*r

    #(5) Goods market equilibrium profit share
    h=(v/un)*(g0[i]/(sp[i]-g1))

  }

  #Save results for different parameterisations in vector
  h_star[i]=h
  g_star[i]=g
  r_star[i]=r
  s_star[i]=s
  c_star[i]=c
}
```

Python code

```
import numpy as np

# Clear the environment (not necessary in Python)
# Set number of scenarios (including baselines)
S = 3

# Create arrays to store equilibrium solutions for different parameterizations
h_star = np.empty(S)  # profit share
g_star = np.empty(S)  # growth rate of capital stock
s_star = np.empty(S)  # saving rate
c_star = np.empty(S)  # consumption rate
r_star = np.empty(S)  # profit rate

# Set constant parameter values
v = 3    # capital-to-potential output ratio
g1 = 0.3  # sensitivity of investment with respect to profit rate
un = 0.9  # normal rate of capacity utilization

# Set exogenous variables whose parameterization changes across regimes
g0 = np.empty(S)  # animal spirits
sp = np.empty(S)  # propensity to save out of profits

# Construct different scenarios
# Scenario 1: baseline
g0[:] = 0.02
sp[:] = 0.6

# Scenario 2: increase in animal spirits
g0[1] = 0.04

# Scenario 3: increase in propensity to save out of profits
sp[2] = 0.9

# Check stability condition for all scenarios
for i in range(S):
    print(sp[i] > g1)



# Initialize endogenous variables at some arbitrary positive value
g = r = s = c = h = 1

# Solve this system numerically through 1000 iterations based on the initialization
for i in range(S):
    for iterations in range(1000):
        # (1) Profit rate
        r = (h * un) / v

        # (2) Saving
        s = sp[i] * r

        # (3) Consumption
        c = un / v - s
```

7.3.3 Plots

```
barplot(h_star, ylab="h", names.arg=c("1: baseline", "2: rise animal spirits", "3:rise savin
```

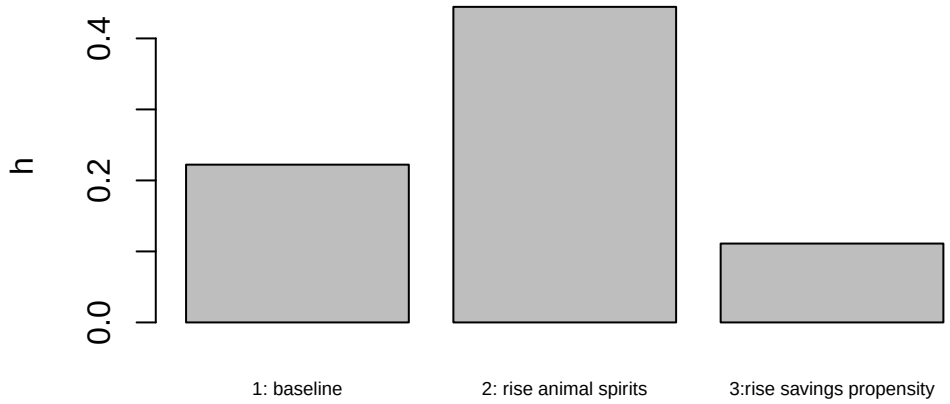


Figure 7.1: Profit share

```
barplot(c_star, ylab="c", names.arg=c("1: baseline", "2: rise animal spirits", "3:rise savin
```

Figure 7.2: Rate of consumption

```
barplot(g_star, ylab="g", names.arg=c("1: baseline", "2: rise animal spirits", "3:rise savin
```

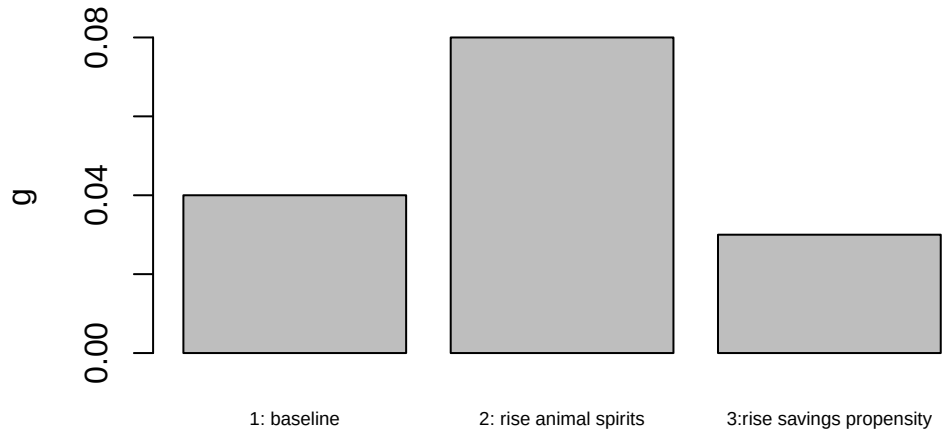


Figure 7.3: Rate of growth

Python code

```python
# Plot results (here only for profit share)
import matplotlib.pyplot as plt

# Scenario labels
scenario_names = ["1: baseline", "2: rise animal spirits", "3: rise savings propensity"]

# Bar plot for h_star
plt.bar(scenario_names, h_star)
plt.ylabel('h')
plt.xticks(scenario_names, rotation=45, fontsize=6)
plt.show()
```

7.4 Directed graph

```r
## Create directed graph
# Construct auxiliary Jacobian matrix for 7 variables:
             # r,h,s,g,g0,sp,un
M_mat=matrix(c(0,1,0,0,0, 0, 1,
               0,0,1,1,0, 0, 1,
               1,0,0,0,0, 1, 0,
               1,0,0,0,1, 0, 0,
               0,0,0,0,0, 0, 0,
               0,0,0,0,0, 0, 0,
               0,0,0,0,0, 0, 0), 7, 7, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg= graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
```

```r
V(dg)$name=c("r", "h", "s", "g", expression(g[0]), expression(s[Pi]), expression(u[n]))

# Plot directed graph
plot(dg, main="", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

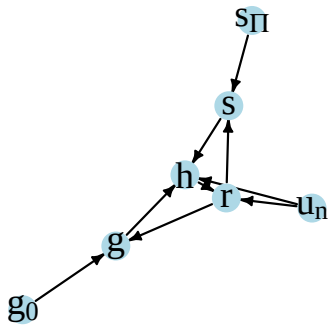


Figure 7.4: Directed graph of Kaldor-Robinson growth model

Python code

```
# Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np


# Define the Jacobian matrix
M_mat = np.array([[0, 1, 0, 0, 0, 0, 1],
                  [0, 0, 1, 1, 0, 0, 1],
                  [1, 0, 0, 0, 0, 1, 0],
                  [1, 0, 0, 0, 1, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0]])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {0: "r", 1: "h", 2: "s", 3: "g", 4: r"$g_0$", 5: r"$s_p$", 6: r"$u_n$"}

# Plot the directed graph
pos = nx.spring_layout(G, seed=43)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=300, node_color='lightblue',
        font_size=10)
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.axis('off')
plt.show()
```

7.5 Analytical discussion

7.5.1 Calculate analytical solutions numerically

```
# Profit rate
for (i in 1:S){
  print((g0[i])/(sp[i]-g1))
}
```

```
# Growth rate
for (i in 1:S){
  print((sp[i]*g0[i])/(sp[i]-g1))
}
```

Python code

```
# Profit rate
for i in range(S):
    print(g0[i] / (sp[i] - g1))

# Growth rate
for i in range(S):
    print((sp[i] * g0[i]) / (sp[i] - g1))
```

References

8 A Post-Kaleckian Distribution and Growth Model

8.1 Overview

[1]

[2]

8.2 The Model

[1] See Hein (2014), chap. 7 and Lavoie (2014), chap. 6 for detailed treatments.
[2] All variables are normalised by the capital stock and thus rendered stationary.

3

[3]Bhaduri and Marglin (1990) further discuss the effects on the profit rate.

8.3 Simulation

8.3.1 Parameterisation

8.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

# Set number of scenarios (including baselines)
S=6

#Create vector in which equilibrium solutions from different parameterisations will be stored
u_star=vector(length=S) # utilisation rate
g_star=vector(length=S) # growth rate of capital stock
s_star=vector(length=S) # saving rate
c_star=vector(length=S) # consumption rate
r_star=vector(length=S) # profit rate

# Set exogenous variables whose parameterisation changes across regimes
g0=vector(length=S) # animal spirits
sw=vector(length=S) # propensity to save out of wages
h=vector(length=S)  # profit share
g1=vector(length=S) # sensitivity of investment with respect to utilisation
```

```
### Construct different scenarios across 3 regimes: (1) WLD/WLG, (2) WLD/PLG, (3) PLD/PLG

# baseline WLD/WLG
g0[1]=0.02
g1[1]=0.1
h[1]=0.2

# increase in profit share in WLD/WLG regime
g0[2]=0.02
g1[2]=0.1
h[2]=0.3

# baseline WLD/PLG
g0[3]=0.02
g1[3]=0.08
h[3]=0.2

# increase in profit share in WLD/PLG regime
g0[4]=0.02
g1[4]=0.08
h[4]=0.3

# baseline PLD/PLG
g0[5]=-0.01
g1[5]=0.1
h[5]=0.2

# increase in profit share in PLD/PLG regime
g0[6]=-0.01
g1[6]=0.1
h[6]=0.3

#Set constant parameter values
v=3    # capital-to-potential output ratio
g2=0.1 # sensitivity of investment with respect to profit share
sp=0.9 # propensity to save out of profits
sw=0.3 # propensity to save out of wages

#Check Keynesian stability condition for all scenarios
for (i in 1:S){
print(((sw+(sp-sw)*h[i])*(1/v) -g1[i])>0)
}
```

```
# Check demand and growth regime for 3 baseline scenarios
for (i in c(1,3,5)){
print(paste("Parameterisation", i, "yields:"))
if(g2*(sw/v - g1[i])-g0[i]*(sp-sw)/v<0){
  print("wage-led demand regime")
  } else{
   print("profit-led demand regime")
  }
if(g1[i]*(g2*(sw/v - g1[i])-g0[i]*(sp-sw)/v)+g2*(((sw+(sp-sw)*h[i])*v^(-1)-g1[i])^2)<0){
  print("wage-led growth regime")
  } else{
  print("profit-led growth regime")
  }
}
```

```
# Initialise endogenous variables at some arbitrary positive value
g=1
r=1
c=1
u=1
s=1

#Solve this system numerically through 1000 iterations based on the initialisation
for (i in 1:S){
```

```
  for (iterations in 1:1000){

    #(1) Profit rate
    r = (h[i]*u)/v

    #(2) Saving
    s = (sw+(sp-sw)*h[i])*(u/v)

    #(3) Consumption
    c= u/v-s

    #(4) Investment
    g = g0[i]+g1[i]*u+g2*h[i]

    #(5) Rate of capacity utilisation
    u = v*(c+g)
  }

  #Save results for different parameterisations in vector
  u_star[i]=u
  g_star[i]=g
  r_star[i]=r
  s_star[i]=s
  c_star[i]=c
}
```

Python code

```
import numpy as np

# Set number of scenarios (including baselines)
S = 6

# Create arrays to store equilibrium solutions for different parameterizations
u_star = np.zeros(S)
g_star = np.zeros(S)
s_star = np.zeros(S)
c_star = np.zeros(S)
r_star = np.zeros(S)

# Set exogenous variables whose parameterization changes across regimes
g0 = np.zeros(S)
sw = np.zeros(S)
h = np.zeros(S)
g1 = np.zeros(S)

# Construct different scenarios across 3 regimes
# Regime 1: WLD/WLG
g0[0] = 0.02
g1[0] = 0.1
h[0] = 0.2

# Regime 2: Increase in profit share in WLD/WLG regime
g0[1] = 0.02
g1[1] = 0.1
h[1] = 0.3

# Regime 3: WLD/PLG
g0[2] = 0.02
g1[2] = 0.08
h[2] = 0.2

# Regime 4: Increase in profit share in WLD/PLG regime
```

8.3.3 Plots

[4]

```
barplot(u_star, ylab="u", names.arg=c("1a:baseline WLD/WLG", "1b:rise prof share", "2a:basel
```

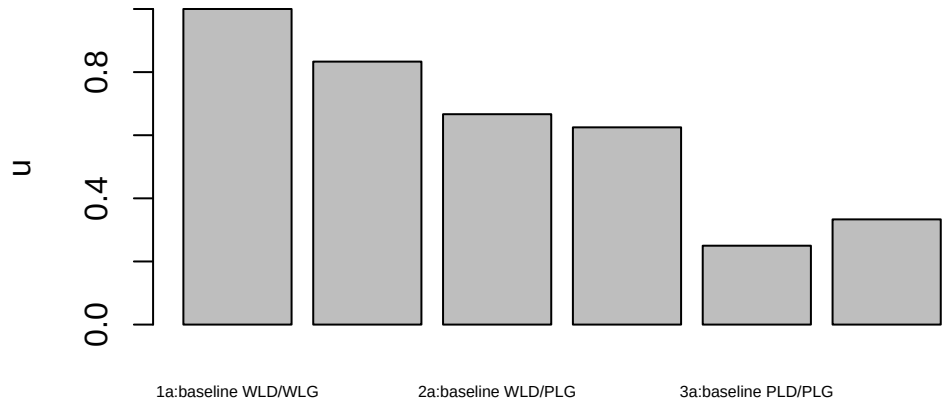


Figure 8.1: Rate of capacity utilisation

```
barplot(g_star, ylab="g", names.arg=c("1:baseline WLD/WLG", "2:rise prof share", "3:baseline
                                      "4:rise prof share",
                                      "5:baseline PLD/PLG", "6: rise prof share"), cex.names
```

[4] If the negative effect on the rate of capacity utilisation was stronger, the profit rate could fall as well. See the analytical discussion for a formal derivation of the condition under which this may happen.

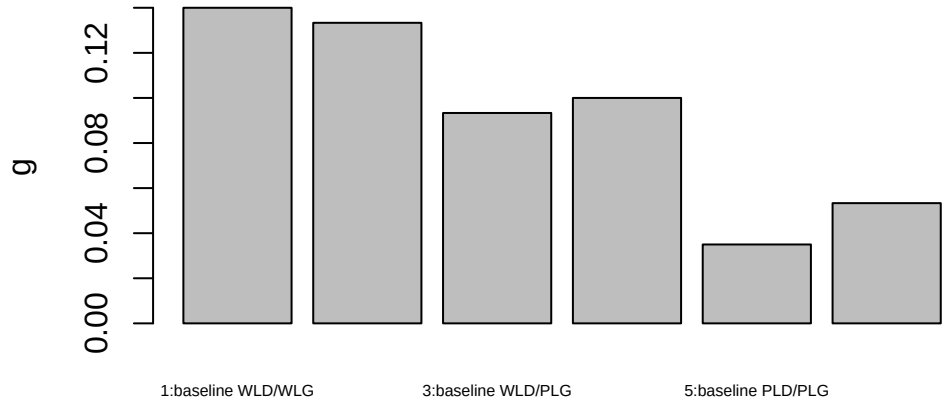


Figure 8.2: Rate of growth

```
barplot(c_star, ylab="c", names.arg=c("1:baseline WLD/WLG", "2:rise prof share", "3:baseline
                                      "4:rise prof share",
                                      "5:baseline PLD/PLG", "6: rise prof share"), cex.names
```

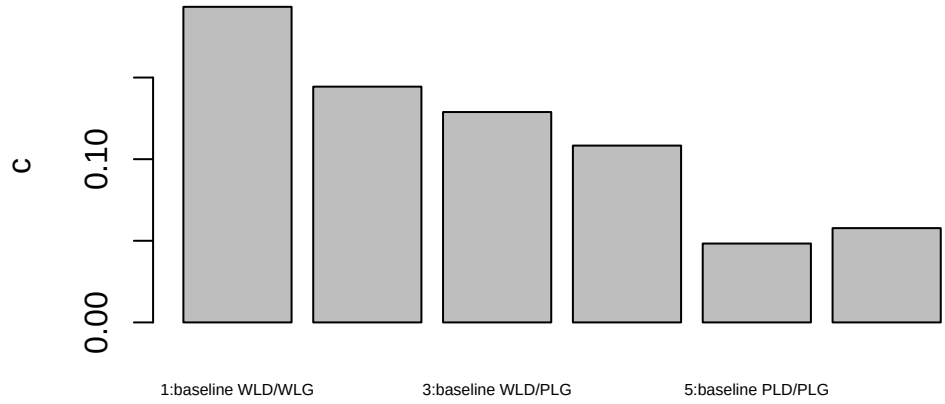


Figure 8.3: Rate of consumption

```
barplot(r_star, ylab="r", names.arg=c("1:baseline WLD/WLG", "2:rise prof share", "3:baseline
                                      "4:rise prof share",
                                      "5:baseline PLD/PLG", "6: rise prof share"), cex.names =
```

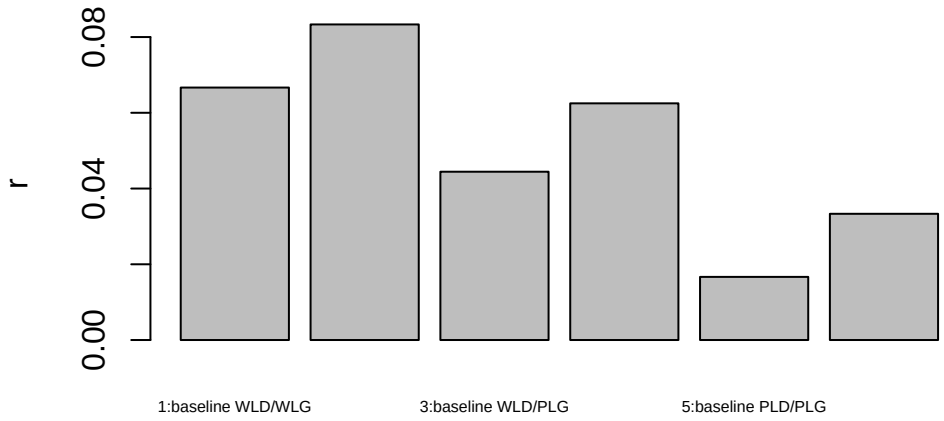


Figure 8.4: Rate of profit

Python code

```
# Plot results (here only for rate of capacity utilisation)
import matplotlib.pyplot as plt

# Scenario labels
scenario_names = ["1a: baseline WLD/WLG", "1b: rise prof share", "2a: baseline WLD/PLG", "2

# Bar plot for u_star
plt.bar(scenario_names, u_star)
plt.ylabel('u')
plt.xticks(scenario_names, rotation=45, fontsize=6)
plt.show()
```

8.4 Directed graph

```
## Create directed graph
# Construct auxiliary Jacobian matrix for 6 variables:
# r, h, u, s, c, g

M_mat=matrix(c(0,1,1,0,0,0,
               0,0,0,0,0,0,
               0,0,0,0,1,1,
               0,1,1,0,0,0,
               0,0,1,1,0,0,
               0,1,1,0,0,0), 6, 6, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian
A_mat=t(M_mat)

# Create and plot directed graph from adjacency matrix
library(igraph)
dg= graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c("r", "h", "u", "s", "c", "g")

# Plot directed graph
plot(dg, main="", vertex.size=20, vertex.color="lightblue",
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

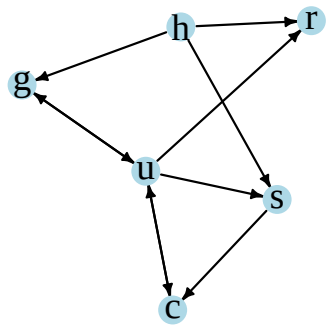


Figure 8.5: Directed graph of post-Kaleckian growth model

Python code

```
# Load relevant libraries
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# Define the Jacobian matrix
M_mat = np.array([[0,1,1,0,0,0],
                  [0,0,0,0,0,0],
                  [0,0,0,0,1,1],
                  [0,1,1,0,0,0],
                  [0,0,1,1,0,0],
                  [0,1,1,0,0,0],
                 ])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {0: "r", 1: "h", 2: "u", 3: "s", 4: "c", 5: "g"}

# Plot the directed graph
pos = nx.spring_layout(G, seed=43)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=300, node_color='lightblue',
        font_size=10)
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.axis('off')
plt.show()
```

[5]

[5]Other important exogenous variables or parameters that may shift but are not depicted here are animal spirits (g_0) or the saving propensities (s_W, s_Π). See Hein (2014), chap. 7.2.2, for a detailed discussion of their effects.

8.5 Analytical discussion

8.5.1 Calculate analytical solutions numerically

```
# Utilisation rate
for (i in 1:S){
print((g0[i]+g2*h[i])/((sw+(sp-sw)*h[i])/v-g1[i]))
}
```

```
# Growth rate
for (i in 1:S){
  print(((g0[i]+g2*h[i])*(sw+(sp-sw)*h[i])/v)/((sw+(sp-sw)*h[i])/v-g1[i]))
}
```

```
# Profit rate
for (i in 1:S){
  print((g0[i]+g2*h[i])*(h[i]/v)/((sw+(sp-sw)*h[i])/v-g1[i]))
}
```

Python code

```
# Utilisation rate
for i in range(S):
    print((g0[i]+g2*h[i])/((sw+(sp-sw)*h[i])/v-g1[i]))

# Growth rate
for i in range(S):
    print(((g0[i]+g2*h[i])*(sw+(sp-sw)*h[i])/v)/((sw+(sp-sw)*h[i])/v-g1[i]))

# Profit rate
for i in range(S):
    print((g0[i]+g2*h[i])*(h[i]/v)/((sw+(sp-sw)*h[i])/v-g1[i]))
```

References

Part II

Dynamic Models

9 An Introduction to the Analysis of Dynamic Models

9.1 Solution of a single first-order linear difference equation

[1]

[1]We will focus here on difference instead of differential equations, i.e. on dynamics in discrete as opposed to continuous time. Most of the continuous-time counterpart is analogous to the material covered here. Sayama (2015) provides a very accessible and applied introduction to dynamic systems with Python code. An introductory treatment of the underlying mathematics is Chiang and Wainwright (2005), chaps. 15-19. Gandolfo (2009) provides a more advanced treatment of the mathematics as well as many economic examples. A great introduction to linear algebra is Anthony and Harvey (2012).

1.
2.

-
-
-
-

9.2 Solution of a linear system of difference equations

[2]

[2]This is because in the product $(PDP^{-1})(PDP^{-1})(PDP^{-1})$..., each P cancels a P^{-1}, except for the first P and last P^{-1}.

```r
#Clear the environment
rm(list=ls(all=TRUE))

## Find eigenvalues and eigenvectors of matrix
# Define matrix
J=matrix(c(7, -15,
           2, -4), 2, 2, byrow=TRUE)

# Obtain eigenvalues and eigenvectors
ev=eigen(J)
(evals = ev$values)
```

```r
(evecs = ev$vector)
```

```r
# Normalise eigenvectors by dividing through by the first element
evecs_norm=evecs
for (i in 1:2){
  evecs_norm[,i]=evecs[,i]/evecs[1,i]
}
evecs_norm
```

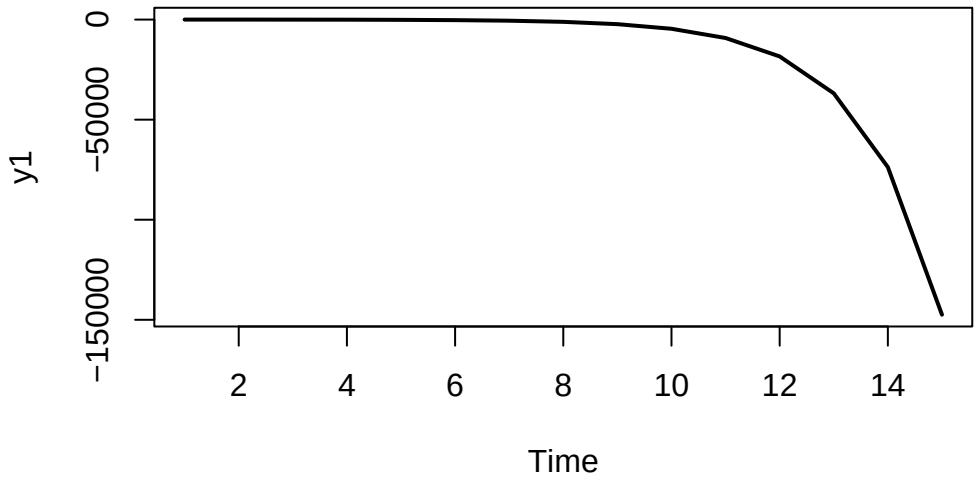
Python code

```
import numpy as np

# Define matrix
J = np.array([[7, -15],
              [2, -4]])

# Obtain eigenvalues and eigenvectors
evals, evecs = np.linalg.eig(J)

# Print eigenvalues and eigenvectors
print(evals)
print(evecs)

# Initialize an array to store the normalized eigenvectors
evecs_norm = np.copy(evecs)

# Normalize the eigenvectors
for i in range(2):
    evecs_norm[:, i] = evecs[:, i] / evecs[0, i]

# Print normalized eigenvectors
print(evecs_norm)
```

-
-
-

L'Hopital's rule

```
# Set number of periods for which you want to simulate
Q=100

# Construct matrices in which values for different periods will be stored; initialise at 1
y1=matrix(data=1, nrow=1, ncol=Q)
y2=matrix(data=1, nrow=1, ncol=Q)

#Solve this system recursively based on the initialisation
  for (t in 2:Q){
    y1[,t] = J[1,1]*y1[, t-1] + J[1,2]*y2[, t-1]
    y2[,t] = J[2,1]*y1[, t-1] + J[2,2]*y2[, t-1]
} # close time loop

# Plot dynamics of y1
plot(y1[1, 1:15],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="y1")
title(main="", cex=0.8)
```

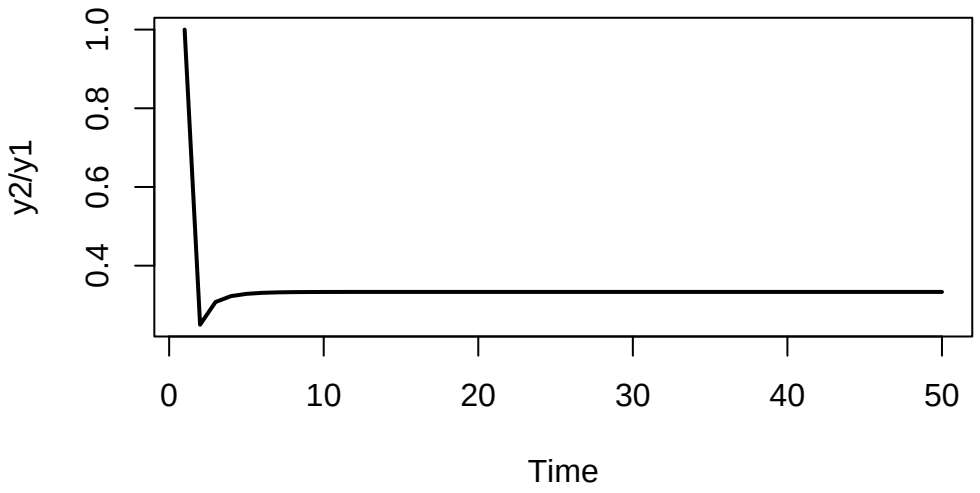



```
# Find arbitrary constants: c=(P^-1)*y0
library(matlib)
y0=c(y1[1,1],y2[1,1])  # create vector with initial conditions y0
c=inv(evecs_norm)%*%y0
c
```

```
## Compute solution manually for y2 at t=10 and compare with simulated solution
t=10
evecs_norm[2,1]*c[1,1]*evals[1]^t + evecs_norm[2,1]*c[2,1]*evals[2]^t # analytical solution
```

```
y2[,t+1] # simulated solution
```

```
# Plot dynamics of y2/y1
y2_y1=y2/y1
plot(y2_y1[, 1:50],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="y2/y1")
title(main="", cex=0.8)
```




```
# Compare y2/y1 with normalised dominant eigenvector
y2_y1[,Q]
```

```
evecs_norm[2,1]
```

Python code

```
import matplotlib.pyplot as plt

# Set the number of periods for simulation
Q = 100

# Initialize arrays to store values for different periods
y1 = np.ones(Q)
y2 = np.ones(Q)

# Solve the system recursively based on the initialization
for t in range(1, Q):
    y1[t] = J[0, 0] * y1[t - 1] + J[0, 1] * y2[t - 1]
    y2[t] = J[1, 0] * y1[t - 1] + J[1, 1] * y2[t - 1]

# Plot dynamics of y1
plt.plot(range(Q), y1, color='b', linewidth=2)
plt.xlabel('Time')
plt.ylabel('y1')
plt.title('Dynamics of y1')
plt.show()


# Define the initial conditions y0
y0 = np.array([y1[0], y2[0]])

# Calculate the arbitrary constants c using the normalized eigenvectors
c = np.linalg.inv(evecs_norm).dot(y0)
c

## Compute solution manually for y2 at t=10 and compare with simulated solution
t = 10 +1
evecs_norm[1, 1] * c[0] * evals[0] ** t + evecs_norm[1, 1] * c[1] * evals[1] ** t
y2[t-1]


# Calculate the ratio y2/y1
y2_y1 = y2 / y1

# Plot dynamics of y2/y1 for the first 50 periods
plt.plot(y2_y1[:50], color='black', linewidth=2, linestyle='-')
plt.xlabel('Time')
plt.ylabel('y2/y1')
plt.show()

# Compare y2/y1 with normalised dominant eigenvector
y2_y1[Q-1]
evecs_norm[1,0]
```

9.3 Complex eigenvalues and cycles

```r
#Clear the environment
rm(list=ls(all=TRUE))

# Set parameter values
c1=0.4
beta=2

# Check if discriminant is negative
(c1*(1+beta))^2-4*c1*beta
```

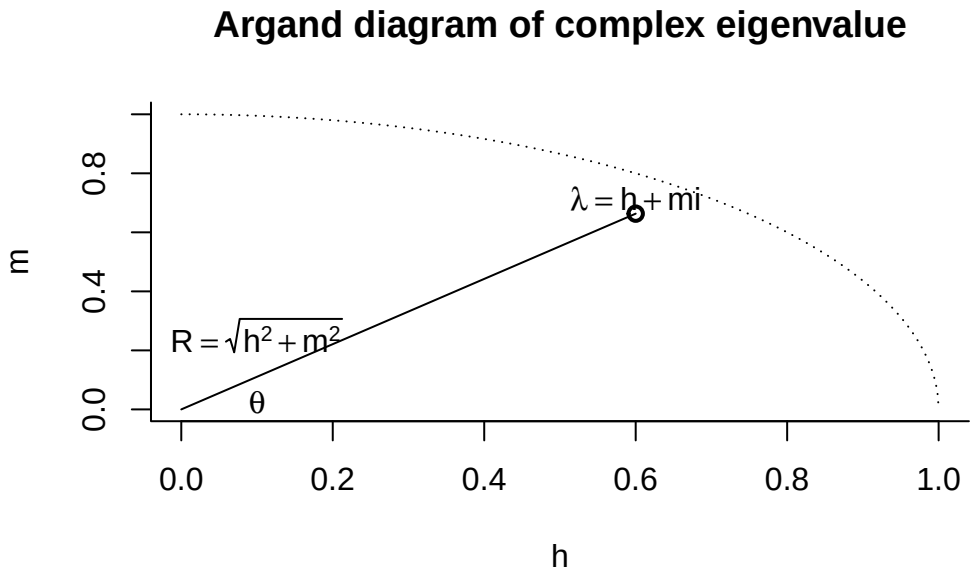
```r
## Find eigenvalues and eigenvectors of matrix
# Define matrix
J=matrix(c(c1, c1,
           beta*(c1-1), beta*c1),
           2, 2, byrow=TRUE)

# Obtain eigenvalues and eigenvectors
ev=eigen(J)
(evals = ev$values)
```

Python code

```
# Set parameter values
c1 = 0.4
beta = 2

# Check if discriminant is negative
(c1 * (1 + beta))**2 - 4 * c1 * beta

# Define the matrix
J = np.array([[c1, c1],
              [beta * (c1 - 1), beta * c1]])

# Calculate eigenvalues and eigenvectors
evals, evecs = np.linalg.eig(J)

print(evals)
print(evals)
```

```
### Draw Argand diagram

# Save real and imaginary part of complex eigenvalue
re=Re(evals[1])
im=Im(evals[1])

# Plot complex eigenvalue
par(bty="l")
plot(re,im, type="o", xlim=c(0, 1), ylim=c(0, 1), lwd=2, xlab="h", ylab="m", main="Argand dia

# Plot unit circle
X=seq(0, 1, by=0.001)
Y = sqrt(1 - X^2)
lines(X,Y, type="l", lty="dotted")
```

```
# Plot a ray from the origin to eigenvalue
segments(0,0,re,im, lty='solid')

# Add labels
text(0.1, 0.025, expression(theta), cex=1)
text(0.1, 0.25, expression(R==sqrt(h^2+m^2)), cex=1)
text(re, im+0.05, expression(lambda==h+mi), cex=1)
```



Python code

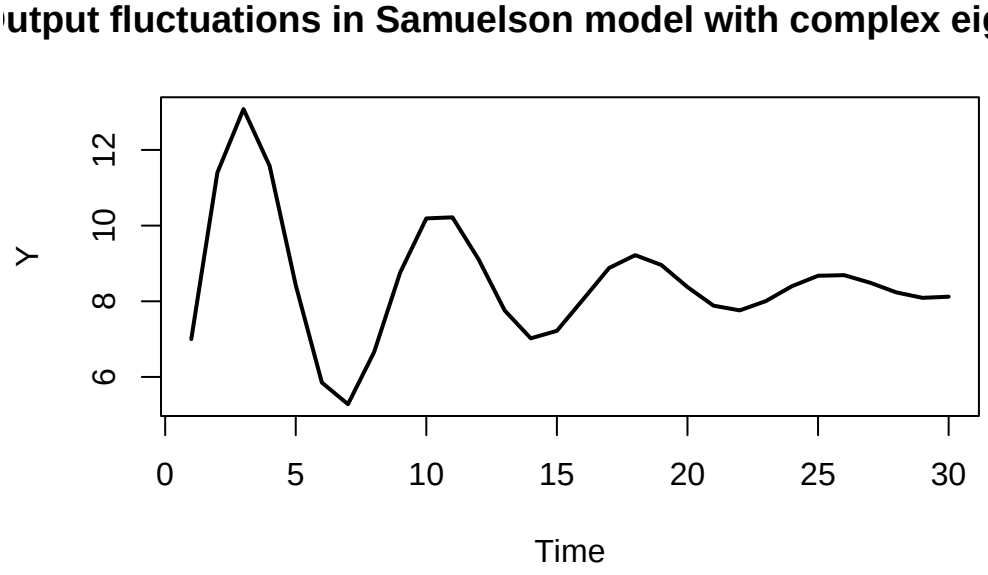
```
### Draw Argand diagram

# Save real and imaginary part of complex eigenvalue
re = evals[0].real
im = evals[0].imag

# Create a figure
fig, ax = plt.subplots()
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_xlabel('h')
ax.set_ylabel('m')
ax.set_title('Argand diagram of complex eigenvalue')

# Plot complex eigenvalue
ax.plot(re, im, 'o', markersize=8, color='k')

# Plot unit circle
X = np.linspace(0, 1, 100)
Y= np.sqrt(1-X**2)
ax.plot(X, Y, 'k--')

# Plot a ray from the origin to the eigenvalue
ax.plot([0, re], [0, im], 'k-')

# Add labels
ax.text(0.1, 0.025, r'$\theta$', fontsize=12)
ax.text(0.001, 0.25, r'$R=\sqrt{h^2+m^2}$', fontsize=12)
ax.text(re, im - 0.1, r'$\lambda=h+mi$', fontsize=12)

plt.show()
```

-
-
-
-

```
# Calculate modulus
mod=Mod(evals[1])
mod
```

```
# Calculate cycle length
L=(2*pi)/(acos(re/mod))
L
```

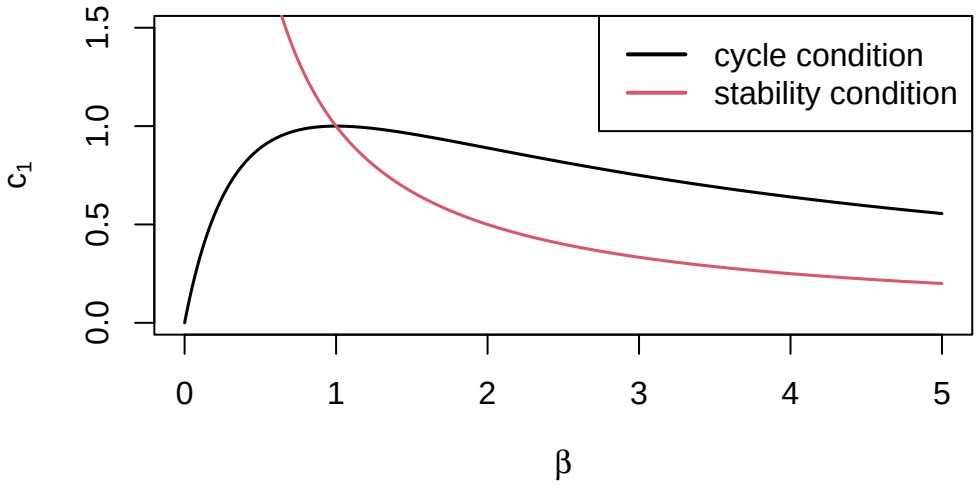
```r
# Set number of periods for which you want to simulate
Q=100

# Set number of parameterisations that will be considered
S=1

# Construct matrices in which values for different periods will be stored; initialise at 1
C=matrix(data=1, nrow=S, ncol=Q)
I=matrix(data=1, nrow=S, ncol=Q)

#Construct matrices for exogenous variable
G0=matrix(data=5, nrow=S, ncol=Q)

#Solve this system recursively based on the initialisation
for (t in 2:Q){
    C[1,t] = c1*(C[1,t-1] + I[1,t-1] + G0[1,t])
    I[1,t] = beta*(c1*(C[1,t-1] + I[1,t-1] + G0[1,t]) - C[1,t-1])
  } # close t1me loop

# Calculate output
Y=C+G0+I

# Time series chart of output dynamics in Samuelson (1939) model
plot(Y[1, 1:30],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="Y")
title(main="Output fluctuations in Samuelson model with complex eigenvalues", cex=0.8)
```



Python code

```
# Calculate modulus
mod = abs(evals[0])
print(mod)

# Calculate cycle length
import math
L = (2 * math.pi) / math.acos(re / mod)
print(L)

# Set the number of periods and parameterizations
Q = 100
S = 1

# Initialize matrices for consumption, investment, and exogenous government spending
C = np.ones((S, Q))
I = np.ones((S, Q))
G0 = np.full((S, Q), 5)

# Solve the system recursively based on the initialization
for t in range(1, Q):
    C[0, t] = c1 * (C[0, t - 1] + I[0, t - 1] + G0[0, t])
    I[0, t] = beta * (c1 * (C[0, t - 1] + I[0, t - 1] + G0[0, t]) - C[0, t - 1])

# Calculate output
Y = C + G0 + I

# Plot the time series chart of output dynamics
plt.plot(Y[0, :30], color='k', linewidth=2, linestyle='-')
plt.xlabel("Time")
plt.ylabel("Y")
plt.title("Output fluctuations in Samuelson model with complex eigenvalues")
plt.show()
```

```r
# Create function for cycle condition: c1 < (4*beta)/(1+beta)^2
cyc= function (beta) {
  (4*beta)/(1+beta)^2
}

# Create function for stability condition: c1 < 1/beta
stab= function (beta) {
  1/beta
}

# Plot the two functions in (beta, c1)-space
curve(cyc, from = 0, to = 5, col = 1, xlab=expression(beta), ylab=expression(c[1]) , main=""
      lwd=1.5, n=10000, ylim=range(0, 1.5))
curve(stab, from = 0, to = 5, col = 2, lwd=1.5, n=10000, add = TRUE)
legend("topright", legend = c("cycle condition", "stability condition"),
       col = c(1, 2), lwd = 2)
```



Python code

```
import numpy as np
import matplotlib.pyplot as plt

# Create function for cycle condition using beta as argument
def cyc(beta):
    return (4 * beta) / (1 + beta)**2

# Create function for stability condition using beta as argument
def stab(beta):
    return 1 / beta

# Define the range of beta values
beta = np.linspace(0.001, 5, 10000)  # start from 0.001 to avoid division by zero

# Plot the two functions in (beta, c1)-space
plt.plot(beta, cyc(beta), label="cycle condition", color='black', linewidth=1.5)
plt.plot(beta, stab(beta), label="stability condition", color='red', linewidth=1.5)

# Set labels and title
plt.xlabel(r'$\beta$')
plt.ylabel(r'$c_1$')
plt.ylim(0, 2)
plt.legend(loc="upper right")

# Display the plot
plt.show()
```

9.4 Nonlinear systems

-
-
-

9.5 Key takeaways

-
-
-
-
  -
  -
-
-

9.6 References

10 A New Keynesian 3-Equation Model

10.1 Overview

[1]

[2]

10.2 The Model

[1] See Galí (2018) for an overview.

[2] Note that this is quite different from conventional New Keynesian dynamic general equilibrium models in which the dynamic element stems from agents with rational expectations that react to serially correlated shocks.

10.3 Simulation

10.3.1 Parameterisation

10.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

# Set number of periods
Q=50

# Set number of scenarios
S=3

# Set period in which shock/shift will occur
s=5

# Create (S x Q)-matrices that will contain the simulated data
y=matrix(data=0,nrow=S,ncol=Q) # Income/output
p=matrix(data=0,nrow=S,ncol=Q) # Inflation rate
r=matrix(data=0,nrow=S,ncol=Q) # Real interest rate
rs=matrix(data=0,nrow=S,ncol=Q) # Stabilising interest rate

# Set constant parameter values
a1=0.3  # Sensitivity of inflation with respect to output gap
a2=0.7  # Sensitivity of output with respect to interest rate
b=1     # Sensitivity of central bank to inflation gap
a3=(a1*(1/(b*a2) + a2))^(-1)

# Set parameter values for different scenarios
A=matrix(data=10,nrow=S,ncol=Q) # autonomous spending
pt=matrix(data=2,nrow=S,ncol=Q) # Inflation target
```

```
ye=matrix(data=5,nrow=S,ncol=Q) # Potential output

A[1,s:Q]=12  # scenario 1: AD boost
pt[2,s:Q]=3  # scenario 2: higher inflation target
ye[3,s:Q]=7  # scenario 3: higher potential output

# Initialise endogenous variables at equilibrium values
y[,1]=ye[,1]
p[,1]=pt[,1]
rs[,1]=(A[,1] - ye[,1])/a1
r[,1]=rs[,1]

# Simulate the model by looping over Q time periods for S different scenarios
for (i in 1:S){

  for (t in 2:Q){

    #(1) IS curve
    y[i,t] = A[i,t] - a1*r[i,t-1]

    #(2) Phillips Curve
    p[i,t] = p[i,t-1] +a2*(y[i,t]-ye[i,t])

    #(3) Stabilising interest rate
    rs[i,t] = (A[i,t] - ye[i,t])/a1

    #(4) Monetary policy rule, solved for r
    r[i,t] = rs[i,t] + a3*(p[i,t]-pt[i,t])

  } # close time loop
}   # close scenarios loop
```

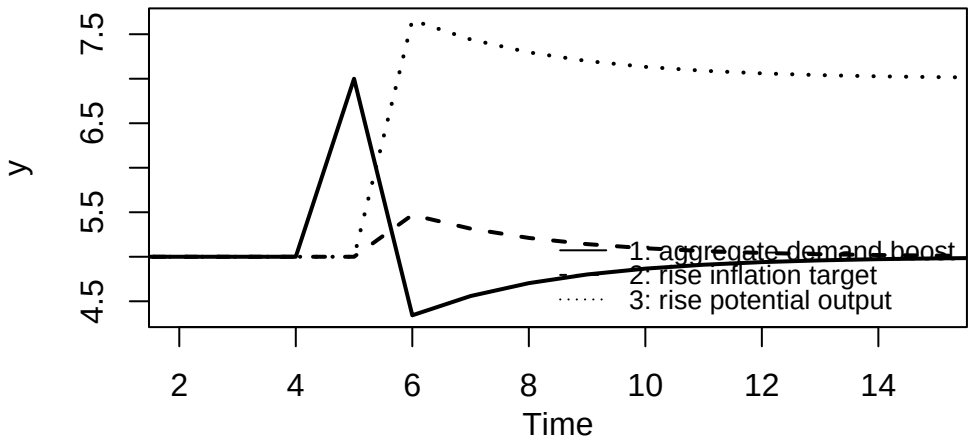
Python code

```
import numpy as np

# Set number of periods
Q = 50

# Set number of scenarios
S = 3

# Set period in which shock/shift will occur
s = 5

# Create (S x Q) arrays to store simulated data
y = np.zeros((S, Q))  # Income/output
p = np.zeros((S, Q))  # Inflation rate
r = np.zeros((S, Q))  # Real interest rate
rs = np.zeros((S, Q))  # Stabilizing interest rate

# Set constant parameter values
a1 = 0.3  # Sensitivity of inflation with respect to output gap
a2 = 0.7  # Sensitivity of output with respect to interest rate
b = 1     # Sensitivity of the central bank to inflation gap
a3 = (a1 * (1 / (b * a2) + a2)) ** (-1)

# Set parameter values for different scenarios
A = np.full((S, Q), 10)  # Autonomous spending
pt = np.full((S, Q), 2)  # Inflation target
ye = np.full((S, Q), 5)  # Potential output

A[0, s:Q] = 12  # Scenario 1: AD boost
pt[1, s:Q] = 3  # Scenario 2: Higher inflation target
ye[2, s:Q] = 7  # Scenario 3: Higher potential output

# Initialize endogenous variables at equilibrium values
y[:, 0] = ye[:, 0]
p[:, 0] = pt[:, 0]
rs[:, 0] = (A[:, 0] - ye[:, 0]) / a1
r[:, 0] = rs[:, 0]

# Simulate the model by looping over Q time periods for S different scenarios
for i in range(S):
    for t in range(1, Q):
        # (1) IS curve
        y[i, t] = A[i, t] - a1 * r[i, t - 1]
        # (2) Phillips Curve
        p[i, t] = p[i, t - 1] + a2 * (y[i, t] - ye[i, t])
        # (3) Stabilizing interest rate
        rs[i, t] = (A[i, t] - ye[i, t]) / a1
        # (4) Monetary policy rule, solved for r
        r[i, t] = rs[i, t] + a3 * (p[i, t] - pt[i, t])
```

10.3.3 Plots

```
### Plot results

### Plots
# Set maximum period for plots
Tmax=15

# Output under different scenarios
plot(y[1, 1:(Tmax+1)],type="l", col=1, lwd=2, lty=1, xlab="", xlim=range(2:(Tmax)), ylab="y"
title(main="Figure 1: Output under different scenarios", xlab = "Time",cex=0.8 ,line=2)
lines(y[2, 1:(Tmax+1)],lty=2, lwd=2)
lines(y[3, 1:(Tmax+1)],lty=3, lwd=2)
legend("bottomright", legend=c("1: aggregate demand boost", "2: rise inflation target", "3: r
```

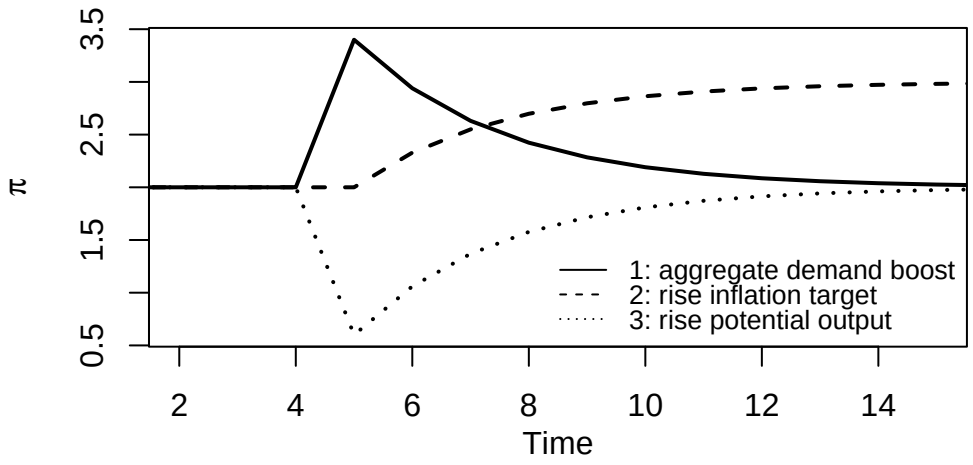
Figure 1: Output under different scenarios




```
# Inflation under different scenarios
plot(p[1, 1:(Tmax+1)],type="l", col=1, lwd=2, lty=1, xlab="", xlim=range(2:(Tmax)), ylab=expr
title(main="Figure 2: Inflation under different scenarios", xlab = "Time",cex=0.8 ,line=2)
lines(p[2, 1:(Tmax+1)],lty=2, lwd=2)
```

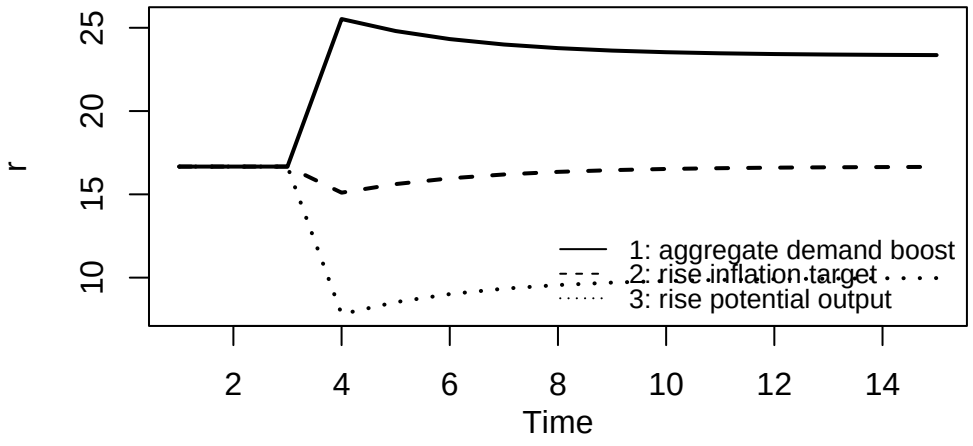
```
lines(p[3, 1:(Tmax+1)],lty=3, lwd=2)
legend("bottomright", legend=c("1: aggregate demand boost", "2: rise inflation target", "3: r
```

Figure 2: Inflation under different scenarios




```
# Policy rate under different scenarios
plot(r[1, 2:(Tmax+1)],type="l", col=1, lwd=2, lty=1, xlab="", xlim=range(1:(Tmax)), ylab="r"
title(main="Figure 3: Policy rate under different scenarios", xlab = "Time",cex=0.8 ,line=2)
lines(r[2, 2:(Tmax+1)],lty=2, lwd=2)
lines(r[3, 2:(Tmax+1)],lty=3, lwd=2)
legend("bottomright", legend=c("1: aggregate demand boost", "2: rise inflation target", "3: r
```

Figure 3: Policy rate under different scenarios



Python code

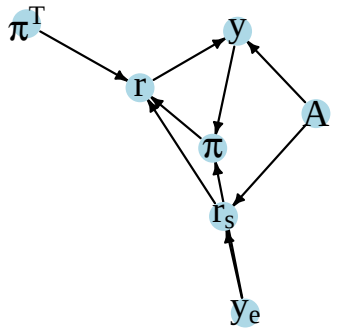
```
import matplotlib.pyplot as plt

# Set maximum period for plots
Tmax = 15

# Plot output under different scenarios
plt.figure(figsize=(8, 6))
plt.plot(y[0, :Tmax + 1], label="Scenario 1: aggregate demand boost",
         color='k', linestyle='solid', linewidth=2)
plt.plot(y[1, :Tmax + 1], label="Scenario 2: Rise inflation target",
         color='k', linestyle='dashed', linewidth=2)
plt.plot(y[2, :Tmax + 1], label="Scenario 3: Rise potential output",
         color='k', linestyle='dotted', linewidth=2)

plt.title("Output under Different Scenarios")
plt.xlabel("Time")
plt.ylabel("y")
plt.xlim(1, Tmax)
plt.ylim(np.min(y), np.max(y))
plt.legend()
plt.show()
```

10.4 Directed graph

```
# Construct auxiliary Jacobian matrix for 7 variables: y, p, r, A, ye, rs, pt
# where non-zero elements in regular Jacobian are set to 1 and zero elements are unchanged
```

```r
M_mat=matrix(c(0,0,1,1,0,0,0,
               1,0,0,0,1,0,0,
               0,1,0,0,0,1,1,
               0,0,0,0,0,0,0,
               0,0,0,0,0,0,0,
               0,0,0,1,1,0,0,
               0,0,0,0,0,0,0),7,7, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg=graph_from_adjacency_matrix(A_mat, mode="directed", weighted=NULL)

# Define node labels
V(dg)$name=c("y", expression(pi), "r", "A", expression(y[e]),expression(r[s]), expression(pi

# Plot directed graph
plot(dg, main="Figure 4: Directed graph of 3-Equation model", vertex.size=20, vertex.color="
     vertex.label.color="black", edge.arrow.size=0.3, edge.width=1.1, edge.size=1.2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1.2,
     vertex.frame.color="NA", margin=-0.08)
```

Figure 4: Directed graph of 3−Equation model



Python code

```
# Directed graph
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np


# Define the Jacobian matrix
M_mat = np.array([[0, 0, 1, 1, 0, 0, 0],
                  [1, 0, 0, 0, 1, 0, 0],
                  [0, 1, 0, 0, 0, 1, 1],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 1, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                 ])

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat = M_mat.transpose()

# Create the graph from the adjacency matrix
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {
    0: "y",
    1: " ",
    2: "r",
    3: "A",
    4: "y ",
    5: "r ",
    6: " "
}

# Plot the directed graph
pos = nx.spring_layout(G, seed=43)
nx.draw(G, pos, with_labels=True, labels=nodelabs, node_size=300, node_color='lightblue',
        font_size=10)
edge_labels = {(u, v): '' for u, v in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='black')
plt.axis('off')
plt.show()
```

10.5 Analytical discussion

10.5.1 Derivation of core equations

10.5.1.1 IS curve

10.5.1.2 PC curve

[3]

[3] See chapter 2 of Carlin and Soskice (2014) for details.

10.5.1.3 Monetary policy rule

[4]

10.5.2 Equilibrium solutions and stability analysis

[4]See chapter 2 of Carlin and Soskice (2014) for details.

[5]

```
# Construct Jacobian matrix
J=matrix(c(0,0,-a1,
           0,1,-a1*a2,
           0,a3,-a1*a2*a3), 3, 3, byrow=TRUE)

# Obtain eigenvalues
ev=eigen(J)
(values <- ev$values)
```

[5] As mentioned in footnote 2, this property of the Carlin-Soskice model is very different from conventional New Keynesian models with rational expectations. In these models, variables such as output and inflation are driven by the 'forward-looking' behaviour of rational agents, i.e. they depend on expectational terms for their current values rather than lagged values. To ensure what is called 'determinancy', these forward-looking variables must adjust fast (or 'jump') to bring the economy back onto a path that is consistent with the optimising equilibrium. This requires the number of jump variables to be matched by an equal number of unstable roots (i.e. being outside the unit circle).

```
# Obtain determinant and trace
det(J)       # determinant
```

```
sum(diag(J)) # trace
```

Python code

```
import numpy as np

# Construct Jacobian matrix
J = np.array([[0, 0, -a1],
              [0, 1, -a1 * a2],
              [0, a3, -a1 * a2 * a3],])

# Calculate eigenvalues
eigenvalues = np.linalg.eigvals(J)

# Print the resulting eigenvalues
print(eigenvalues)

# Calculate the determinant and trace of the Jacobian matrix
determinant = np.linalg.det(J)
trace = np.trace(J)
print(determinant)
print(trace)
```

10.6 References

11 A Conflict Inflation Model

11.1 Overview

[1]

three-sector conflict inflation model [2]

[1] With textbook treatments in Blecker and Setterfield (2019), chap. 5, Hein (2023), chap. 5, and Lavoie (2022), chap. 8.

[2] Wildauer et al. (2023) use this model to study the 2021-23 inflationary episode in the US. Besides an energy sector providing intermediate inputs, they consider a final goods and a final services sector. In addition to the energy price shock, there is a positive demand shift towards goods and a negative supply shock due to lockdowns, creating a rise in the rate of capacity utilisation of the final goods sector. These additional shocks allow the final goods sector to raise its markup. The present simplified version of the model only considers the energy sector and a final output sector, and treats the markup of the final output sector as constant.

11.2 The Model

[3]

[4]

[3]Blecker and Setterfield (2019), chap. 5 and Lavoie (2022), chap. 8 analyse extensions which endogenise the real wage target of workers.

[4]In Wildauer et al. (2023), the target markup of the final goods sector is endogenous with respect to the rate of capacity utilisation.

11.3 Simulation

11.3.1 Parameterisation

11.3.2 Simulation code

```
#Clear the environment
rm(list=ls(all=TRUE))

#Set number of periods
Q = 300

# Set number of scenarios (including baselines)
S=4

# Set period in which exogenous shift will occur
q=50

#Create (S x Q) matrices in which equilibrium solutions from different parameterisations wil
```

```
omega=matrix(data=0.5, nrow=S, ncol=Q)   # real wage
w=matrix(data=0.1, nrow=S, ncol=Q)       # nominal wage
p=matrix(data=0.5, nrow=S, ncol=Q)       # price level
w_hat=matrix(data=0.1, nrow=S, ncol=Q)   # growth rate of nominal wages
p_hat=matrix(data=0.1, nrow=S, ncol=Q)   # growth rate of prices = inflation rate
p_T=matrix(data=0.5, nrow=S, ncol=Q)     # firms' desired price level
omega_f=matrix(data=0.5, nrow=S, ncol=Q) # firms' desired real wage
p_E=matrix(data=0.5, nrow=S, ncol=Q)   # nominal energy price
r_T=matrix(data=0.5, nrow=S, ncol=Q)     # target profit margin
r=matrix(data=0.5, nrow=S, ncol=Q)       # realised profit margin

# Set constant parameter values
phi=1     # adjustment speed of nominal wages
psi=1   # adjustment speed of prices
delta=1   # energy intensity

# Set and initialise exogenous variables/parameters that will be shifted
epsilon=matrix(data=0.55, nrow=S, ncol=Q)  # real energy price
theta_T=matrix(data=0.4, nrow=S, ncol=Q)       # target markup
omega_w=matrix(data=0.2, nrow=S, ncol=Q)   # target real wage of workers

# Set parameter values for different scenarios
epsilon[2,q:Q]=0.8 # scenario 2: rise in real energy price
theta_T[3,q:Q]=0.6     # scenario 3: rise in target markup
omega_w[4,q:Q]=0.3 # scenario 4: rise in target real wage of workers

# Simulate the model by looping over Q time periods for S different scenarios
for (i in 1:S){

  for (t in 2:Q){

    for (iterations in 1:1000){ # iterate the model 1000-times in each period

    #(1) Real wage
    omega[i,t] =w[i,t]/p[i,t]

    #(2) Nominal wage
    w[i,t] = w[i,t-1]*(1 + phi*(omega_w[i,t-1] - omega[i,t-1]))

    #(3) Price level
    p[i,t] = p[i, t-1]*(1 + psi*(r_T[i, t-1] - r[i, t-1]))
```

```
    #(4) Firms' desired real wage
    omega_f[i,t] = w[i,t]/p_T[i,t]

    #(5) Firms' desired price level
    p_T[i,t] = (1 + theta_T[i,t])*(w[i,t] + delta * p_E[i,t])

    #(6) Nominal energy price
    p_E[i,t]=epsilon[i,t]*p[i,t]

    # Target profit margin
    r_T[i, t-1] = theta_T[i, t]/(1 + theta_T[i, t])

    # Realised profit margin
    r[i, t] = 1 - omega[i, t] - epsilon[i,t]*delta

    # Growth rate of nominal wages
    w_hat[i,t]= (w[i,t]- w[i,t-1])/w[i,t-1]

    # Growth rate of prices = inflation rate
    p_hat[i,t]= (p[i,t]- p[i,t-1])/p[i,t-1]

  } # close iterations loop
 }  # close time loop
}   # close scenarios loop
```

Python code

```
import numpy as np

# Set number of periods
Q = 300

# Set number of scenarios (including baseline)
S = 4

# Set period in which exogenous shift will occur
q = 50

# Create (S x Q) matrices in which equilibrium solutions from different parameterizations w
omega = np.full((S, Q), 0.5)  # real wage
w = np.full((S, Q), 0.1)      # nominal wage
p = np.full((S, Q), 0.5)      # price level
w_hat = np.full((S, Q), 0.1)  # growth rate of nominal wages
p_hat = np.full((S, Q), 0.1)  # growth rate of prices = inflation rate
p_T = np.full((S, Q), 0.5)    # firms' desired price level
omega_f = np.full((S, Q), 0.5) # firms' desired real wage
p_E = np.full((S, Q), 0.5)  # nominal energy price
r_T = np.full((S, Q), 0.5)    # target profit margin
r = np.full((S, Q), 0.5)      # realised profit margin

# Set constant parameter values
phi = 1     # adjustment speed of nominal wages
psi = 1   # adjustment speed of prices
delta = 1   # energy intensity

# Set and initialize exogenous variables/parameters that will be shifted
epsilon = np.full((S, Q), 0.55)  # real energy price
theta_T = np.full((S, Q), 0.4)       # target markup
omega_w = np.full((S, Q), 0.2)   # target real wage of workers

# Set parameter values for different scenarios
epsilon[1, q:Q] = 0.8  # scenario 2: rise in real energy price
theta_T[2, q:Q] = 0.6      # scenario 3: rise in target markup
omega_w[3, q:Q] = 0.3  # scenario 4: rise in target real wage of workers

# Simulate the model by looping over Q time periods for S different scenarios
for i in range(S):
    for t in range(1, Q):
        for _ in range(1000):  # iterate the model 1000-times in each period

            # (1) Real wage
            omega[i, t] = w[i, t] / p[i, t]

            # (2) Nominal wage
            w[i, t] = w[i, t-1] * (1 + phi * (omega_w[i, t-1] - omega[i, t-1]))

            # (3) Price level
            p[i, t] = p[i, t-1] * (1 + psi * (r_T[i, t-1] - r[i, t-1]))

            # (4) Firms' desired real wage
```

11.3.3 Plots

```
# Set start and end periods for plots
Tmin =3
Tmax=300

# Prices and wages (baseline)
plot(p[1, Tmin:Tmax],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="p, w")
title(main="Prices and wages (baseline)", cex=0.8)
lines(w[1, Tmin:Tmax],lty=2)
legend("topleft", legend=c("p", "w"),
       lty=1:2, cex=0.8, bty = "n", y.intersp=0.8)
```

Figure 11.1: Prices and wages (baseline)

```
# Set start and end periods for plots
Tmin =3
Tmax=60

# Price and wage inflation (in %) (baseline)
```

```
plot(100*p_hat[1, Tmin:Tmax],type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="%", ylim=rang
title(main="Price and wage inflation (baseline)", cex=0.8)
lines(100*w_hat[1, Tmin:Tmax],lty=2)
legend("topright", legend=c(expression(hat(p)), expression(hat(w))),
       lty=1:2, cex=0.8, bty = "n", y.intersp=0.8)
```

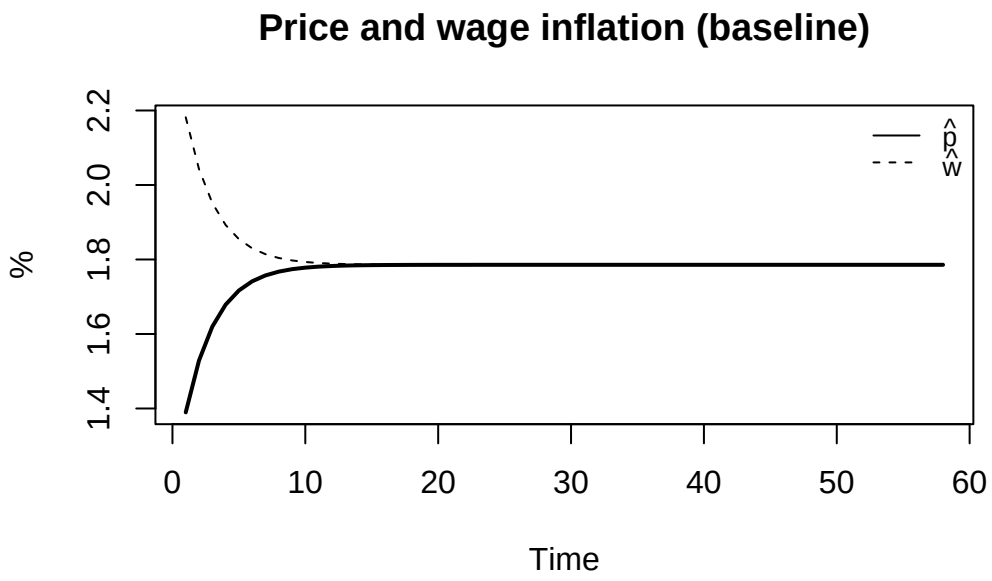


Figure 11.2: Price and wage inflation (baseline)

[5]

```
# Set start and end periods for plots
Tmin =10
Tmax=100

# Inflation and real wage in energy price shock scenario
```

[5] More accurately this is an energy price 'shift'.

```
plot(100*p_hat[2, Tmin:Tmax], type="l", col=1, lwd=2, lty=1, xlab="Time", ylab=expression(hat
title(main="Wage and price inflation and real wage, energy price shock (scen. 2)", cex=0.8)
lines(100*w_hat[2, Tmin:Tmax],lty=2, lwd=2)
mtext(expression(hat(w)), side = 2, line = 3, at=12)
par(mar = c(5, 4, 4, 4) + 0.3)
par(new = TRUE)
plot(omega[2, Tmin:Tmax], type="l", col=1, lwd=2, lty=3, font.main=1, cex.main=1,ylab = '', a
     xlab = '', ylim = range(omega[2, Tmin:Tmax]), cex=0.8)
axis(side = 4, at=pretty(omega[2, Tmin:Tmax]))
mtext(expression(Omega), side = 4, line = 3)
legend("topright", legend=c(expression(hat(p)),expression(hat(w)), expression(Omega)),
       lty=1:3, cex=0.8, bty = "n", y.intersp=0.8)
```

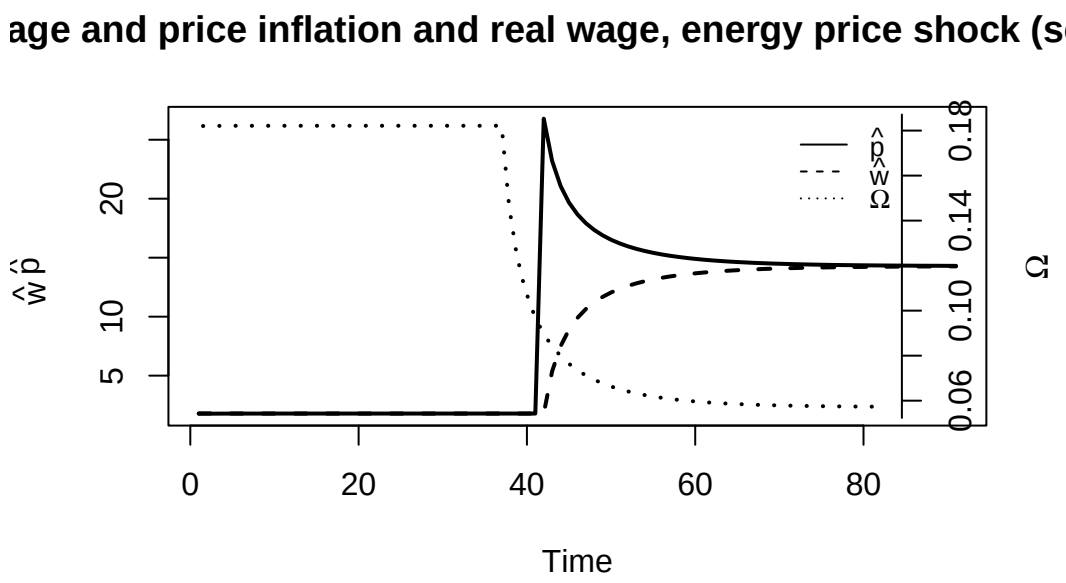


Figure 11.3: Price and wage inflation and real wage (energy price shock)

```
# Set start and end periods for plots
Tmin =30
Tmax=80
```

```
# Inflation and real wage in scenario 3
plot(100*p_hat[3, Tmin:Tmax], type="l", col=1, lwd=2, lty=1, xlab="Time", ylab=expression(hat
title(main="Price and wage inflation and real wage, rise in target markup (scen. 3)", cex=0.8
lines(100*w_hat[3, Tmin:Tmax],lty=2, lwd=2)
mtext(expression(hat(w)), side = 2, line = 3, at=7)
par(mar = c(5, 4, 4, 4) + 0.3)
par(new = TRUE)
plot(omega[3, Tmin:Tmax], type="l", col=1, lwd=2, lty=3, font.main=1, cex.main=1,ylab = '', a
     xlab = '', ylim = range(omega[3, Tmin:Tmax]), cex=0.8)
axis(side = 4, at=pretty(omega[3, Tmin:Tmax]))
mtext(expression(Omega), side = 4, line = 3)
legend("topright", legend=c(expression(hat(p)),expression(hat(w)), expression(Omega)),
       lty=1:3, cex=0.8, bty = "n", y.intersp=0.8)
```

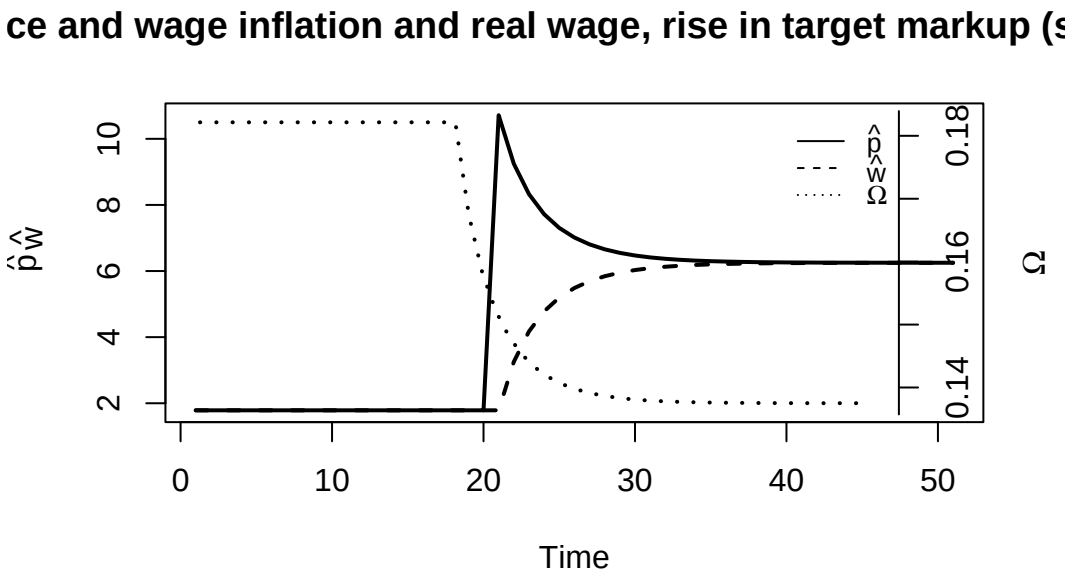


Figure 11.4: Increase in target markup

```r
# Inflation and real wage in scenario 4
plot(100*p_hat[4, Tmin:Tmax], type="l", col=1, lwd=2, lty=1, xlab="Time", ylab=expression(hat
title(main="Price and wage inflation real wage, rise in target real wage (scen. 4)", cex=0.8)
lines(100*w_hat[4, Tmin:Tmax],lty=2, lwd=2)
mtext(expression(hat(w)), side = 2, line = 3, at=7.5)
par(mar = c(5, 4, 4, 4) + 0.3)
par(new = TRUE)
plot(omega[4, Tmin:Tmax], type="l", col=1, lwd=2, lty=3, font.main=1, cex.main=1,ylab = '', a
     xlab = '', ylim = range(omega[4, Tmin:Tmax]), cex=0.8)
axis(side = 4, at=pretty(omega[4, Tmin:Tmax]))
mtext(expression(Omega), side = 4, line = 3)
legend("topright", legend=c(expression(hat(p)),expression(hat(w)), expression(Omega)),
       lty=1:3, cex=0.8, bty = "n", y.intersp=0.8)
```

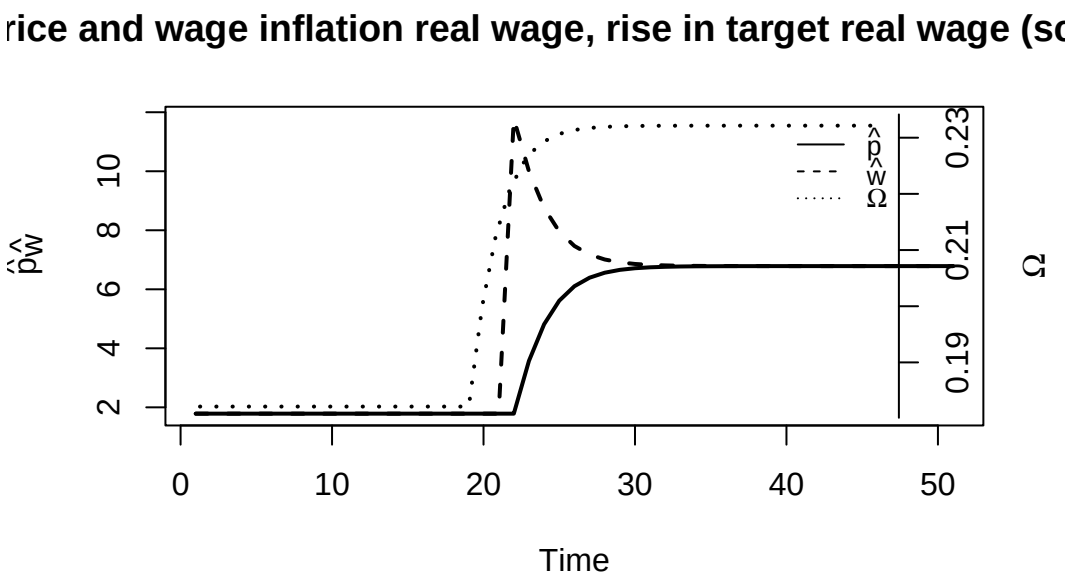


Figure 11.5: Increase in target real wage of workers

Python code

```
## Plot (here only inflation in scenario 2)
import matplotlib.pyplot as plt

# Set start and end periods for plots
Tmin = 9   # Zero-based index (equivalent to Tmin=10 in R)
Tmax = 100

# Set up the main plot
fig, ax1 = plt.subplots()

# Plot inflation rates (price and wage growth rates) on the primary y-axis
ax1.plot(100 * p_hat[1, Tmin:Tmax], label=r'$\hat{p}$', color="black", linewidth=2, linesty
ax1.plot(100 * w_hat[1, Tmin:Tmax], label=r'$\hat{w}$', color="black", linewidth=2, linesty
ax1.set_xlabel("Time")
ax1.set_ylabel(r'$\hat{p}$, $\hat{w}$')
ax1.set_ylim(0, 30)  # Adjust this limit as needed based on your data

# Add secondary y-axis for real wage
ax2 = ax1.twinx()
ax2.plot(omega[1, Tmin:Tmax], label=r'$\Omega$', color="black", linewidth=2, linestyle=":")
ax2.set_ylabel(r'$\Omega$')
ax2.set_ylim(0, 0.2)  # Set range based on omega values

# Titles and legends
plt.title("Wage and price inflation and real wage, energy price shock (scenario 2)", fontsi
ax1.legend(loc="upper left", frameon=False, fontsize=8)
ax2.legend(loc="upper right", frameon=False, fontsize=8)

plt.show()
```

11.4 A closer look at the effect of energy price shocks on profits

[6]

[7]

[6]We derive the exact effects in the analytical discussion below.
[7]See analytical discussion below.

```
# Calculate final output profit share
pi=(p-w-p_E*delta)/(p-p_E*delta)

# Check if psi*r_T > phi*omega_w, implying that the effect of an energy price increase on the
print(psi*r_T[1,1] - phi*omega_w[1,1] >0)
```

```
# Plot profit margin and profit share in energy price shock scenario
plot(r[2, Tmin:Tmax], type="l", col=1, lwd=2, lty=1, xlab="Time", ylab="r,", ylim=range(r[2,
title(main="Profit margin vs profit share, energy price shock (scenario 2)", cex=0.8)
lines(pi[2, Tmin:Tmax],lty=2, lwd=2)
mtext(expression(pi), side = 2, line = 3, at=0.4)
legend("right", legend=c("r", expression(pi)),
       lty=1:2, cex=0.8, bty = "n", y.intersp=0.8)
```

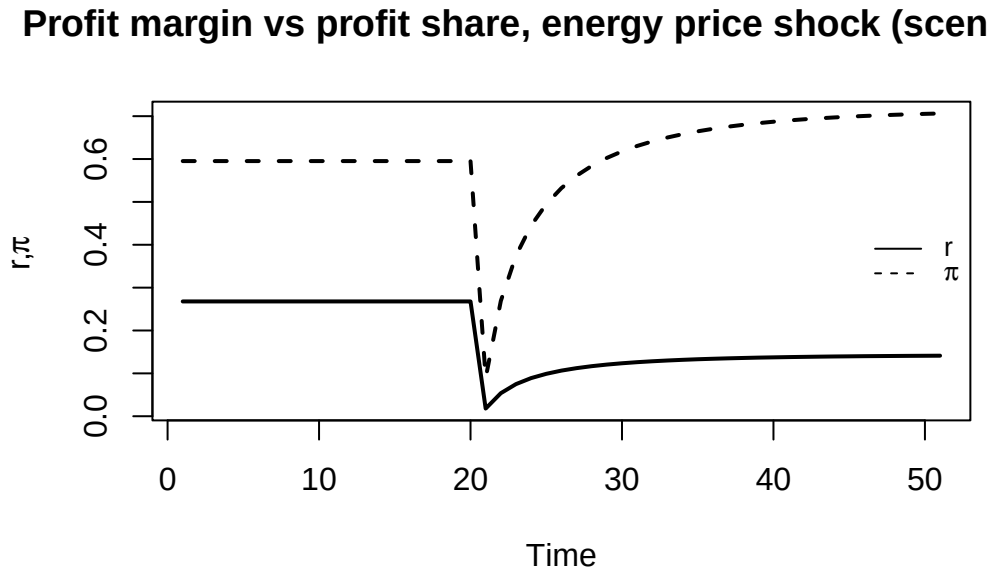
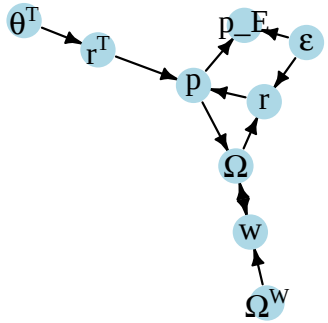


Figure 11.6: Profit margin and profit share (energy price shock)

11.5 Directed graph

```
## Create directed graph
# Construct auxiliary Jacobian matrix for 9 variables:
  # endogenous: (1) Omega, (2) w, (3) p, (4) r^T, (5) r, (6) p_E
  # exogenous: (7) epsilon, (8) \theta^T, (9) Omega^W

              #O w p rF r g e mF OW
M_mat=matrix(c(0,1,1, 0,0,0,0, 0, 0,  # Omega
               1,0,0, 0,0,0,0, 0, 1,  # w
               0,0,0, 1,1,0,0, 0, 0,  # p
               0,0,0, 0,0,0,0, 1, 0,  # rf
               1,0,0, 0,0,0,1, 0, 0,  # r
               0,0,1, 0,0,0,1, 0, 0,  # p_E
               0,0,0, 0,0,0,0, 0, 0,  # epsilon
               0,0,0, 0,0,0,0, 0, 0,  # \theta^T
               0,0,0, 0,0,0,0, 0, 0),  # Omega^W
               9, 9, byrow=TRUE)

# Create adjacency matrix from transpose of auxiliary Jacobian and add column names
A_mat=t(M_mat)

# Create directed graph from adjacency matrix
library(igraph)
dg=graph_from_adjacency_matrix(A_mat, mode="directed", weighted= NULL)

# Define node labels
V(dg)$name=c(expression(Omega), "w", "p", expression(r^T), "r",
             expression(p_E), expression(epsilon), expression(theta^T), expression(Omega^W))


# Plot directed graph matrix
plot(dg, main="Directed graph of conflict inflation model", vertex.size=25, vertex.color="lig
     vertex.label.color="black", edge.arrow.size=0.4, edge.width=1.1, edge.size=2,
     edge.arrow.width=1.2, edge.color="black", vertex.label.cex=1,
     vertex.frame.color="NA", margin=-0.08)
```

Directed graph of conflict inflation model



Python code

```
## Directed graph
import networkx as nx

# Construct auxiliary Jacobian matrix for 9 variables
# endogenous: (1) Omega, (2) w, (3) p, (4) r^T, (5) r, (6) p_E
# exogenous: (7) epsilon, (8) \theta^T, (9) Omega^W
M_mat = np.array([
    [0, 1, 1, 0, 0, 0, 0, 0, 0],  # Omega
    [1, 0, 0, 0, 0, 0, 0, 0, 1],  # w
    [0, 0, 0, 1, 1, 0, 0, 0, 0],  # p
    [0, 0, 0, 0, 0, 0, 0, 1, 0],  # r^T
    [1, 0, 0, 0, 0, 0, 1, 0, 0],  # r
    [0, 0, 1, 0, 0, 0, 1, 0, 0],  # p_E
    [0, 0, 0, 0, 0, 0, 0, 0, 0],  # epsilon
    [0, 0, 0, 0, 0, 0, 0, 0, 0],  # \theta^T
    [0, 0, 0, 0, 0, 0, 0, 0, 0]   # Omega^W
])

# Create adjacency matrix by transposing the auxiliary Jacobian matrix
A_mat = M_mat.T

# Create a directed graph from the adjacency matrix using networkx
G = nx.DiGraph(A_mat)

# Define node labels
nodelabs = {0: r'$\Omega$', 1: "w", 2: "p", 3: r'$r^T$', 4: "r", 5: r'$p^E$',
            6: r'$\epsilon$', 7: r'$\theta^T$', 8: r'$\Omega^W$'}

# Plot the graph
pos = nx.spring_layout(G, k=0.6)
nx.draw_networkx(G, pos, node_size=200, node_color="lightblue",
                 edge_color="black", width=1.2, arrowsize=10,
                 arrowstyle='->', font_size=8, font_color="black",
                 with_labels=True, labels=nodelabs)

plt.title("Directed graph of conflict inflation model", fontsize=12)
plt.show()
```

11.6 Analytical discussion