Economic Models

Python is a very powerfull tool when dealing with economic models. It is widely used in quantitative macroeconomic fields and others. In this section we will learn some applications of Python using models. If you have a special interest it is highly recommendable to visit the website QuantEcon created by Thomas J. Sargent and John Stachurski where a large amount of model applications can be found.

Collusion with Cournot competition

The first model that we will see is a model of collusion with Bertrand competition. Remember in such a model firms compete in quantities. For this example we will be following the notes here.

Let's present the model.

demand and cost

The demand is a linear curve $p_i(x_i, x_j) = 1 - x_i - bx_j$ where $b \in (0, 1]$ determines the elasticity of substitution between the goods. With $b = 1$ the goods are perfect substitutes, with $b < 1$ the goods are differentiated.

Marginal costs are constant with $c(x) = cx$. If we would like to introduce a fixed cost, $c(x) = cx + f$ for some $f > 0$ we need to make sure that if the firm remains inactive the cost is 0 ($c(0) = 0$). Let's try to write it in Python.

In [12]:
```
def demand(x1,x2,b):
    return 1-x1-b*x2

def cost(x,c):
    if x == 0:
     cost = 0
    else:
     cost = c*x
    return cost
```

Profits

They are defined by:

$$\pi(x_i, x_j) = p_i(x_i, x_j)x_i - c(x_i)$$

And we can translate this to Python by:

In [13]:
```
def profit(x1,x2,c1,b):
    return demand(x1,x2,b)*x1-cost(x1,c1)
```

Reaction Functions

Under Cournot competition the decision of one firm takes as given the choice of the other. That is, x_1^*, x_2^* is a Nash equilibrium if and only if

$$x_i^* = \arg\max_{x_i} \pi(x_i, x_j^*)$$

For each $i \neq j \in (1, 2)$. To find such a function we would take the first order condition equal to 0 and solve. However, with Python we can directly look for the profit maximizing outcome.

The scipy optimze library has a number of routines to optimize functions. They are all defined as minimization problems, so to maximize we just need to multiply by -1.

In [1]:
```
!pip install scipy
```

```
Requirement already satisfied: scipy in c:\users\sergi\anaconda3\lib\site-packages
(1.9.1)
Requirement already satisfied: numpy<1.25.0,>=1.18.5 in c:\users\sergi\anaconda3\l
ib\site-packages (from scipy) (1.21.5)
```

In [14]:
```
from scipy import optimize,arange
def reaction(x2,c1,b):
    x1 = optimize.brute(lambda x: -profit(x,x2,c1,b), ((0,1,),)) # brute minimizes
                                                                 # when we minimiz
    return x1[0]
```

Equilibrium as fixed point

To find for the equilibrium we are looking for the fixed point of both reaction functions. Let $r_i(x_j)$ denote firm $i's$ optimal response to $j's$ output level x_j. Then we need to find a point such that:

$$\begin{pmatrix} x_1^* \\ x_2^* \end{pmatrix} = \begin{pmatrix} r_1(x_1^*) \\ r_2(x_2^*) \end{pmatrix}$$

Defining the vector function $f(x_1, x_2)$ as

$$f(x) = \begin{pmatrix} r_1(x_1^*) \\ r_2(x_2^*) \end{pmatrix}$$

we are looking for a point $x^* = (x_1^*, x_2^*)$ such that $x^* = f(x^*)$. So the problem to find the fixed point is equivalent to find the x^* such that $x^* - f(x^*) = 0$. We can define this function in Python as:

In [15]:
```
from numpy import array
def vector_reaction(x,param): # vector param = (b,c1,c2)
    return array(x)-array([reaction(x[1],param[1],param[0]),reaction(x[0],param[2]
```

Where `param` = $[b, c_1, c_2]$.

Cournot equilibrium

To calculate the equilibrium we just need to find the point at which the previous function equals to 0. To do that we will use `fsolve` from `scipy.optimize`. Let's see how it works:

In [17]:
```
param = [1,0,0]   # Give the parameter values
x0 = [0.1,0.1]    # Initial guess

ans  = optimize.fsolve(vector_reaction, x0, args=(param))
print(ans)
```

```
[0.33332648 0.33332648]
```

Collison

The interesting part about this model is to see what would happen if the firms would instead maximize the joint profit function, instead of competing against each other. To simplify, we would focus on a case where firms are symmetric $c_1 = c_2 = c$ and have a symmetric output $x_1 = x_2 = x$.

It is then of our interest to see if the firms have incetives to collute, and wether the collusion would be sustained period after period.

We know that if the firms collude, the expected profit is $\pi(x, x)\frac{1}{1-\delta}$. Furthermore, if a firm decides to deviate, it will change a price slightly below the monopoly price. In the deviation period it will earn almost the monopoly profit, but for then on profits are zero since the firm will be punished by the other firm. Therefore, for collusion to be sustained we need:

$$\pi(x, x)\frac{1}{1-\delta} \geq \max_{\hat{x}} \pi(\hat{x}, x) + \frac{\delta}{1-\delta}\pi(x^*, x^*)$$

where δ is the discount factor and $\hat{x}$ is the optimal response of the firm to the opponent choosnig output level x.

We can rewrite the previous inequality as

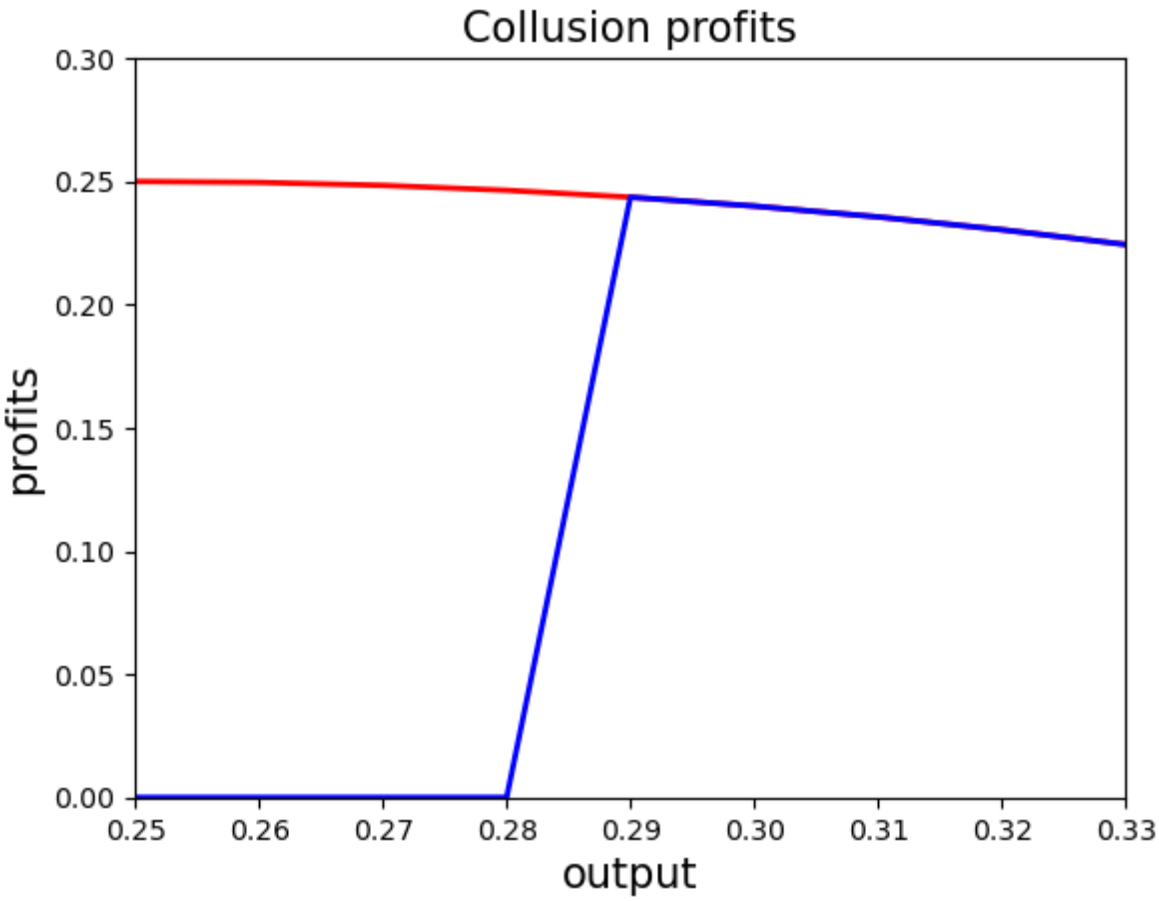
$$\pi(x, x) \geq (1-\delta)\max_{\hat{x}} \pi(\hat{x}, x) + \delta\pi(x^*, x^*)$$

In [18]:
```
def collusion_profits(x,b,c,delta): # we only do this for the symmetric case: c1 =
    profits = profit(x,x,c,b)
    ans = optimize.fsolve(vector_reaction, x0, args = ([b,c,c]))
    if profits >= (1-delta)*profit(reaction(x,c,b),x,c,b)+delta*profit(ans[0],ans[
        industry_profits = 2*profits # profits can be sustained as collusion profi
    else:
        industry_profits = 0 # profits cannot be sustained as collusion profits
    return industry_profits
```

In [19]:
```
import matplotlib.pyplot as plt

range_x = arange(0.25,ans[0],0.01)
delta1 = 0.8
delta2 = 0.3
```

```
range_profits = [collusion_profits(x,1.0,0.0,delta1) for x in range_x]
range_profits2 = [collusion_profits(x,1.0,0.0,delta2) for x in range_x]

plt.clf()

plt.plot(range_x, range_profits,'-', color = 'r', linewidth = 2)
plt.plot(range_x, range_profits2,'-', color = 'b', linewidth = 2)
plt.title("Collusion profits",fontsize = 15)
plt.xlabel("output",fontsize = 15)
plt.ylabel("profits",fontsize = 15,rotation = 90)
plt.xlim(0.25,0.33)
plt.ylim(0.0,0.3)
plt.savefig('collusion.png')
```

```
C:\Users\Sergi\AppData\Local\Temp\ipykernel_18664\2770697996.py:3: DeprecationWarn
ing: scipy.arange is deprecated and will be removed in SciPy 2.0.0, use numpy.aran
ge instead
  range_x = arange(0.25,ans[0],0.01)
```



Solving the model symbolically.

Follow the tutorial here.

Computing Transitions in a Representative Agent Economy

In this section we will see how to compute a transition from one steady state to another after a MIT shock is introduced to the economy.

The model

It is an optimal growth economy populated by a large number of identical infinitely lived households that maximize

$$E\{\sum_{t=0}^{\infty}\beta^t u(c_t)\}$$

over consumption and leisure $u(c_t) = lnc_t$, subject to:

$$c_t + i_t = y_t$$

$$y_t = k_t^{1-\theta}(zh_t)^{\theta}$$

$$i_t = k_{t+1} - (1-\delta)k_t$$

For this exercice we will set the labor share $\theta = 0.67$ and $h_t = 0.31$ for every t, so population does not grow.

The first thing we need to do is compute the initial steady state of the economy. We will do it by matching z to an annual capital-output ratio of 4, and an investment-output ratio of 0.25.

To do that we will solve for the solution of the economy. With a bit of algebra we can get:

$$k^* = zh^*\left[\frac{(1-\theta)\beta}{1-(1-\delta)\beta}\right]^{\frac{1}{\theta}}$$

Now we can impose that the capital-output ratio should be 4 and the investment-output ratio should be 0.25. With this we get that:

$$i^* = \delta k^* \leftrightarrow \delta = \frac{0.25}{4} = 0.0625$$

$$c^* = y^* - i^* = 1 - 0.25 = 0.75$$

$$y^* = k^*(1-\theta)(zh^*)^{\theta} \leftrightarrow z = \left(\frac{y}{k^{*1-\theta}h^{*\theta}}\right)^{\frac{1}{\theta}} = 1.629$$

And solving for β gives:

$$\beta = 0.98$$

The new Steady State

The economy will face a productivity show that will doblue the productivity, so that $z_{new} = 2z$.

The new values are:

$$k^*_{new} = 2zh^*\left[\frac{(1-\theta)\beta}{1-(1-\delta)\beta}\right]^{\frac{1}{\theta}}$$

$$y^*_{new} = (2k^*)^{1-\theta}(2h^*)^{\theta} = 2(k^{*1-\theta}(zh^*)^{\theta}) = 2$$

$$i^*_{new} = \delta k^*_{new} = 0.5$$

$$c^*_{new} = y^*_{new} - i^*_{new} = 1.5$$

The transition

To move from one steady state to another we will need to use the law of motion of capital. This law of motion needs to satisfy the optimality condition (Euler equation), so for this reason our transition path will be characterized by n Euler equaitons, that will be written only as a function of capital. So take the Euler equation as a function of capital:

$$k_{t+1}^{1-\theta}(zh_{t+1})^{\theta} + (1-\delta)k_{t+1} - k_{t+2} = \beta(k_t^{1-\theta}(zh_t)^{\theta} + (1-\delta)k_t - k_{t+1})((1-\theta)k_{t+1}^{-\theta}(zh_{t+1})$$

The procedure to characterize the transition path is the following:

1. We need to characterize n different periods Euler equations, so will will have n equations and $n+2$ unknowns
2. To have an identified system of equations, we can establish the initial capital level, which is the initial steady state and also the final capital level, which is the second steady state capital value.
3. Once we have a perfect identified system of n equations and n unknowns we can solve for it. Notice that our n unknowns are nothing else than the transition values of capital towards the steady state.

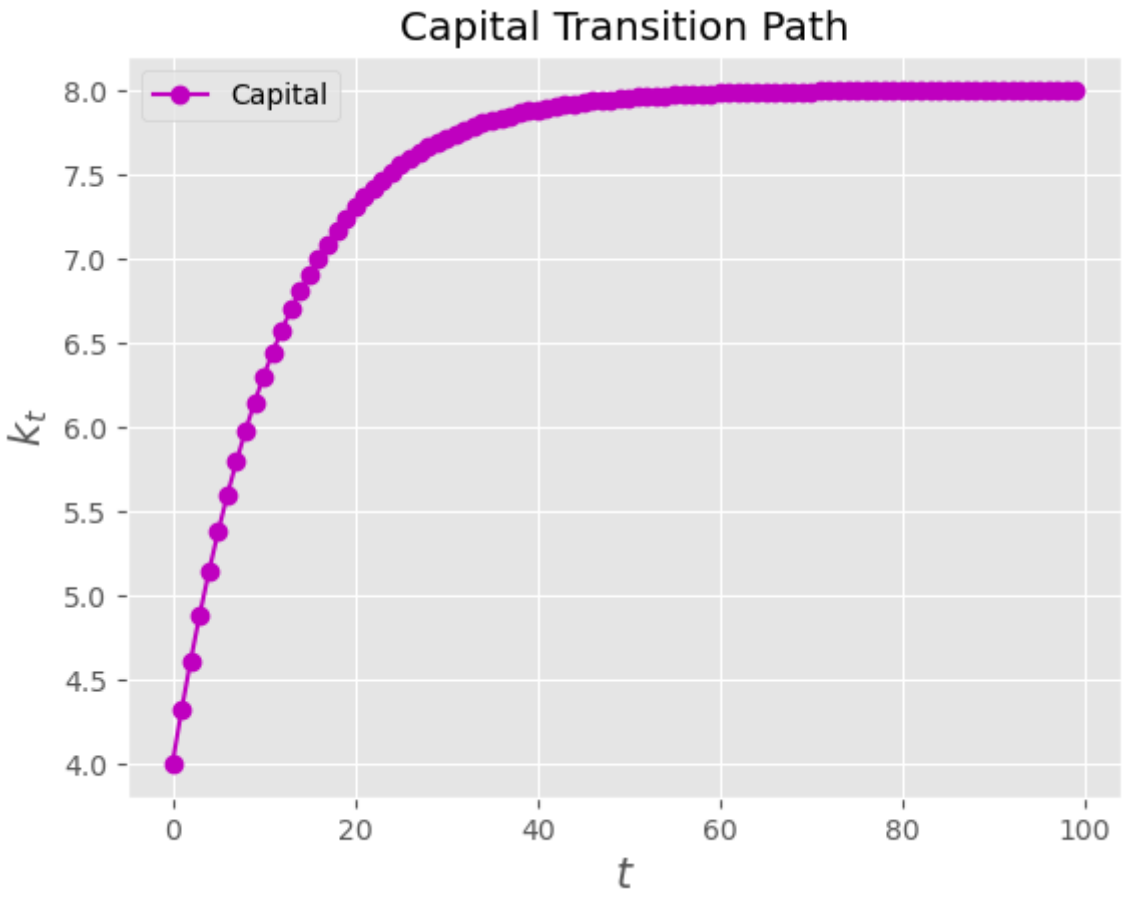
In [125…
```
# Define the parameters and values that I have computed analytically:
θ=0.67       # Labor Share
h=0.31       # Labor
k_i=4        # Capital at initial steady state
i_i=0.25     # Investment at initial steady state
c_i=0.75     # Consumption at initial steady state
y_i=1        # Normalized Output at intial steady state
δ=i_i/k_i        # Depreciation
z=1/((k_i**(1 - θ) * h ** θ) ** (1/θ))             # Labor productivity
β= 1 / ((1-θ) * k_i **(-θ) * (z*h) ** θ + 1 - δ)   # Discount factor
```

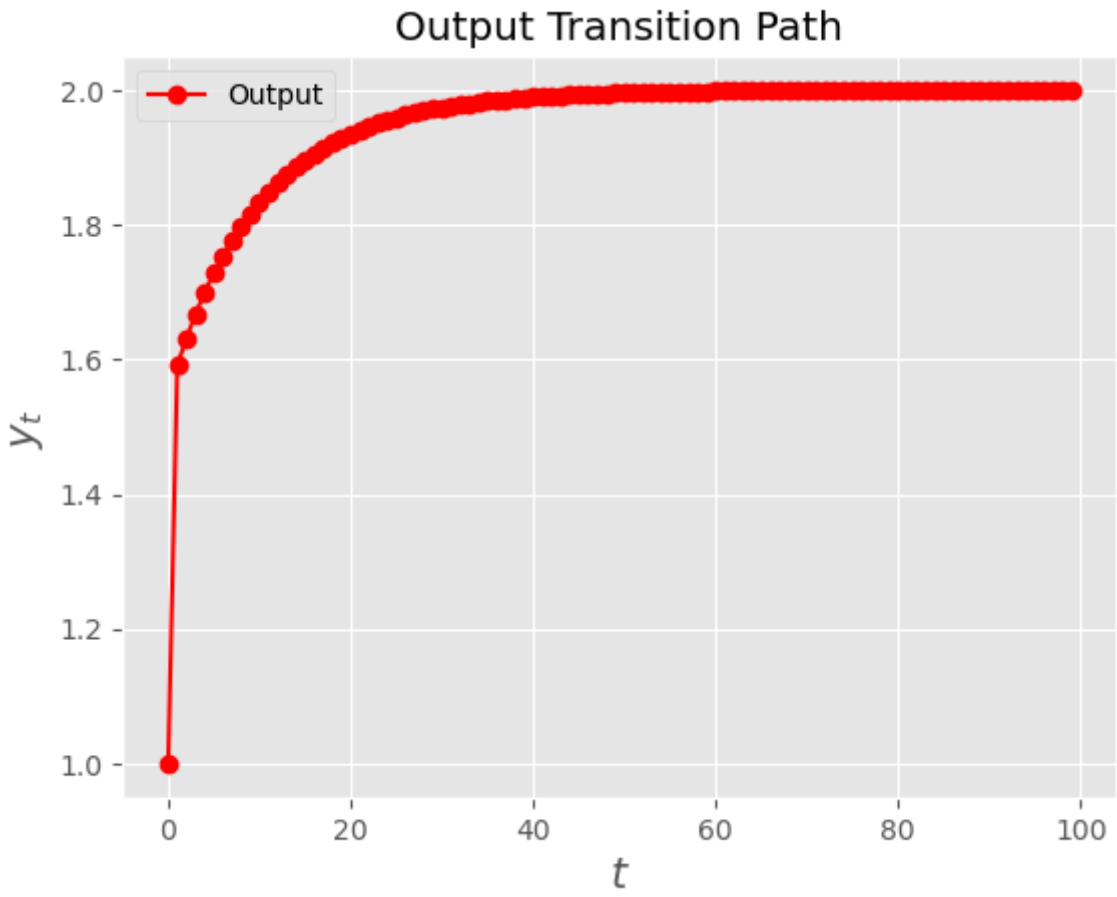
In [126…
```
# Values for the final steady state.
k_f=8
i_f=0.5
c_f=1.5
y_f=2
z=2*z    #Shock
```

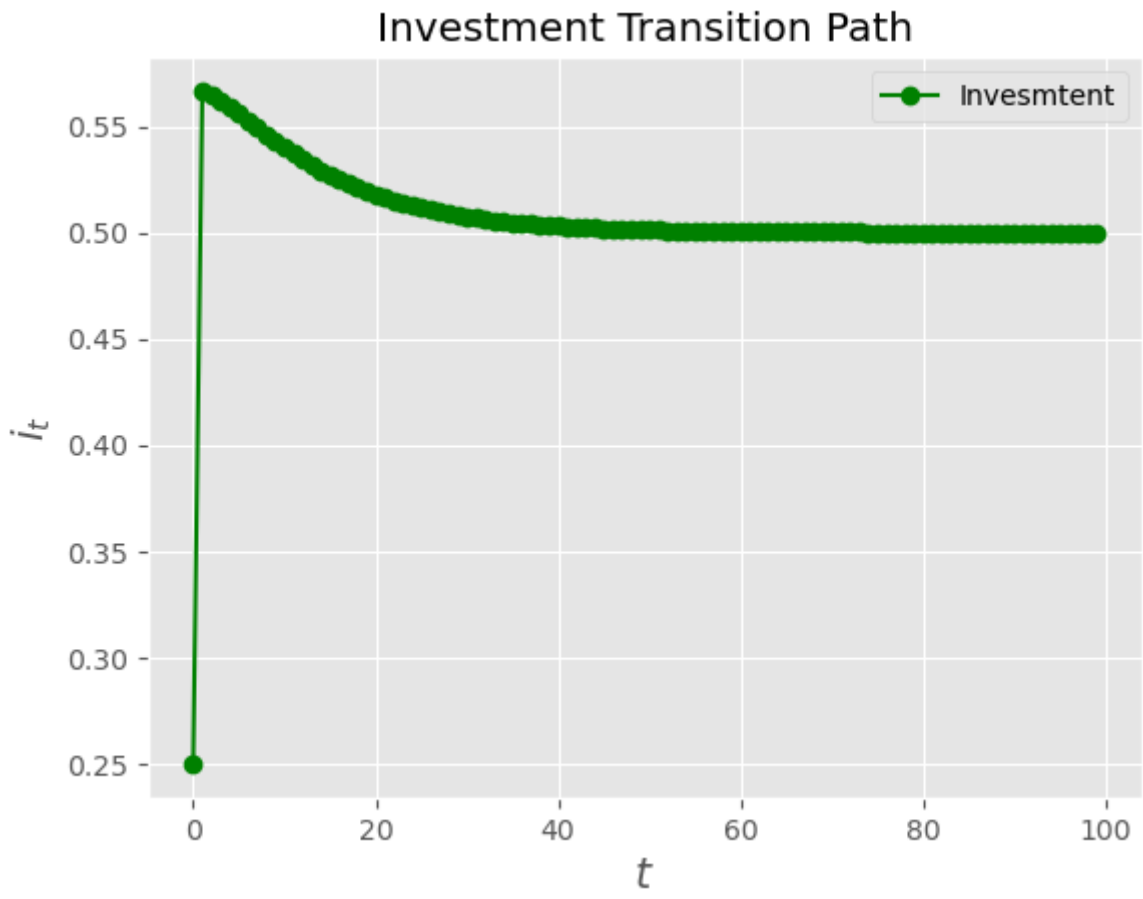
In [127…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
plt.style.use('ggplot')


########
# Define our variables:

def output(k,y_i=y_i):
    y = list( k[:-1]**(1-θ)*(z*h)**θ)
```

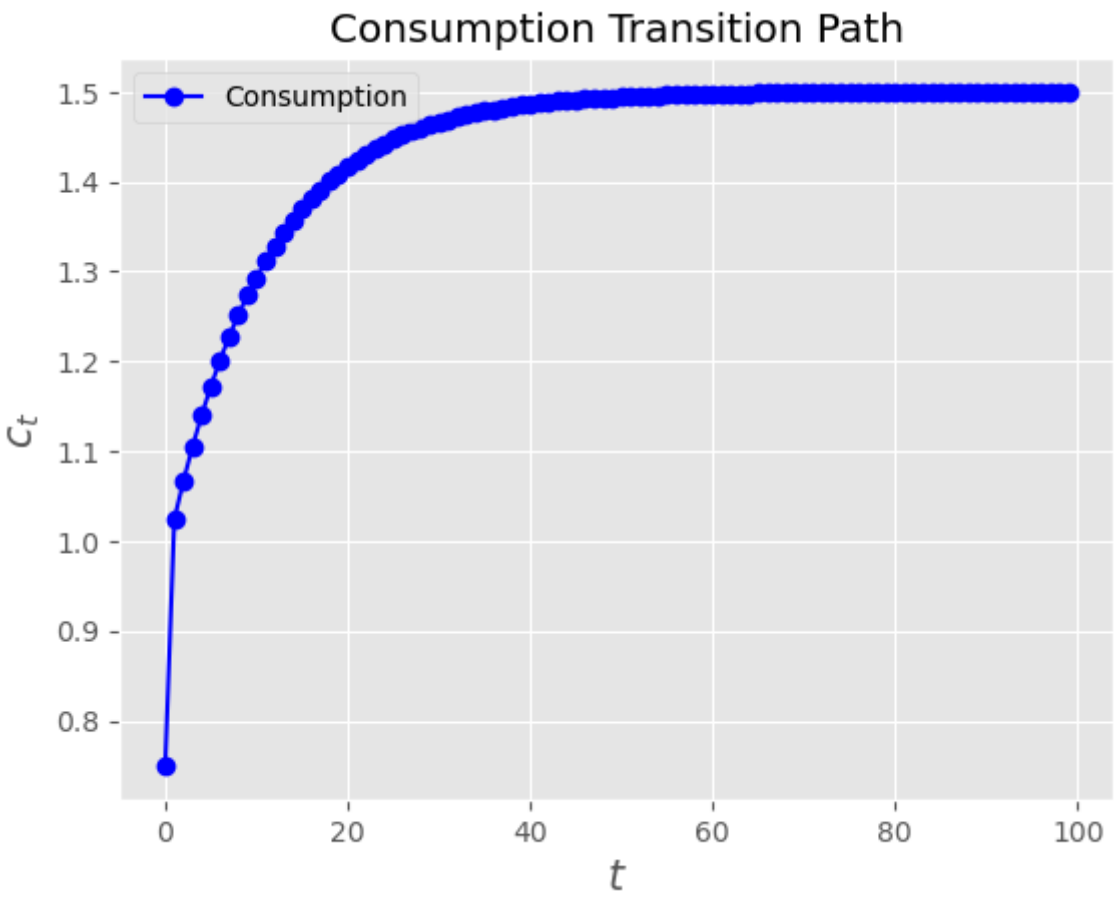
```
    return np.array([[y_i]+y]).reshape(n,)


def investment (k,i_i=i_i):
    i = list(k[1:]-(1-δ)*k[:-1])
    return np.array([[i_i] + i]).reshape(n,)


def consumption(y,i):
     return y-i



#Characterize the transition path:

n = 100 # Number of periods of transition


def transition_path(k,k_i=k_i,k_f=k_f,n=n):
    "This function returns the transition path towards the steady state over n peri
    k[0] = k_i      # Initial Capital Level (First Steady State)
    k[n-1] = k_f     # Final Capital Level (Second Steady State)

    #Now let's generate the "n" different Euler equations:
    trans = np.zeros(n)
    for i in range(n-2):
        if i == (n-2): #Since the other function is not well defined at n-2
            trans[i+1] = k[i+1]**(1-θ)*(z*h)**θ+(1-δ)*k[i+1]-k_f-β*(k[i]**(1-θ)*(z
        else:
            trans[i+1] = k[i+1]**(1-θ)*(z*h)**θ+(1-δ)*k[i+1]-k[i+2]-β*(k[i]**(1-θ)
    return trans

#Define an initial guess:

x0 = np.linspace(k_i,k_f,n)                 # Grid of n points between 4 and 8, wh
k_trans = fsolve(transition_path,x0)    # Solves the system of equations. Gives the


#Generate the transition path for the other variables:

## Output:
y = output(k_trans)
## Investment:
i_path = investment(k_trans)
## Consumption:
c = consumption(y,i_path)
# Plots:

def create_plots():
    fig, ax = plt.subplots()
    ax.plot(k_trans,'m-o',label="Capital")
    plt.ylabel("$k_t$",fontsize=15)
    plt.xlabel("$t$",fontsize=15)
    plt.legend()
    plt.title("Capital Transition Path")


    fig, ax = plt.subplots()
    ax.plot(y,'r-o',label = "Output")
    plt.ylabel("$y_t$",fontsize = 15)
    plt.xlabel("$t$",fontsize = 15)
    plt.legend()
    plt.title("Output Transition Path")
```

```
    fig, ax = plt.subplots()
    ax.plot(i_path,'g-o',label = "Invesmtent")
    plt.ylabel("$i_t$",fontsize = 15)
    plt.xlabel("$t$",fontsize = 15)
    plt.legend()
    plt.title("Investment Transition Path")

    fig, ax = plt.subplots()
    ax.plot(c,'b-o',label = "Consumption")
    plt.ylabel("$c_t$",fontsize = 15)
    plt.xlabel("$t$",fontsize = 15)
    plt.legend()
    plt.title("Consumption Transition Path")

create_plots()
```




Output Transition Path
Output
2.0
1.8
1.6
1.4
1.2
1.0
y_t
0
20
40
60
80
100
t


Investment Transition Path
Invesmtent
0.55
0.50
0.45
0.40
0.35
0.30
0.25
i_t
0
20
40
60
80
100
t



(Class Exercice!) Unexpected shocks. Let the agents believe productivity z_t doubles once and for all periods. However, after 10 periods, surprise the economy by cutting the productivity z_t back to its original value. Compute the transition for savings, consumption, labor and output.

In [ ]:

Stationary Economy Value Function Iteration

Consider a Stationary economy populated by a large number of identical infinitely lived households that maximize:

$$E_0\left\{\sum_{t=0}^{\infty}\beta^t u(c_t,h_t)\right\}$$

over consumption and leisure $u(c_t,1-h_t)=\ln c_t-\kappa\frac{h_t^{1+\frac{1}{\nu}}}{1+\frac{1}{\nu}}$ subject to:

$$c_t+i_t=y_t$$

$$y_t=k_t^{1-\theta}(h_t)^{\theta}$$

$$i_t=k_{t+1}-(1-\delta)k_t$$

Set $\theta = .679$,$\beta = .988$, $\delta = .013$. Also, to start with, set $h_t = 1$, that is, labor is inelastically supplied. To compute the steady-state normalize output to one.

Write the problem in recursive formulation

1. Control Variable: k
2. Choice Variables: k', c

$$V(k) = \max_{\{k',c\} \in A} \ln c_t - \kappa \frac{1}{1+\frac{1}{\nu}} + \beta V(k')$$

where:

$$A = \{(k', c) | c + i = y, y = k^{1-\theta}, i_t = k' - (1-\delta)k, c > 0\}$$

So we can rewrite the problem as:

$$V(k) = \max_{\{k'\} \in [0, k^{1-\theta} + (1-\delta)k)} \ln(k^{1-\theta} - k' + (1-\delta)k) - \kappa \frac{1}{1+\frac{1}{\nu}} + \beta V(k')$$

Notice that it is interesting to find the value of k at the steady state to define our grid properly. To do so I will characterize the Euler equation in sequential form and then impose stationary. The problem is:

$$\max_{k_{t+1}} \sum_{t=0}^{\infty} \beta^t \left(\ln(k_t^{1-\theta} + (1-\delta)k_t - k_{t+1}) - \frac{\kappa}{1+\frac{1}{\nu}} \right)$$

the Foc gives:

$$\frac{\partial}{\partial k_{t+1}} = 0 \iff \frac{\beta^{t+1}((1-\theta)k_{t+1}^{-\theta} + 1 - \delta)}{(k_{t+1}^{1-\theta} + (1-\delta)k_{t+1} - k_{t+2})} = \frac{\beta^t}{(k_t^{1-\theta} + (1-\delta)k_t - k_{t+1})}$$

And now if we impose stationary we get:

$$\beta((1-\theta)k^{-\theta} + 1 - \delta) = 1 \iff k_{ss} = \left(\frac{1-\theta}{\frac{1}{\beta} - 1 + \delta} \right)^{\frac{1}{\theta}}$$

And now, we can discretize k in a grid with k_{max} slightly above the steady state.

Solve with brute force iterations of the value function. Plot your value function.

In [111…

```
!pip install quantecon
```

```
Collecting quantecon
  Downloading quantecon-0.6.0-py3-none-any.whl (206 kB)
     ------------------------------------ 206.9/206.9 kB 4.2 MB/s eta 0:00:00
Requirement already satisfied: scipy>=1.5.0 in c:\users\sergi\anaconda3\lib\site-p
ackages (from quantecon) (1.9.1)
Requirement already satisfied: requests in c:\users\sergi\anaconda3\lib\site-packa
ges (from quantecon) (2.28.1)
Requirement already satisfied: sympy in c:\users\sergi\anaconda3\lib\site-packages
(from quantecon) (1.10.1)
Requirement already satisfied: numpy>=1.17.0 in c:\users\sergi\anaconda3\lib\site-
packages (from quantecon) (1.21.5)
Requirement already satisfied: numba in c:\users\sergi\anaconda3\lib\site-packages
(from quantecon) (0.55.1)
Requirement already satisfied: setuptools in c:\users\sergi\anaconda3\lib\site-pac
kages (from numba->quantecon) (63.4.1)
Requirement already satisfied: llvmlite<0.39,>=0.38.0rc1 in c:\users\sergi\anacond
a3\lib\site-packages (from numba->quantecon) (0.38.0)
Requirement already satisfied: charset-normalizer<3,>=2 in c:\users\sergi\anaconda
3\lib\site-packages (from requests->quantecon) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\sergi\anaconda3\lib
\site-packages (from requests->quantecon) (2022.9.14)
Requirement already satisfied: idna<4,>=2.5 in c:\users\sergi\anaconda3\lib\site-p
ackages (from requests->quantecon) (3.3)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\sergi\anaconda3\l
ib\site-packages (from requests->quantecon) (1.26.11)
Requirement already satisfied: mpmath>=0.19 in c:\users\sergi\anaconda3\lib\site-p
ackages (from sympy->quantecon) (1.2.1)
Installing collected packages: quantecon
Successfully installed quantecon-0.6.0
```

In [ ]:

The speed of the iteration is fundamental. Let's try to improve our algorithm:

Iterations of the value function taking into account monotonicity of the optimal decision rule.

Now the intention is to reduce the number of points to be searched on the grid by the VFI algorithm. We know that the optimal decision rule increases in k. Therefore, if $k_j > k_i$, then $g(k_j) \geq g(k_i)$. For this reason we know that if we want to find $g(k_j)$ when $k_j > k_i$ we can rule out searching for all grid values of capital that are smaller than $g(k_i)$.

Iterations of the value function taking into account concavity of the value function.

It is known that the maximand in the Bellman equation, $M_{i,j} + \beta V_j$, is strictly concave in k'. Therefore, if $M_{i,j} + \beta V_j > M_{i,j+1} + \beta V_{k+1}$, then $M_{i,j} + \beta V_j > M_{i,j+2} + \beta V_{k+2}$. Therefore, to make more efficient our algorithm we can include this condition.

Iterations of the value function taking into account local search on the decision rule.

We know that if $k_j = g(k_i)$ then it is reasonable to think that $g(k_{i+1})$ is in a small neigborhood of k_j.

In [ ]: