María González Cabrera (in collaboration with Germán Sánchez Arce)

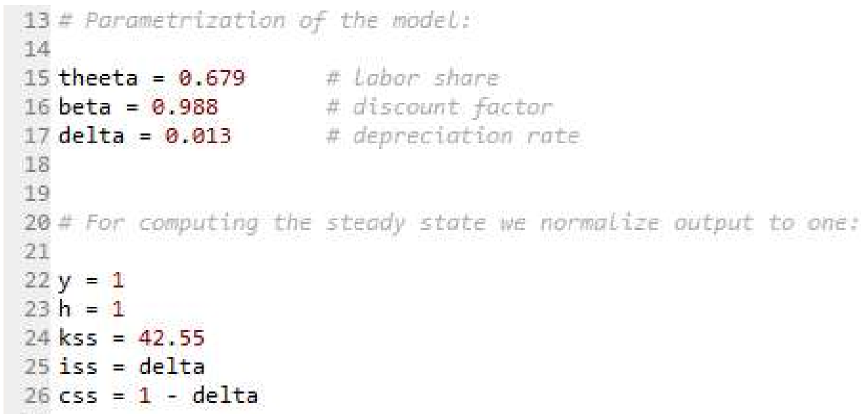
## Question 1. Value Function Iteration

1. Recursive formulation without labor

First of all we compute the steady state of the economy taking into account the normalization of the output to one (y = 1) and setting labor equal to one (h = 1).

Given that labor is setting equal one, this first part of the question one is assuming that utility function is only the logarithm of consumption today, because if there was the case of having a utility function that takes into account the disutility of labor, optimal hours worked would not be equalize to one anymore.

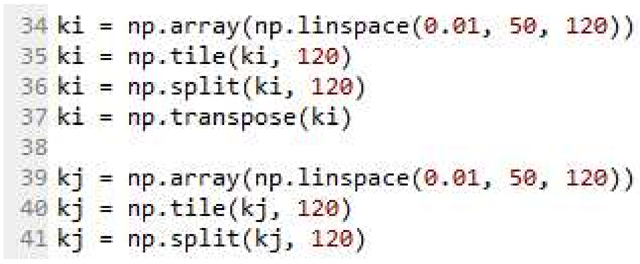
Therefore, the steady state and parametrization of the model is given by:



```
# Parametrization of the model:

theeta = 0.679      # labor share
beta = 0.988        # discount factor
delta = 0.013       # depreciation rate


# For computing the steady state we normalize output to one:

y = 1
h = 1
kss = 42.55
iss = delta
css = 1 - delta
```

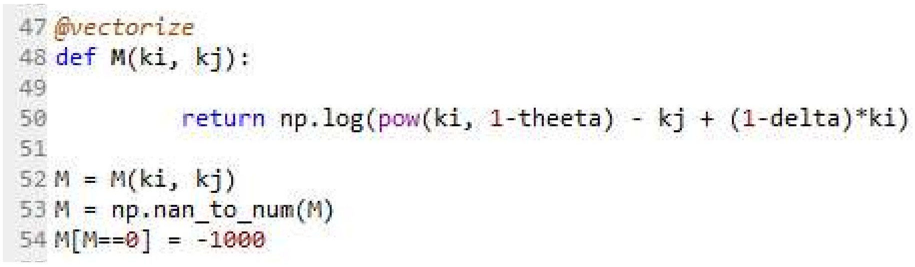
The first steps that we have used for the different approaches of solving the value function are the following ones:

- **Step 1:** discretize the variable of interest. That is, we set a grid for capital today (ki) and capital tomorrow (kj), where the lower bound is close to zero (we cannot set it equal to zero for avoiding problems with the solution) and the upper bound is a bit larger than the steady state that we have obtained (so we can see the path followed until the steady state).



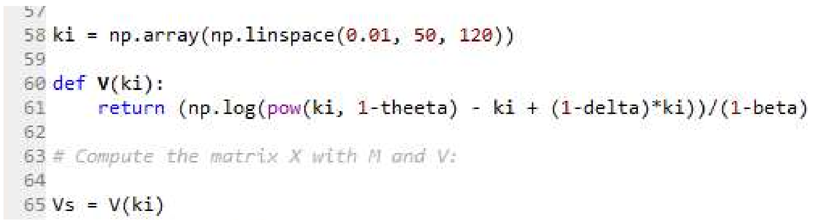
```
ki = np.array(np.linspace(0.01, 50, 120))
ki = np.tile(ki, 120)
ki = np.split(ki, 120)
ki = np.transpose(ki)

kj = np.array(np.linspace(0.01, 50, 120))
kj = np.tile(kj, 120)
kj = np.split(kj, 120)
```

- **Step 2:** create the matrix M, that is the return matrix which give us the utility of all the possible combinations of ki and kj. The numbers of the matrix M that are *nan* are those associated with non feasible points, so we need to set those points to a very negative number.



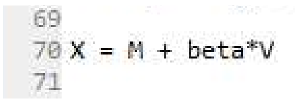
```python
@vectorize
def M(ki, kj):

        return np.log(pow(ki, 1-theeta) - kj + (1-delta)*ki)

M = M(ki, kj)
M = np.nan_to_num(M)
M[M==0] = -1000
```

- **Step 3:** define the initial guess for the value function. The idea is that, independently on the initial guess that we use, the value function will converge to the true one. Given that, this initial guess can be a vector of zeros, or a vector of the utility evaluated when ki is equal to kj.



```python
ki = np.array(np.linspace(0.01, 50, 120))

def V(ki):
    return (np.log(pow(ki, 1-theeta) - ki + (1-delta)*ki))/(1-beta)

# Compute the matrix X with M and V:

Vs = V(ki)
```

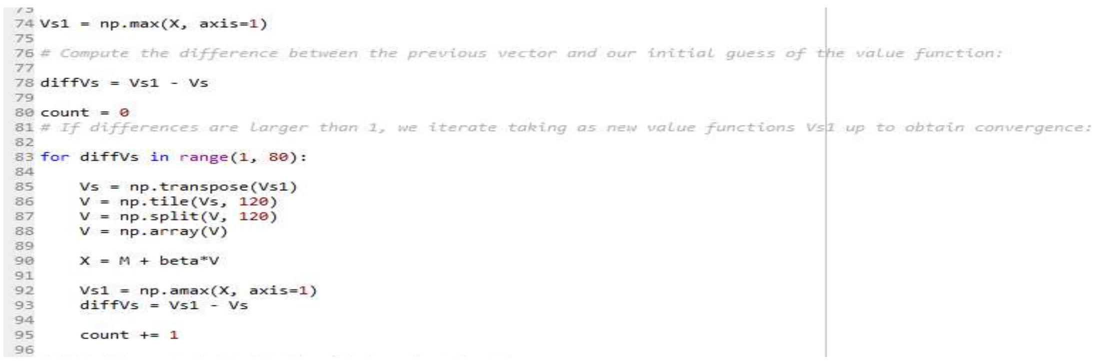
- **Step 4:** compute the matrix X, which is containing the Bellman equations for all combinations of ki and kj.



```python
X = M + beta*V
```

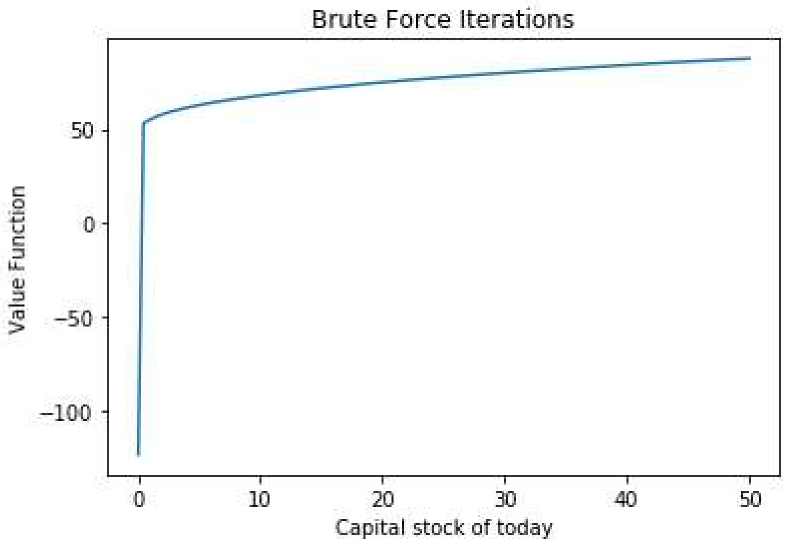
a) Brute force iterations:

Given the previous steps, the brute force iteration approach, approximates the value function in the following form. Given the matrix X formed with the initial guess for the value function, the value function used in the following iteration will be the maximum of each of the rows of matrix X. The iterations will end when the differences between those value functions would be less than $\varepsilon$.



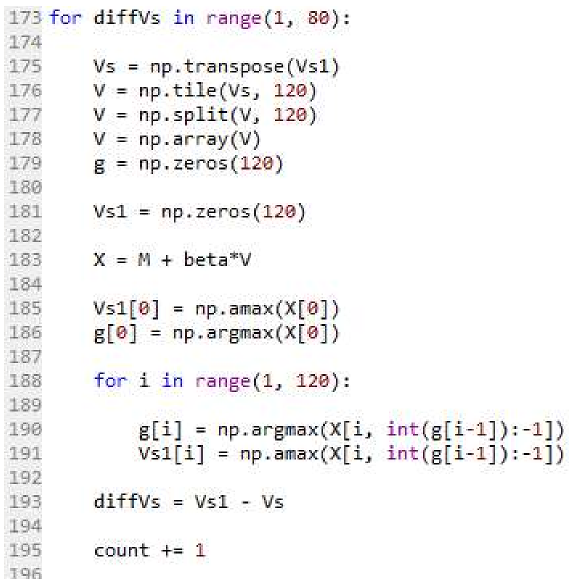
```python
Vs1 = np.max(X, axis=1)

# Compute the difference between the previous vector and our initial guess of the value function:

diffVs = Vs1 - Vs

count = 0
# If differences are larger than 1, we iterate taking as new value functions Vs1 up to obtain convergence:

for diffVs in range(1, 80):

    Vs = np.transpose(Vs1)
    V = np.tile(Vs, 120)
    V = np.split(V, 120)
    V = np.array(V)

    X = M + beta*V

    Vs1 = np.amax(X, axis=1)
    diffVs = Vs1 - Vs

    count += 1
```

After 79 iterations and 0.05 seconds, we obtain the following value function:

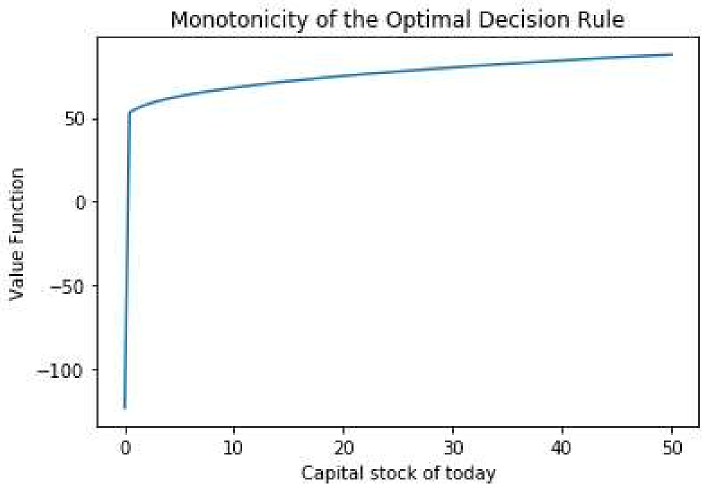


b) Monotonicity of the optimal decision rule:

The idea behind this approach is that we don't need to compute the entire matrix X, so if we have obtained an optimal kj for a given ki, the next kj will not be less than the previous one, because of the fact that the optimal decision rule is monotone (if ki<kj, then gi<gj). Given this idea, our code for the iterations looks as follows:

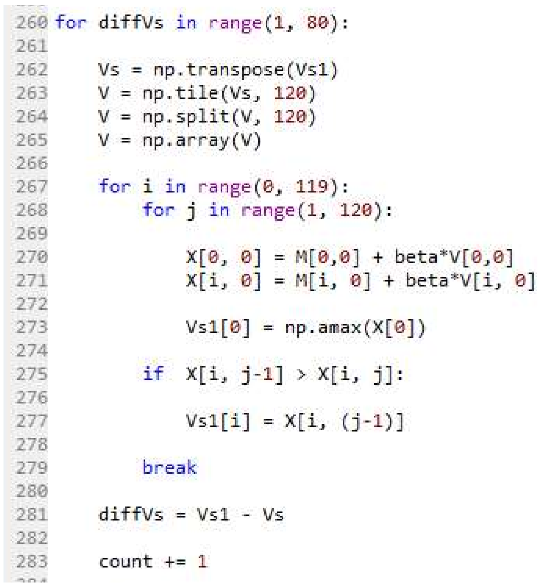

```
for diffVs in range(1, 80):

    Vs = np.transpose(Vs1)
    V = np.tile(Vs, 120)
    V = np.split(V, 120)
    V = np.array(V)
    g = np.zeros(120)

    Vs1 = np.zeros(120)

    X = M + beta*V

    Vs1[0] = np.amax(X[0])
    g[0] = np.argmax(X[0])

    for i in range(1, 120):

        g[i] = np.argmax(X[i, int(g[i-1]):-1])
        Vs1[i] = np.amax(X[i, int(g[i-1]):-1])

    diffVs = Vs1 - Vs

    count += 1
```

Given that, we have obtained the following approximation for the value function after 79 iterations and 0.1 seconds of execution time:



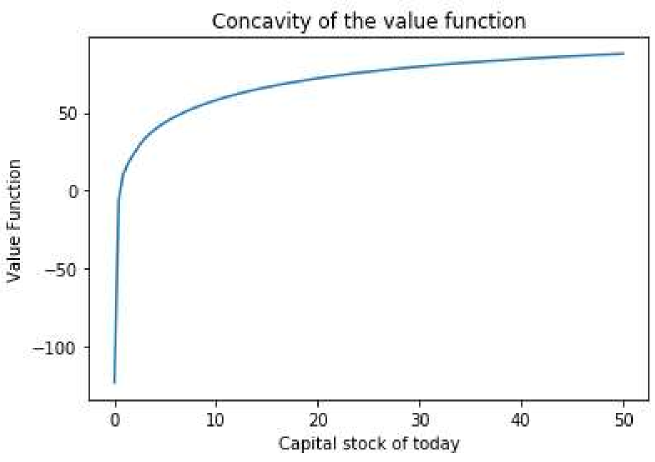
c) Concavity of the value function

This approach follows the same idea as the previous one. In order to avoid to compute the entire matrix X, we need to take into account the concavity of the value function. That is, we compute the elements of a given row of X up to a point where this value is larger than the next one. All those points would be our value function for the following iteration, as it is shown below:



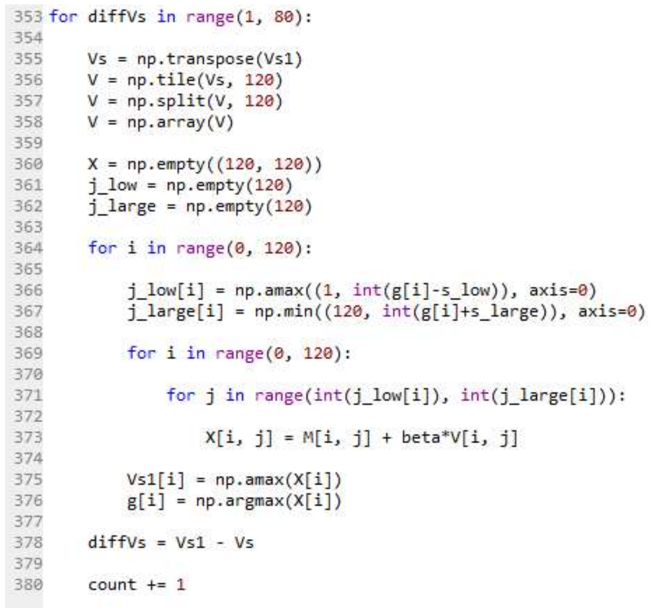
```
for diffVs in range(1, 80):

    Vs = np.transpose(Vs1)
    V = np.tile(Vs, 120)
    V = np.split(V, 120)
    V = np.array(V)

    for i in range(0, 119):
        for j in range(1, 120):

            X[0, 0] = M[0,0] + beta*V[0,0]
            X[i, 0] = M[i, 0] + beta*V[i, 0]

            Vs1[0] = np.amax(X[0])

        if  X[i, j-1] > X[i, j]:

            Vs1[i] = X[i, (j-1)]

        break

    diffVs = Vs1 - Vs

    count += 1
```

After 79 iterations and 0.08 seconds of execution time, we have obtained the following value function:



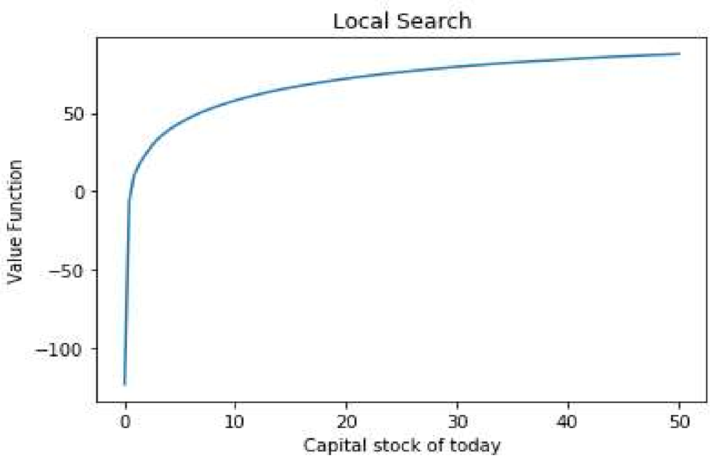
d) Local search:

This approach is similar to the previous one. Given that we have obtained an optimal kj for a ki (which is given by the optimal decision rule g), the next kj will be in a grid around the previous one. It is shown below:



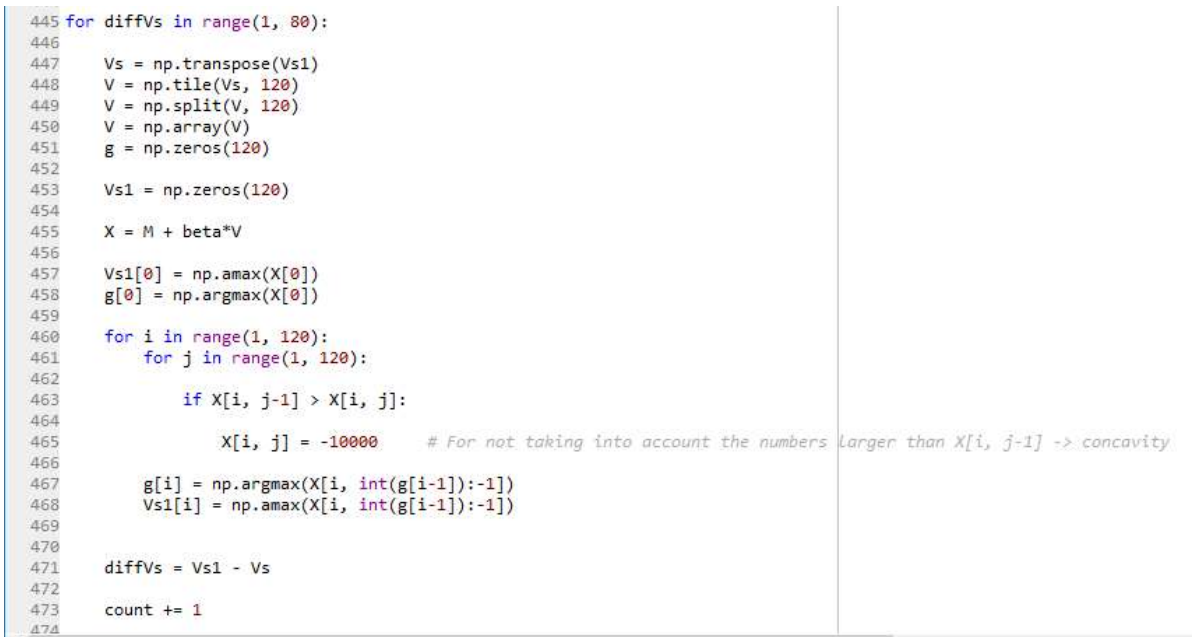
```
for diffVs in range(1, 80):

    Vs = np.transpose(Vs1)
    V = np.tile(Vs, 120)
    V = np.split(V, 120)
    V = np.array(V)

    X = np.empty((120, 120))
    j_low = np.empty(120)
    j_large = np.empty(120)

    for i in range(0, 120):

        j_low[i] = np.amax((1, int(g[i]-s_low)), axis=0)
        j_large[i] = np.min((120, int(g[i]+s_large)), axis=0)

        for i in range(0, 120):

            for j in range(int(j_low[i]), int(j_large[i])):

                X[i, j] = M[i, j] + beta*V[i, j]

        Vs1[i] = np.amax(X[i])
        g[i] = np.argmax(X[i])

    diffVs = Vs1 - Vs

    count += 1
```

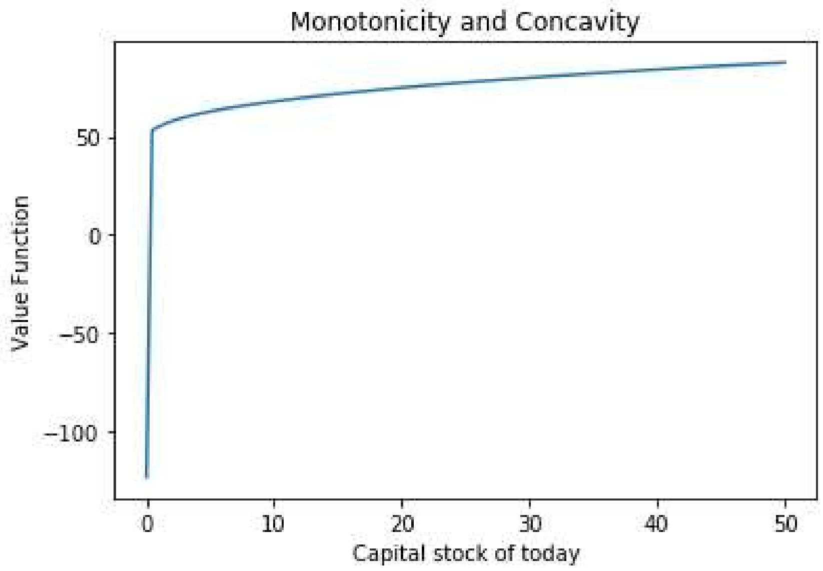
This procedure, give us the following graph of the value function (after 79 iterations and 41.03 seconds).



e) Concavity of the value function and monotonicity of the optimal decision rule:

The code and graph obtained taking into account both, concavity of the value function and monotonicity of the optimal decision rule are as follows:

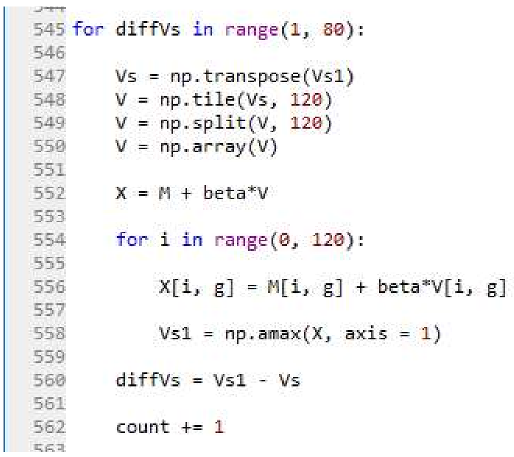

```
for diffVs in range(1, 80):

    Vs = np.transpose(Vs1)
    V = np.tile(Vs, 120)
    V = np.split(V, 120)
    V = np.array(V)
    g = np.zeros(120)

    Vs1 = np.zeros(120)

    X = M + beta*V

    Vs1[0] = np.amax(X[0])
    g[0] = np.argmax(X[0])

    for i in range(1, 120):
        for j in range(1, 120):

            if X[i, j-1] > X[i, j]:

                X[i, j] = -10000     # For not taking into account the numbers larger than X[i, j-1] -> concavity

        g[i] = np.argmax(X[i, int(g[i-1]):-1])
        Vs1[i] = np.amax(X[i, int(g[i-1]):-1])


    diffVs = Vs1 - Vs

    count += 1
```



(obtained after 79 iterations and 0.49 seconds).

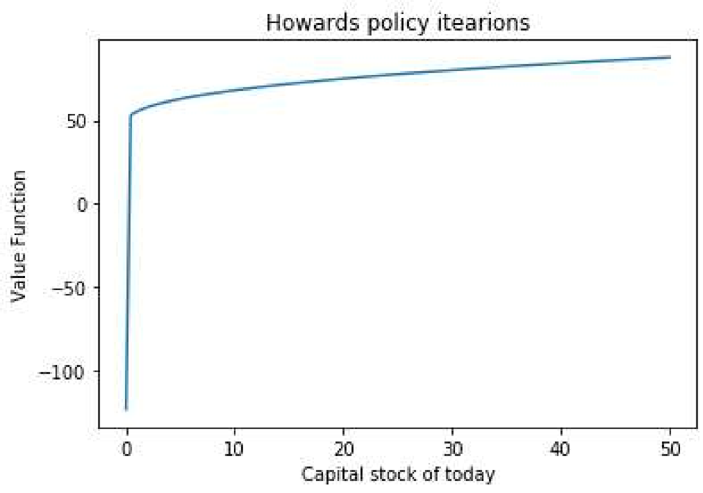
f) Howard's policy iterations:

This approach is very similar to the first one (brute force iterations). The only difference now is that the value function of each iteration is taking into account the capital associated to the optimal decision rule of the first iteration, as follows:



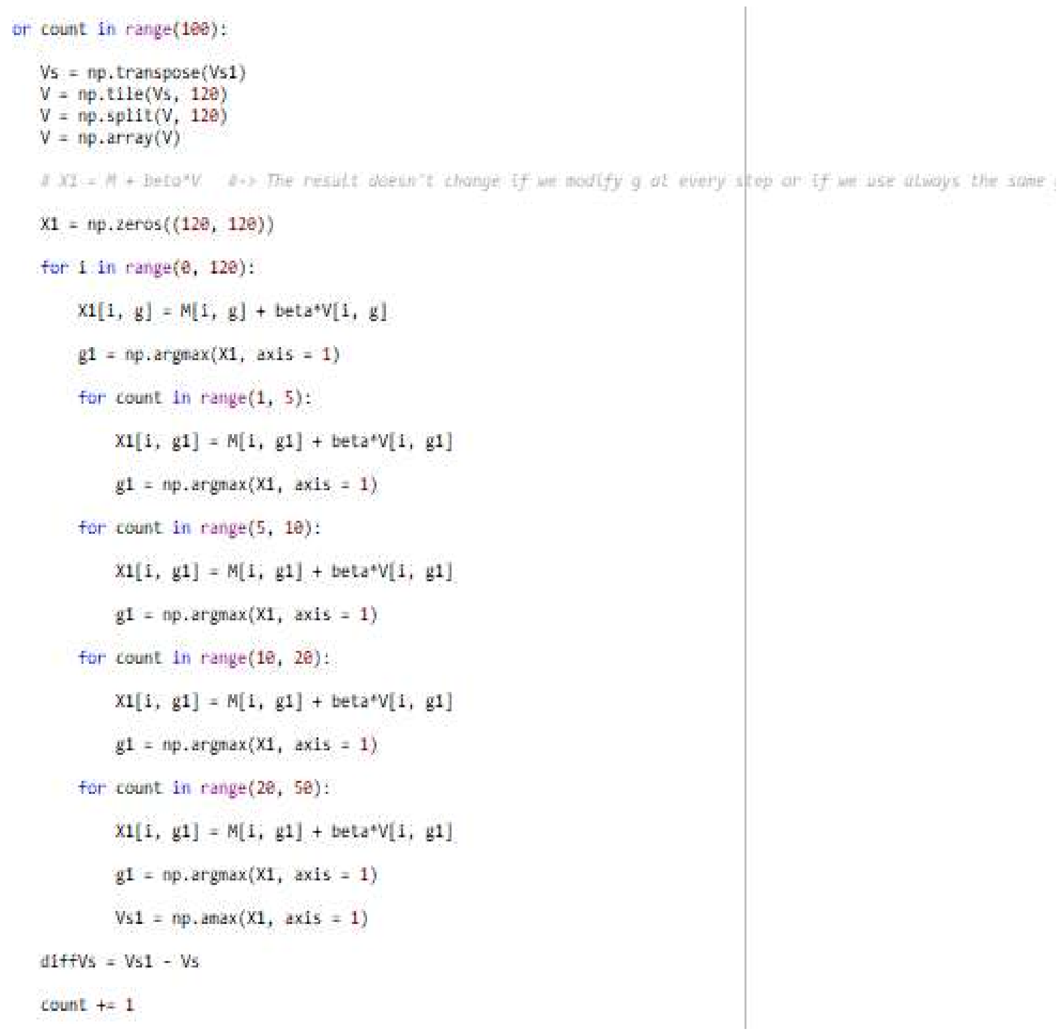
```
for diffVs in range(1, 80):

    Vs = np.transpose(Vs1)
    V = np.tile(Vs, 120)
    V = np.split(V, 120)
    V = np.array(V)

    X = M + beta*V

    for i in range(0, 120):

        X[i, g] = M[i, g] + beta*V[i, g]

        Vs1 = np.amax(X, axis = 1)

    diffVs = Vs1 - Vs

    count += 1
```

Given this approach, we have obtained the following graph of the value function, after 79 iterations and 0.24 seconds:

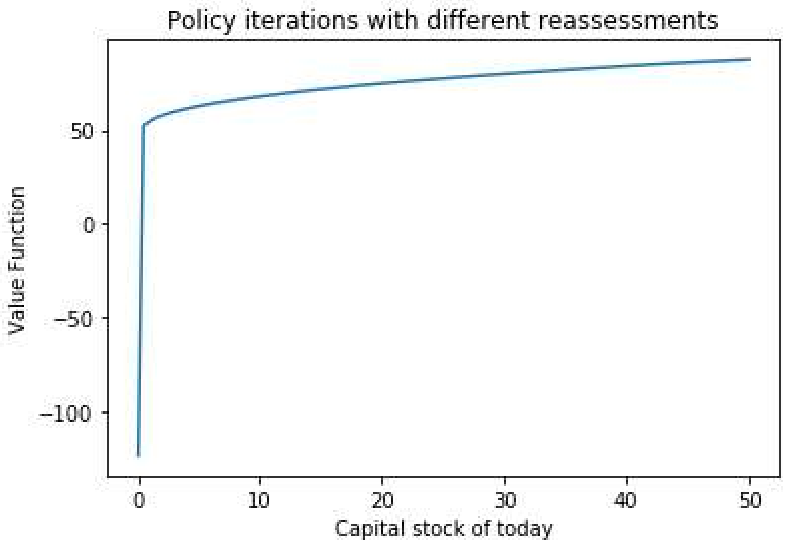


g) Policy iterations with 5, 10, 20 and 50 steps:

The code and the graph are shown below (this one has been obtained after 50 iterations and 21.42 seconds):

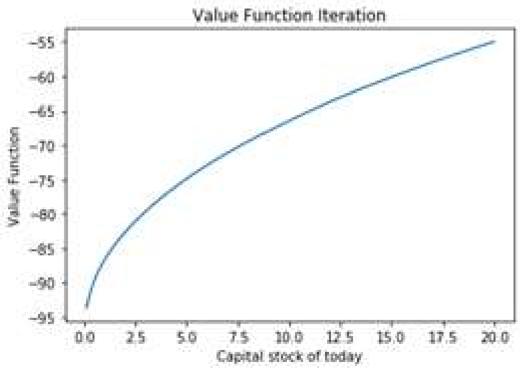


```
or count in range(100):

   Vs = np.transpose(Vs1)
   V = np.tile(Vs, 120)
   V = np.split(V, 120)
   V = np.array(V)

   # X1 = M + beta*V   #-> The result doesn't change if we modify g at every step or if we use always the same

   X1 = np.zeros((120, 120))

   for i in range(0, 120):

       X1[i, g] = M[i, g] + beta*V[i, g]

       g1 = np.argmax(X1, axis = 1)

       for count in range(1, 5):

           X1[i, g1] = M[i, g1] + beta*V[i, g1]

           g1 = np.argmax(X1, axis = 1)

       for count in range(5, 10):

           X1[i, g1] = M[i, g1] + beta*V[i, g1]

           g1 = np.argmax(X1, axis = 1)

       for count in range(10, 20):

           X1[i, g1] = M[i, g1] + beta*V[i, g1]

           g1 = np.argmax(X1, axis = 1)

       for count in range(20, 50):

           X1[i, g1] = M[i, g1] + beta*V[i, g1]

           g1 = np.argmax(X1, axis = 1)

           Vs1 = np.amax(X1, axis = 1)

   diffVs = Vs1 - Vs

   count += 1
```



2. Redo item 1 adding a labour choice that is continuous

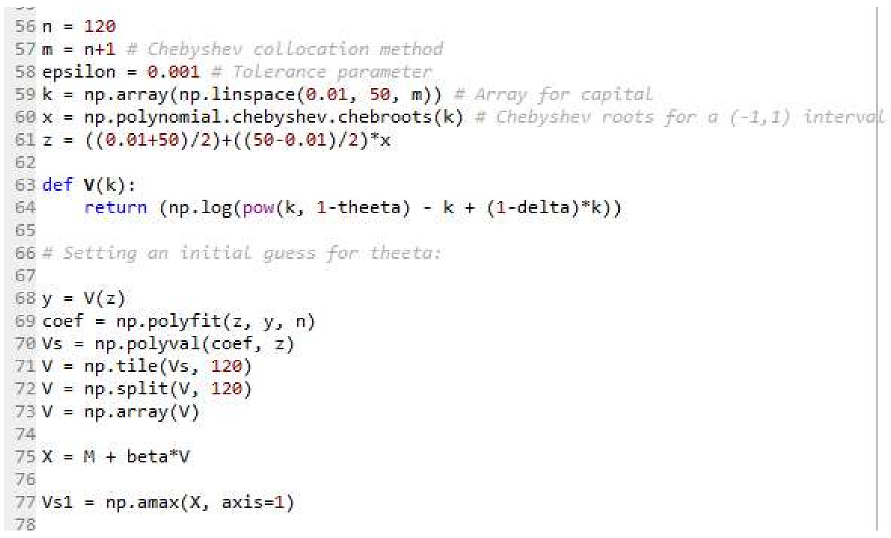
Now, labour is not going to be constant but individuals we will have to choose it each period. Then, it is a control variable. However, the process should be the same as in part 1.1 (without labour). However, now we should choose today the labour that maximizes the utility in each period for each amount of capital.



The shape of the value function is identical as in the first model.

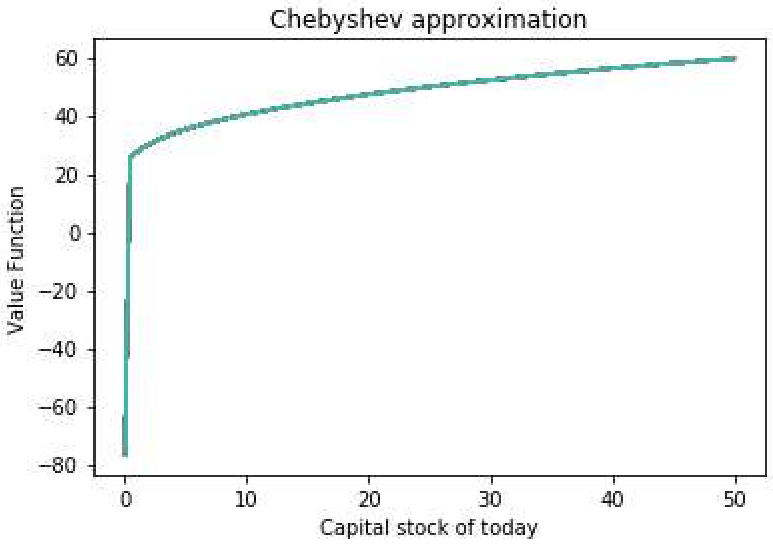
3. Chebyshev approximation of the value function

For this approach, the initial guess for the value function is using the coefficients of the Chebyshev polynomial and the Chebyshev nodes:



```
n = 120
m = n+1 # Chebyshev collocation method
epsilon = 0.001 # Tolerance parameter
k = np.array(np.linspace(0.01, 50, m)) # Array for capital
x = np.polynomial.chebyshev.chebroots(k) # Chebyshev roots for a (-1,1) interval
z = ((0.01+50)/2)+((50-0.01)/2)*x

def V(k):
    return (np.log(pow(k, 1-theeta) - k + (1-delta)*k))

# Setting an initial guess for theeta:

y = V(z)
coef = np.polyfit(z, y, n)
Vs = np.polyval(coef, z)
V = np.tile(Vs, 120)
V = np.split(V, 120)
V = np.array(V)

X = M + beta*V

Vs1 = np.amax(X, axis=1)

```

After 0.08 seconds and 79 iterations we obtain the following graph:



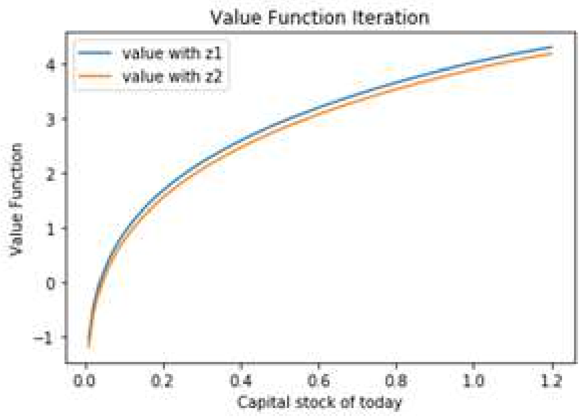
**Question 2. Business Cycle Fluctuations**

1. Add productivity shocks, Zt, to the previous model:

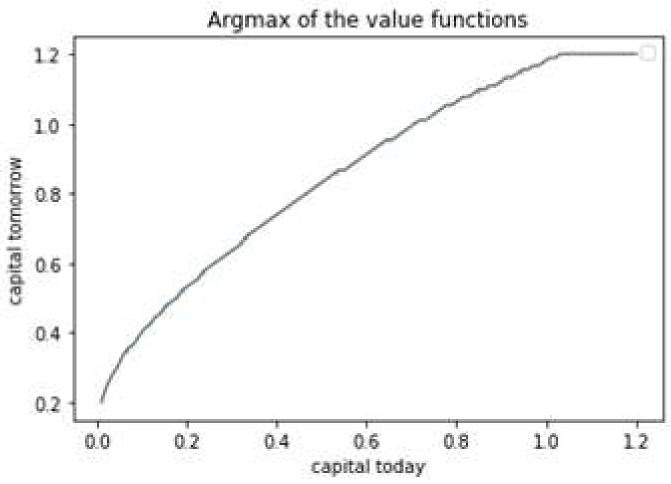
Now, we are going to add to the first model (keeping the labor constant to 1) a stochastic process based on productivity shocks zt. We can have a positive shock: z1=1.01 or a negative shock: z2=1/1.01. Then, what we did was to evaluate the value functions as expectancies of having V with "z" positive or negative. To calculate this, since the unconditional mean of "z" is equal to one, we can compute the probabilities of each case.

We obtain that  (represents the probability of obtaining the negative shock 1/1.01. Consequently,  is the probability of the positive shock to happen in the next period.

Proceeding as in the previous case, we obtain a matrix through which we will iterate so as to get the following result:



As we can observe, the value function is increasing with respect to the initial capital, as it happened in the deterministic case. From the data we can obtain as well the argmax of the value function, which is going to map our capital today with our optimal choice of capital tomorrow.

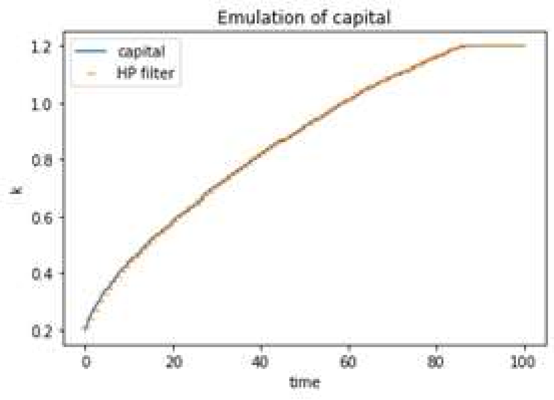


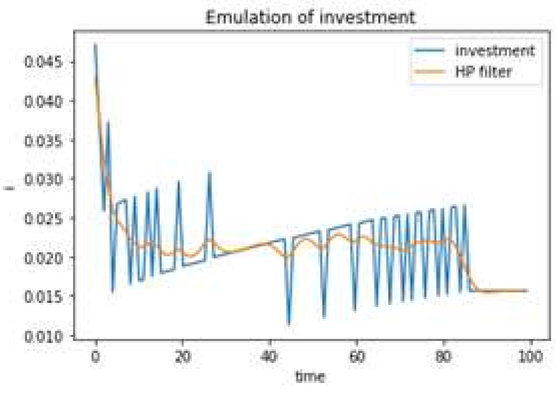
Again, capital is slightly increasing across the different values of capital today (it means that in general your capital for next period is going to increase with respect to the current period).

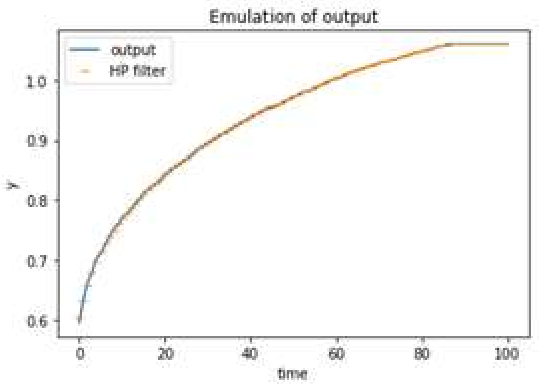
2. Simulate the economy

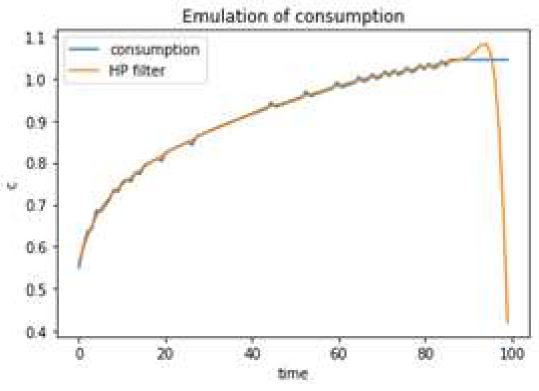
For this, we have created a random vector of shocks for a period of 100 years. These shocks, as we described above, can have two values: 1.01 or 1/1.01. To start with the simulation we have used an initial capital today of 0.1 (our first point of the grid). Then, we have used the optimal decisions of k' depending on our current shock and capital today. Once we have it, we can easily compute by applying the formulas a path for the rest of the variables: investment, output and consumption.

At the same time, we have used a Hodrick-Prescott filter in order to remove the cyclical component of the data paths. In the graphs, it can be easily seen as an smooth line that follows the real trend of the different variables.


Emulation of capital
capital
HP filter
k
time


Emulation of investment
investment
HP filter
i
time


Emulation of output
output
HP filter
y
time


Emulation of consumption
consumption
HP filter
c
time