OSM Boot Camp: **Econ ProbSet1**

Harrison Beard

25 June 2018

Exercise 1.

Question. Consider the problem of the owner of an oil field. The owner has B barrels of oil. She can sell these barrels at price p, at time t. Her objective is to maximize the discounted present value of sales of oil—we'll assume there are no extraction costs. The owner discounts the future at a rate given by $\frac{1}{1+r}$ (where r is the real interest rate and assumed to be constant). Answer the following:

1. What are the state variables?
2. What are the control variables?
3. What does the transition equation look like?
4. Write down the sequence problem of the owner. Write down the Bellman equation.
5. What does the owner's Euler equation look like?
6. What would the solution of the problem look like if $p_{t+1} = p_t$ for all t? What would the solution look like if $p_{t+1} > (1+r)p_t$ for all t? What is the condition on the path of prices necessary for an interior solution (where the owner will extract some, but not all, of the oil)?

Tips:

1. *No need to use a computer here—This equation wants you to apply your theory of dynamic programming.*
2. *Pay attention to binding constraints.*

Answer.

1. The state variable is B_t (how many barrels of oil left at time t) and the parameter is r (the constant interest rate).
2. The control variables are the price p_t (and t, the time at which to sell the barrel). And, say that b_t is the amount of barrels sold in period t.
3. Transition equation: $B' = B - b$. The amount of barrels of oil left tomorrow is the amount left today, minus the amount of barrels sold.
4. The sequence problem:

$$V_T(B) = \max_{\{(p_1,\ldots,p_T),(b_1,\ldots,b_T)\}} \sum_{t=1}^{T} \left(\frac{1}{1+r}\right)^t \pi(b_t, p_t),$$

where $\pi(b_t, p_t)$ is our profit function, which we can represent as $p_t b_t$. So we have

$$V_T(B) = \max_{\{(p_1,\ldots,p_T),(b_1,\ldots,b_T)\}} \sum_{t=1}^{T} \frac{p_t b_t}{(1+r)^t},$$

such that $B = \sum_{t=1}^{T} b_t$. The Bellman Equation would be

$$V_{T+1}(B) = \max_{\{p_1,b_1\}} \{p_1 b_1 + V_T(B - b_1)\}.$$

5. Setting up the Lagrangian:

$$\mathcal{L} = \max_{\{(p_1,\ldots,p_T),(b_1,\ldots,b_T)\}} \sum_{t=1}^{T} \frac{p_t b_t}{(1+r)^t} - \lambda_1 \left(B - \sum_{t=1}^{T} b_t \right) - \lambda_2 (p_t)$$

Taking some FOCs:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial b_1} &= \frac{p_1}{(1+r)} + \lambda_1 \implies -\lambda_1 = \frac{p_1}{(1+r)} \\ \frac{\partial \mathcal{L}}{\partial b_2} &= \frac{p_2}{(1+r)^2} + \lambda_1 \implies -\lambda_1 = \frac{p_2}{(1+r)^2} \\ &\vdots \end{aligned}$$

From here, we can find the Euler equation:

$$\frac{p_1}{(1+r)} = \frac{p_2}{(1+r)^2} \implies p_2 = (1+r)p_1.$$

Note that this relationship holds for all p_t and p_{t+1} for $t \in \{1, \ldots, T-1\}$. So in general, our Euler equation is

$$p_{t+1} = (1+r)p_t.$$

6. What would the solution of the problem look like if $p_{t+1} = p_t$ for all t? What would the solution look like if $p_{t+1} > (1+r)p_t$ for all t? What is the condition on the path of prices necessary for an interior solution (where the owner will extract some, but not all, of the oil)?

- If $p_{t+1} = p_t$ for all t, the solution would be sell a larger amount at the beginning than later on (in fact, sell all of them in the first period, since there's no intertemporal utility constraint to even out the selling of oil), since compounded interest would erode away the present-value price of barrels for large t.
- If $p_{t+1} > (1+r)p_t$, then the solution would be to sell more barrels in the future (in fact, sell all of them in the last period, for similar reasons as mentioned above), since the price mark-up effect would exceed the compounded interest effect when discounting back to present-value prices.
- The owner would extract some, but not all, of the oil (achieve an *interior solution*) if the Inada condition on the path of prices was put in place.

Exercise 2.

Question. The Neoclassical Growth Model is a workhorse model in macroeconomics. The problem for the social planner is to maximize the discounted expected utility for agents in the economy:

$$\max_{\{c\}_{t=0}^{\infty}} \mathrm{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t).$$

The resource constraint is given as:

$$y_t = c_t + i_t.$$

The law of motion for the capital stock is:

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

Output is determined by the aggregate production function:

$$y_t = z_t k_t^{\alpha}.$$

Assume that z_t is stochastic. In particular, it is an i.i.d. process distributed as $\log(z) \sim \mathcal{N}(0, \sigma_z)$.

1. What is (are) the state variables(s)?
2. What are the control variables?
3. Write down the Bellman Equation that represents this sequence problem.
4. Solve the growth model given the following parameterization (you may use VFI or PFI):

Parameter	Description	Value
$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$	CRRA utility	
γ	Coefficient of Relative Risk Aversion	0.5
β	Discount factor	0.96
δ	Rate of physical depreciation	0.05
α	Curvature of production function	0.4
σ_z	SD of productivity shocks	0.2

- Plot the value function.
- Plot the policy function for the choice of consumption.
- Plot the policy function for the choice of capital next period.

Tips:

1. *The fact that the shocks are i.i.d. makes the computation simpler. Consider integrating over a Monte Carlo simulation of the shocks to find the expected values needed.*
2. *Write functions for the utility function and the production function—and depending on your solution method, variants on these as well such as the marginal utility function.*
3. *Be careful when it's possible that infeasible values of c_t of k_t may be chosen in your solution method.*

Answer.

1. The state variable is k_t, the capital stock; the parameters are $\beta, \delta, \alpha, \sigma_z, k_t$: the discount rate, the

depreciation rate, the capital intensity, and the SD of productivity shocks.

2. The control variables are c_t and i_t: the consumption and investment.
3. The Bellman equation can be represented as

$$V_{T+1}(k_0) = \max_{c_0, i_0} \{u(c_0) + V_T(c_1)\},$$

where $c_1 = y_1 - i_1$ and $y_1 = z_1 k_1^\alpha$, and $k_1 = (1-\delta)k_0 + i_0$, so we have that

$$\begin{aligned} c &= y - i \\ &= zk^\alpha - i \\ &= zk^\alpha - (k' - (1-\delta)k). \end{aligned}$$

$$V_{T+1}(k_0) = \max_{c_0, i_0} \left\{ u(c_0) + \mathrm{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(z_1 \left((a-\delta)k_0 + i_0\right)^\alpha - i_1\right) \right\};$$

note that $V_T(c_1) = \mathrm{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$.

4. See attached.

Exercise 3.

Question. Use the same neoclassical growth model as above, but consider the case where there is serial correlation in the productivity shock. In particular, assume that z_t is given by:

$$\log(z_t) = \rho \log(z_{t-1}) + v_t$$

where $v_t \sim \mathcal{N}(0, \sigma_v)$. Let $\rho = 0.8$ and $v = 0.1$.

1. Write down the Bellman Equation that represents the planner's problem in this case.
2. Approximate the AR(1) process with a Markov chain and solve the model:

- Plot the value function for at least 3 values of the productivity shock.
- Plot the policy function for the choice of consumption for at least 3 values of the productivity shock.
- Plot the policy function for the choice of capital next period for at least 3 values of the productivity shock.

Tips:

1. *Use* `quantecon.markov.approximate` *or the* `ar1_approximate.py` *module to approximate the* AR(1) *process.*

Answer.

1. Similar to the planner's Bellman equation can be represented as (now replacing in the formulas for $u(\ )$ and $V_T(\ )$):

$$V_{T+1}(k_0) = \max_{c_0} \left\{ \frac{c_0^{1-\gamma}}{1-\gamma} + \sum_{t=1}^{T} \beta^t \frac{c_1^{1-\gamma}}{1-\gamma} \right\}.$$

2. See attached.

Exercise 4.

Question. The search and matching model of labor markets is a key model in the macroeconomic labor literature. In one version of this model, potential workers receive wage offers from a distribution of wages in each period. Potential workers must decide whether to accept and begin work at this age (and work at this age forever) or decline the offer and continue to "search" (i.e., receive wage offers from some exogenous distribution).
The potential workers seek to maximize the expected, discounted sum of earnings:

$$\mathrm{E}_0 \sum_{t=0}^{\infty} \beta^t y_t.$$

Income, y_t, is equal to w_t if employed. If unemployed, agents receive unemployment benefits b.
Assume that wage offers are distributed as $\log(w_t) \sim \mathcal{N}(\mu, \sigma)$.

1. Write down the Bellman equation representing this optimal stopping problem.
2. Solve the model, using the following parameterization:

Parameter	Description	Value
β	Rate of time preference	0.96
b	Unemployment benefits	0.05
μ	Mean of log wages	0.0
σ	SD of wage draws	0.15

- Plot the value function.
- Find the "reservation wage" for the unemployed worker (i.e., the wage that makes her indifferent between accepting the job offer and not)
- Vary b from 0.5 to 1.0 and plot the reservation wage for each value of b. How do unemployment benefits affect the reservation wage?

Answer.

1. The Bellman equation is
$$V(w) = \max\left\{\frac{w}{1-\beta}, b + \beta \mathrm{E}_{w'} V(w')\right\}.$$
2. See attached.

Code for Exercise 4.

For this problem, I am using the dynamic programming template created by Jason DeBacker.

- Let our control be
$$\{\text{work}, \text{don}'\text{t work}\} \to \text{binary}(0, 1 \text{ choice}).$$
Call this control z.
- The state is w_t and b, since we know both at the time of the decision.
- The transition equation is $w' = \mathrm{E}_0\left[\sum_{t=0}^{\infty} \beta^t \mathrm{e}^{\mathcal{N}(\mu,\sigma)}\right]$ if $z = 0$; otherwise, $w' = \mathrm{E}_0\left[\sum_{t=0}^{\infty} \beta^t b\right]$ if $z = 1$.

First let's import the relevant modules:

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

Here, we set our parameters equal to those specified in the problem:

```
β = 0.96 # rate of time preference
 b = 0.05 # unemployment benefits
 μ = 0.0 # mean of log wages
 σ = 0.15 # SD of wage draws
```

We now set up our state space grid:

```
'''
 ------------------------------------------------------------------------
 Create Grid for State Space
 ------------------------------------------------------------------------
 lb_w      = scalar, lower bound of grid
 ub_w      = scalar, upper bound of grid
 size_w    = integer, number of grid points in state space
 w_grid    = vector, size_w x 1 vector of grid points
 ------------------------------------------------------------------------
 '''
lb_w = .5
ub_w = 10
size_w = 200  # Number of grid points
w_grid = np.linspace(lb_w, ub_w, size_w)

'''
 ------------------------------------------------------------------------
 Create grid of current utility values
 ------------------------------------------------------------------------
 Y        = matrix, current income
 W        = wage, stochastically determined
 ------------------------------------------------------------------------
 '''
```

```
Y = np.zeros((size_w, size_w))
W = []
for i in range(size_w):
    W.append(np.exp(np.random.normal(μ,σ)))
W=np.array(W)
for i in range(size_w): # loop over w
    for j in range(size_w): # loop over w'
        Y[i,j] = np.max((W[i]/(1-β), b + β*w_grid[j]))

# replace 0 and negative consumption with a tiny value
# This is a way to impose non-negativity on cons
Y[Y<=0] = 1e-15

'''
------------------------------------------------------------------------
Value Function Iteration
------------------------------------------------------------------------
VFtol     = scalar, tolerance required for value function to converge
VFdist    = scalar, distance between last two value functions
VFmaxiter = integer, maximum number of iterations for value function
V         = vector, the value functions at each iteration
Vmat      = matrix, the value for each possible combination of k and k'
Vstore    = matrix, stores V at each iteration
VFiter    = integer, current iteration number
TV        = vector, the value function after applying the Bellman operator
PF        = vector, indicies of choices of k' for all k
VF        = vector, the "true" value function
------------------------------------------------------------------------
'''
VFtol = 1e-8
VFdist = 5.0
VFmaxiter = 3000
V = np.zeros(size_w)
Vmat = np.zeros((size_w, size_w))
Vstore = np.zeros((size_w, VFmaxiter))
VFiter = 1
while VFdist > VFtol and VFiter < VFmaxiter:
    for i in range(size_w): # loop over w
        for j in range(size_w): # loop over w'
            Vmat[i, j] = Y[i, j] + β * V[j]

    Vstore[:, VFiter] = V.reshape(size_w,) # store value function at each iteration for
    TV = Vmat.max(1) # apply max operator to Vmat (to get V(w))
    PF = np.argmax(Vmat, axis=1)
    VFdist = (np.absolute(V - TV)).max()  # check distance
    V = TV
    VFiter += 1
```

```
        if VFiter < VFmaxiter:
            print('Value function converged after this many iterations:', VFiter)
        else:
            print('Value function did not converge')


        VF = V # solution to the functional equation
```

```
Value function converged after this many iterations: 541
```

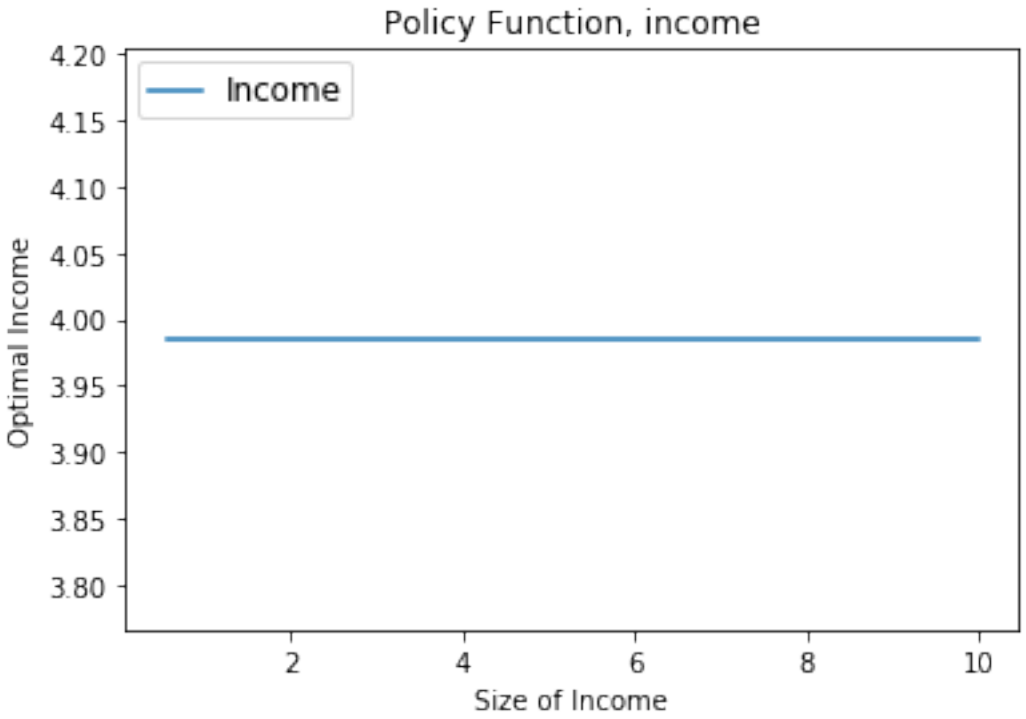
```
        '''
        --------------------------------------------------------------------------
        Find consumption and savings policy functions
        --------------------------------------------------------------------------
        optW  = vector, the optimal choice of w' for each w
        optC  = vector, the optimal choice of y' for each y
        --------------------------------------------------------------------------
        '''
        optW = w_grid[PF]
        optY = optW
```

Plotting the **value function**, we have:

```
        # Plot value function
        plt.figure()
        # plt.plot(wvec, VF)
        plt.scatter(w_grid[1:], VF[1:])
        plt.xlabel('Income')
        plt.ylabel('Value Function')
        plt.title('Value Function - income')
        plt.show()
```

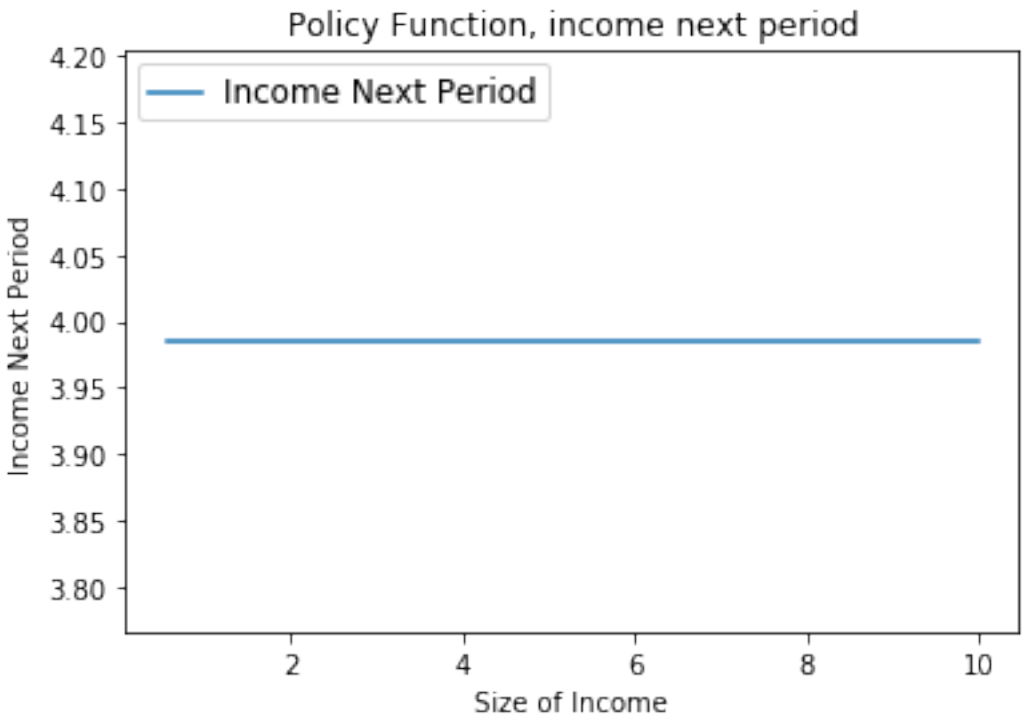

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(w_grid[1:], optY[1:], label='Income')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Income')
plt.ylabel('Optimal Income')
plt.title('Policy Function, income')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```




```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(w_grid[1:], optW[1:], label='Income Next Period')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Income')
plt.ylabel('Income Next Period')
plt.title('Policy Function, income next period')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```


Policy Function, income next period
Income Next Period
4.20
4.15
4.10
4.05
4.00
3.95
3.90
3.85
3.80
Income Next Period
2
4
6
8
10
Size of Income

Code for Exercise 3.

For this problem, I am using the dynamic programming template created by Jason DeBacker.

First let's import the relevant modules:

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

Here, we set our parameters equal to those specified in the problem:

```
gamma = 0.5 # CRRA coefficient
beta = 0.96 # discount factor
delta = 0.05 # depreciation rate
alpha = 0.4 # curvature of production function
sigma_v = 0.2 # SD of productivity shocks
rho = 0.8 # constant for the serial correlation
```

We now set up our state space grid:

```
'''
------------------------------------------------------------------------
Create Grid for State Space
------------------------------------------------------------------------
lb_k      = scalar, lower bound of capital grid
ub_k      = scalar, upper bound of capital grid
size_k    = integer, number of grid points in capital state space
k_grid    = vector, size_k x 1 vector of capital grid points
------------------------------------------------------------------------
'''
lb_k = .5
ub_k = 10
size_k = 200  # Number of grid points
k_grid = np.linspace(lb_k, ub_k, size_k)
```

Next, we construct an array `Z` by iterating an AR(1) process of stochastic shocks `size_k` times. And, we define some regularity conditions, and set the CRRA utility function based off the consumption we found.

```
'''
------------------------------------------------------------------------
Create grid of current utility values
------------------------------------------------------------------------
C        = matrix, current consumption
I        = matrix, current investment
Z        = matrix, productivity shocks
U        = matrix, current period utility value for all possible
           choices of k and k' (rows are k, columns k')
------------------------------------------------------------------------
```

```
        '''
        C = np.zeros((size_k, size_k))
        Y = np.zeros((size_k, size_k))
        I = np.zeros((size_k, size_k))
        Z = [0.1]

        for i in range(size_k):
            Z.append(np.exp(rho * np.log(Z[i]) + np.random.normal(0,sigma_v)))

        Z=np.array(Z)

        print(Z)

        for i in range(size_k): # loop over k
            for j in range(size_k): # loop over k'
                Y[i,j] = Z[i] * k_grid[i] ** alpha # production function = stoch shock * capita
                I[i,j] = k_grid[j] - (1-delta) * k_grid[i] # investment, determined by the k'=(
                C[i, j] = Y[i,j] - I[i,j] # resource constraint

        # replace 0 and negative consumption with a tiny value
        # This is a way to impose non-negativity on cons
        C[C<=0] = 1e-15
        I[I<=0] = 1e-15
        if gamma == 1:
            U = np.log(C)
        else:
            U = (C ** (1-gamma)) / (1-gamma)
        U[C<0] = -1e10
```

```
[0.1        0.16985324 0.13679666 0.21298965 0.28636382 0.49574118
 0.61502425 0.56781049 0.54970526 0.78633673 0.94826106 1.19114137
 1.23625805 1.24788345 1.44567458 1.20544063 0.96192802 1.14327621
 1.02664832 1.03023454 0.82620914 1.18951098 1.17272709 1.38074157
 1.49786175 1.28155941 1.36401833 1.43070268 1.31685919 1.61970971
 1.55180698 2.15264805 2.0981616  1.84442644 1.81937936 1.20976628
 1.53417086 1.0488133  1.24293448 1.04034409 0.90232281 0.74467291
 0.9518998  0.89443857 1.26689588 1.27676605 1.1055865  1.19091865
 1.25873721 2.19209308 2.21980366 2.15353917 1.62389491 1.80270181
 1.68507414 1.45788849 1.00114093 0.55053844 0.56707766 0.66895343
 0.79837907 0.62302573 0.66428692 0.80356333 0.82712929 0.80879731
 0.89011573 1.15578627 0.83555796 0.85857373 1.01508183 0.88638704
 1.49730572 1.26122016 1.00467495 1.04148602 0.89687673 1.09277206
 0.89631673 0.6339384  0.66332717 0.78362448 0.56516493 0.54928781
 0.60588885 0.73269247 0.78156763 1.11760161 0.92487396 0.6233298
 0.80140421 1.20404962 0.98217136 0.71192662 0.78856054 0.63710724
 0.70350367 1.39613646 1.35365287 1.9136075  1.51210256 1.57342686
 1.85177317 2.15981903 1.24543301 1.184226   0.79616779 0.81397004
 0.88054069 1.03003815 0.87945854 0.92170372 0.85746969 0.8568811
```

```
0.72295476 0.68184037 0.61165514 0.67960496 0.57488943 0.61836368
0.61042648 0.93480958 0.82272108 0.74901312 0.75673857 0.61268759
0.67293482 0.46174077 0.71970235 0.73161231 0.60464992 0.9723395
0.81378929 0.59976695 0.55755548 0.85936874 0.71183907 0.76670693
0.9336629  1.45077979 1.44654029 1.45150701 1.3635857  1.22702624
0.78899443 0.58290609 0.4243708  0.37154824 0.35918553 0.48293211
0.49789523 0.46296427 0.58991248 0.83867868 0.76587094 0.75885362
0.75402865 1.26299869 1.38877802 1.30751508 0.97395542 0.76654665
0.57076467 0.6183363  0.66991151 0.63002174 0.78803368 0.74736793
0.69326202 0.64620227 0.79994543 0.88167875 0.56117495 0.48919474
0.44060023 0.48674495 0.62955101 0.67257276 0.52012083 0.4860773
0.56063885 0.54689808 0.5664858  0.50833307 0.55333273 0.81788645
0.6819643  0.66298582 0.71061095 0.77164401 1.18285109 1.10314167
1.11208027 0.86075851 0.67663236 0.78904695 1.08323564 1.36325633
0.9649143  1.0642067  0.95604777]
```

Now, this next step is virtually the same as we covered in class, where I apply a Bellman equation $V_{T+1} = u(c) + V_T$ looped over multiple iterations, recalculation the value function many times and corresponding policy functions.

```
'''
--------------------------------------------------------------------------
Value Function Iteration
--------------------------------------------------------------------------
VFtol     = scalar, tolerance required for value function to converge
VFdist    = scalar, distance between last two value functions
VFmaxiter = integer, maximum number of iterations for value function
V         = vector, the value functions at each iteration
Vmat      = matrix, the value for each possible combination of k and k'
Vstore    = matrix, stores V at each iteration
VFiter    = integer, current iteration number
TV        = vector, the value function after applying the Bellman operator
PF        = vector, indicies of choices of k' for all k
VF        = vector, the "true" value function
--------------------------------------------------------------------------
'''
VFtol = 1e-8
VFdist = 5.0
VFmaxiter = 3000
V = np.zeros(size_k) # initial guess at value function
Vmat = np.zeros((size_k, size_k)) # initialize Vmat matrix
Vstore = np.zeros((size_k, VFmaxiter)) #initialize Vstore array
VFiter = 1
while VFdist > VFtol and VFiter < VFmaxiter:
    for i in range(size_k): # loop over k
        for j in range(size_k): # loop over k'
            Vmat[i, j] = U[i, j] + beta * V[j]
```

```
        Vstore[:, VFiter] = V.reshape(size_k,) # store value function at each iteration for
        TV = Vmat.max(1) # apply max operator to Vmat (to get V(k))
        PF = np.argmax(Vmat, axis=1)
        VFdist = (np.absolute(V - TV)).max()  # check distance
        V = TV
        VFiter += 1



    if VFiter < VFmaxiter:
        print('Value function converged after this many iterations:', VFiter)
    else:
        print('Value function did not converge')


    VF = V # solution to the functional equation
```

```
Value function converged after this many iterations: 487
```

There were 489 iterations.

This next part involves using the formula for the choice of consumption based off capital, organized in the order of the PF-selected values. I used the formula

$$\texttt{optC} = \texttt{Z}_{\texttt{PF}} \cdot \texttt{k\_grid}^{\texttt{alpha}} - (\texttt{optK} - (1 - \texttt{delta}) \cdot \texttt{k\_grid}),$$

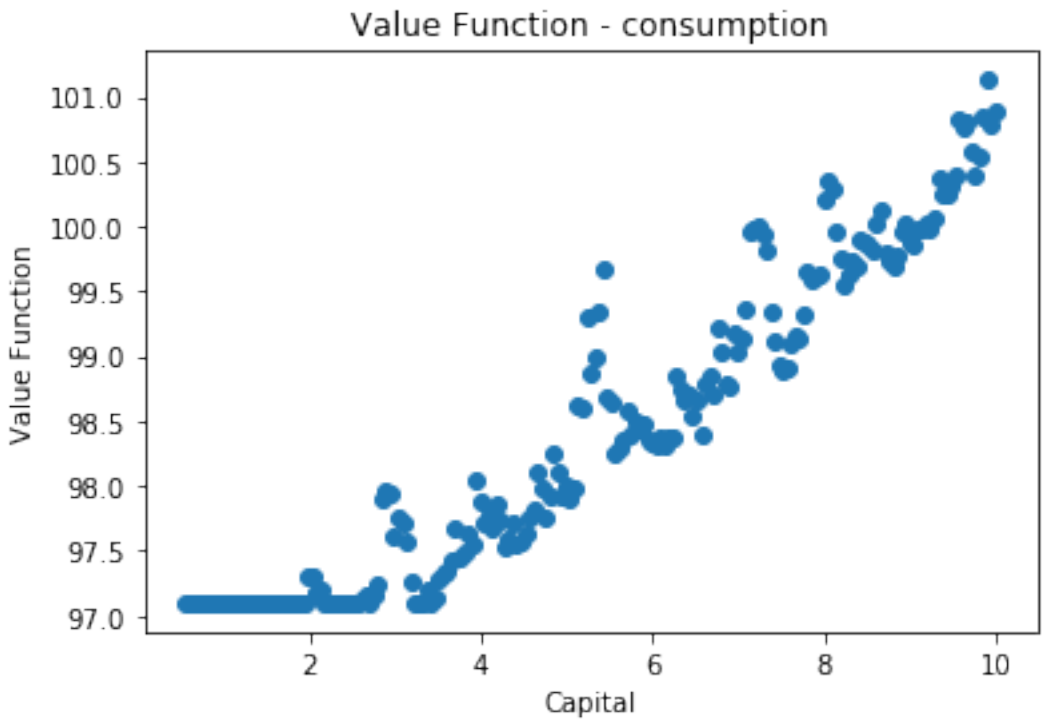
i.e.,

$$c = z \cdot k^{\alpha} - (k' - (1 - \delta) \cdot k).$$

```
    '''
    ------------------------------------------------------------------------
    Find consumption and savings policy functions
    ------------------------------------------------------------------------
    optK  = vector, the optimal choice of k' for each k
    optC  = vector, the optimal choice of c' for each c
    ------------------------------------------------------------------------
    '''
    optK = k_grid[PF] # tomorrow's optimal capital size (savings function)
    optC = Z[PF] * k_grid ** alpha - (optK - (1-delta) * k_grid) # optimal consumption, get
```

Plotting the **value function**, we have:

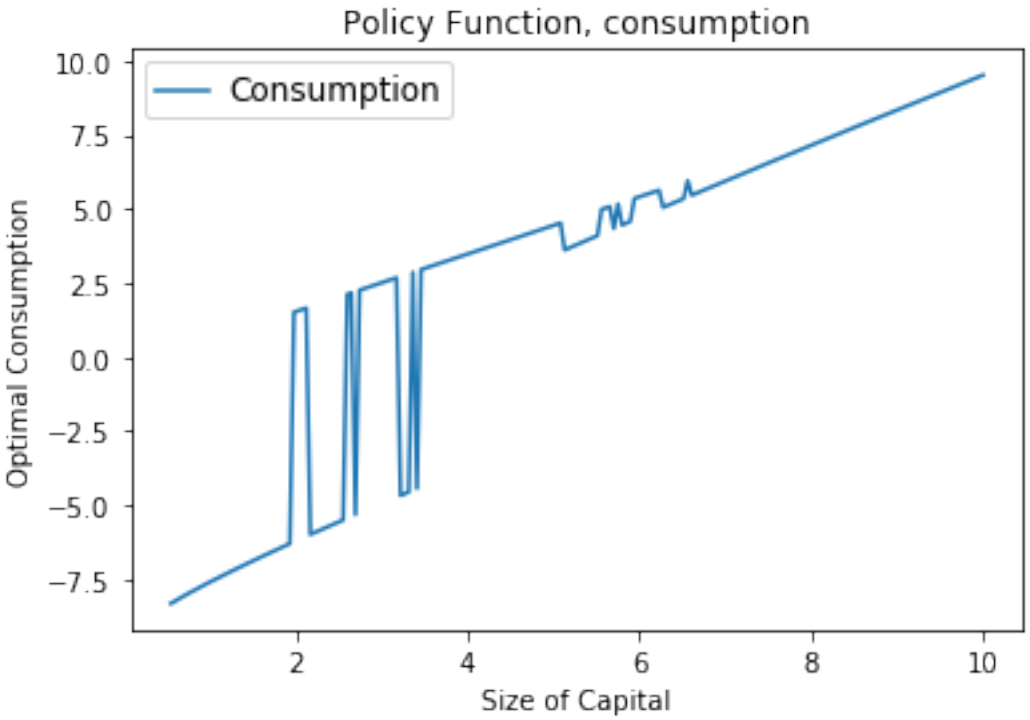
```
# Plot value function
plt.figure()
# plt.plot(wvec, VF)
plt.scatter(k_grid[1:], VF[1:])
plt.xlabel('Capital')
plt.ylabel('Value Function')
plt.title('Value Function - consumption')
plt.show()
```



Now we plot the **policy function** for the choice of consumption:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optC[1:], label='Consumption')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Optimal Consumption')
plt.title('Policy Function, consumption')
plt.show()
```

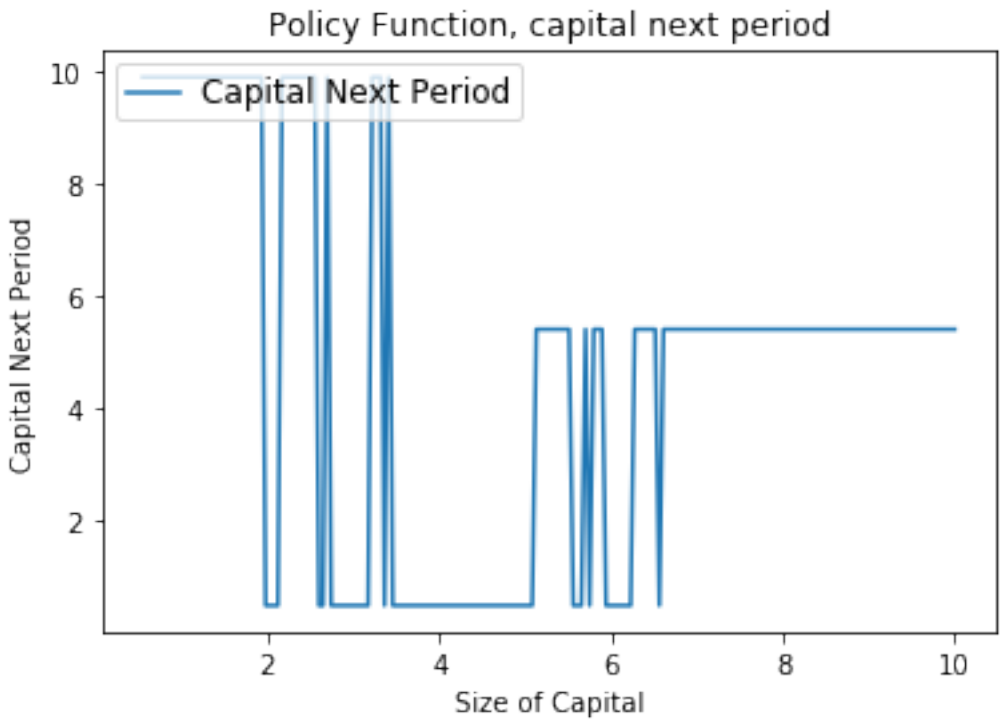
```
<Figure size 432x288 with 0 Axes>
```



Finally, we plot the **policy function** for the choice of capital next period:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optK[1:], label='Capital Next Period')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Capital Next Period')
plt.title('Policy Function, capital next period')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



Let us now repeat this process for another set of values of the productivity shock v. Here we will choose 0.3.

```
'''
--------------------------------------------------------------------------
Create grid of current utility values
--------------------------------------------------------------------------
C         = matrix, current consumption
I         = matrix, current investment
Z         = matrix, productivity shocks
U         = matrix, current period utility value for all possible
            choices of k and k' (rows are k, columns k')
--------------------------------------------------------------------------
'''
C = np.zeros((size_k, size_k))
Y = np.zeros((size_k, size_k))
I = np.zeros((size_k, size_k))
Z = [0.3]

for i in range(size_k):
    Z.append(np.exp(rho * np.log(Z[i]) + np.random.normal(0,sigma_v)))

Z=np.array(Z)

print(Z)
```

```
        for i in range(size_k): # loop over k
            for j in range(size_k): # loop over k'
                Y[i,j] = Z[i] * k_grid[i] ** alpha # production function = stoch shock * capita
                I[i,j] = k_grid[j] - (1-delta) * k_grid[i] # investment, determined by the k'=(
                C[i, j] = Y[i,j] - I[i,j] # resource constraint

        # replace 0 and negative consumption with a tiny value
        # This is a way to impose non-negativity on cons
        C[C<=0] = 1e-15
        I[I<=0] = 1e-15
        if gamma == 1:
            U = np.log(C)
        else:
            U = (C ** (1-gamma)) / (1-gamma)
        U[C<0] = -1e10
```

```
[0.3        0.502852   0.40152731 0.51872779 0.58368681 0.69351571
 0.77866069 0.86236139 0.77982512 0.90988453 0.94947247 0.71700475
 0.67642738 0.80508959 0.79100585 0.95015379 0.98572508 1.14369339
 1.04880528 1.14462566 1.03843884 1.29742994 1.37393317 1.19316756
 1.24044732 0.94116163 0.97869911 0.86040791 1.17624945 1.01877691
 1.09122051 1.28796738 1.46632095 1.55235464 1.57549608 1.63169145
 1.2561653  1.24424985 1.41601331 1.5333454  1.21406735 1.6032606
 1.35519777 1.53182595 1.38661129 1.75048778 2.05493551 1.55394497
 1.63283724 1.41409016 1.29237714 1.29510349 1.08103263 0.89642952
 1.02414978 1.06637178 1.24884222 1.35025141 1.18995312 0.94469111
 1.25441413 1.34924994 1.38520385 1.18917627 1.46225896 1.51778054
 1.06434776 1.59123375 1.16616956 1.17892867 1.33702892 1.12861115
 1.03702077 1.86733248 2.04203503 1.32994212 1.12036848 1.13612698
 1.30315795 1.42902784 2.07128318 1.9557584  1.87601445 2.3745277
 2.10543549 1.87635263 1.4554365  1.39489283 1.00028617 1.34034028
 0.88696009 0.92849955 1.16431061 0.78828507 0.83874827 0.78759996
 0.90769492 0.70400777 0.64563095 0.98647146 0.97796053 0.83057672
 0.91389636 0.87465189 0.57635652 0.47705775 0.61006851 0.59648887
 0.76128852 0.83264068 0.79795721 0.84542371 1.11260895 1.29438733
 1.04902379 0.83749621 0.81915891 0.79268169 0.97223519 0.94001734
 0.71874166 0.58439106 0.7562559  0.87348327 1.04846292 0.89144895
 0.9354715  1.01565531 0.79585245 0.77537921 0.70425676 0.57294932
 0.59027515 0.59042195 0.87578228 1.46823864 1.4058241  1.74041544
 1.50200398 1.66914261 2.1377505  1.29337412 1.1635818  1.08057059
 1.18403812 1.45641394 1.90131359 2.54617922 2.36887995 1.72217763
 1.71812526 1.52127354 1.36547517 1.09898167 1.41134166 1.41370149
 1.46353745 1.36789668 1.85387082 1.50403039 1.91302911 1.44615317
 0.98361721 1.16531697 1.35328818 1.35629408 1.79835786 2.12522213
 2.01433324 1.24479028 1.17391529 1.4042787  1.06989442 0.55333957
 0.6978953  0.57084817 0.44518452 0.54345608 0.67784863 1.13314228
 0.86217621 1.07539662 1.24795237 1.19645444 1.41384307 1.89681504
```

```
1.46937244 1.48816444 1.12770593 1.05146944 1.02558375 1.20530012
1.54941098 1.13978007 1.22097342 1.44125761 0.88574001 1.03838156
0.73341579 0.81747917 0.81426637]
```

Now, this next step is virtually the same as we covered in class, where I apply a Bellman equation $V_{T+1} = u(c) + V_T$ looped over multiple iterations, recalculation the value function many times and corresponding policy functions.

```
'''
--------------------------------------------------------------------------
Value Function Iteration
--------------------------------------------------------------------------
VFtol     = scalar, tolerance required for value function to converge
VFdist    = scalar, distance between last two value functions
VFmaxiter = integer, maximum number of iterations for value function
V         = vector, the value functions at each iteration
Vmat      = matrix, the value for each possible combination of k and k'
Vstore    = matrix, stores V at each iteration
VFiter    = integer, current iteration number
TV        = vector, the value function after applying the Bellman operator
PF        = vector, indicies of choices of k' for all k
VF        = vector, the "true" value function
--------------------------------------------------------------------------
'''
VFtol = 1e-8
VFdist = 5.0
VFmaxiter = 3000
V = np.zeros(size_k) # initial guess at value function
Vmat = np.zeros((size_k, size_k)) # initialize Vmat matrix
Vstore = np.zeros((size_k, VFmaxiter)) #initialize Vstore array
VFiter = 1
while VFdist > VFtol and VFiter < VFmaxiter:
    for i in range(size_k): # loop over k
        for j in range(size_k): # loop over k'
            Vmat[i, j] = U[i, j] + beta * V[j]

    Vstore[:, VFiter] = V.reshape(size_k,) # store value function at each iteration for
    TV = Vmat.max(1) # apply max operator to Vmat (to get V(k))
    PF = np.argmax(Vmat, axis=1)
    VFdist = (np.absolute(V - TV)).max()  # check distance
    V = TV
    VFiter += 1



if VFiter < VFmaxiter:
    print('Value function converged after this many iterations:', VFiter)
```

```
        else:
            print('Value function did not converge')


        VF = V # solution to the functional equation
```

```
Value function converged after this many iterations: 491
```

There were 489 iterations.

This next part involves using the formula for the choice of consumption based off capital, organized in the order of the PF-selected values. I used the formula

$$\texttt{optC} = \texttt{Z}_{\texttt{PF}} \cdot \texttt{k\_grid}^{\texttt{alpha}} - (\texttt{optK} - (1 - \texttt{delta}) \cdot \texttt{k\_grid}),$$

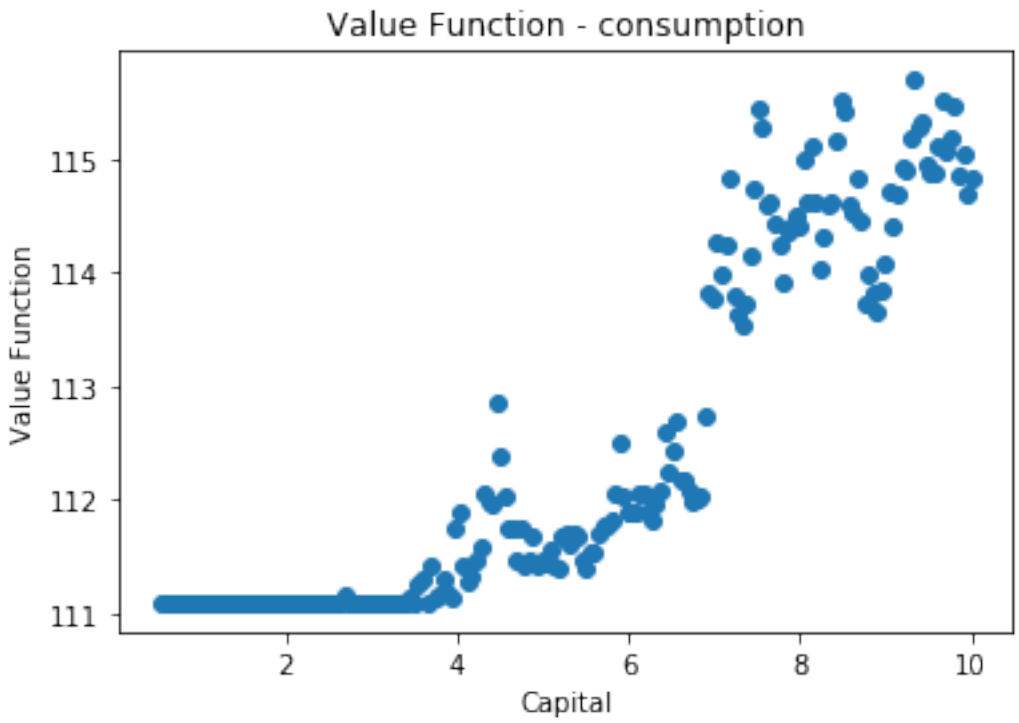
i.e.,

$$c = z \cdot k^{\alpha} - (k' - (1 - \delta) \cdot k).$$

```
        '''
        ------------------------------------------------------------------------
        Find consumption and savings policy functions
        ------------------------------------------------------------------------
        optK  = vector, the optimal choice of k' for each k
        optC  = vector, the optimal choice of c' for each c
        ------------------------------------------------------------------------
        '''
        optK = k_grid[PF] # tomorrow's optimal capital size (savings function)
        optC = Z[PF] * k_grid ** alpha - (optK - (1-delta) * k_grid) # optimal consumption, get
```

Plotting the **value function**, we have:

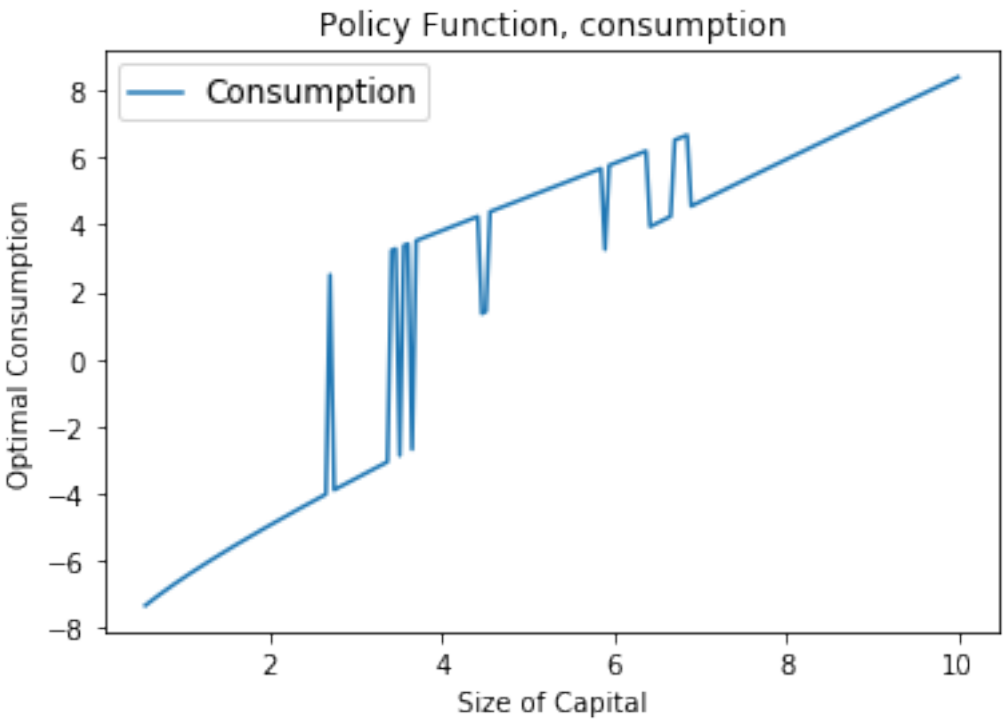
```
        # Plot value function
        plt.figure()
        # plt.plot(wvec, VF)
        plt.scatter(k_grid[1:], VF[1:])
        plt.xlabel('Capital')
        plt.ylabel('Value Function')
        plt.title('Value Function - consumption')
        plt.show()
```



Now we plot the **policy function** for the choice of consumption:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optC[1:], label='Consumption')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Optimal Consumption')
plt.title('Policy Function, consumption')
plt.show()
```

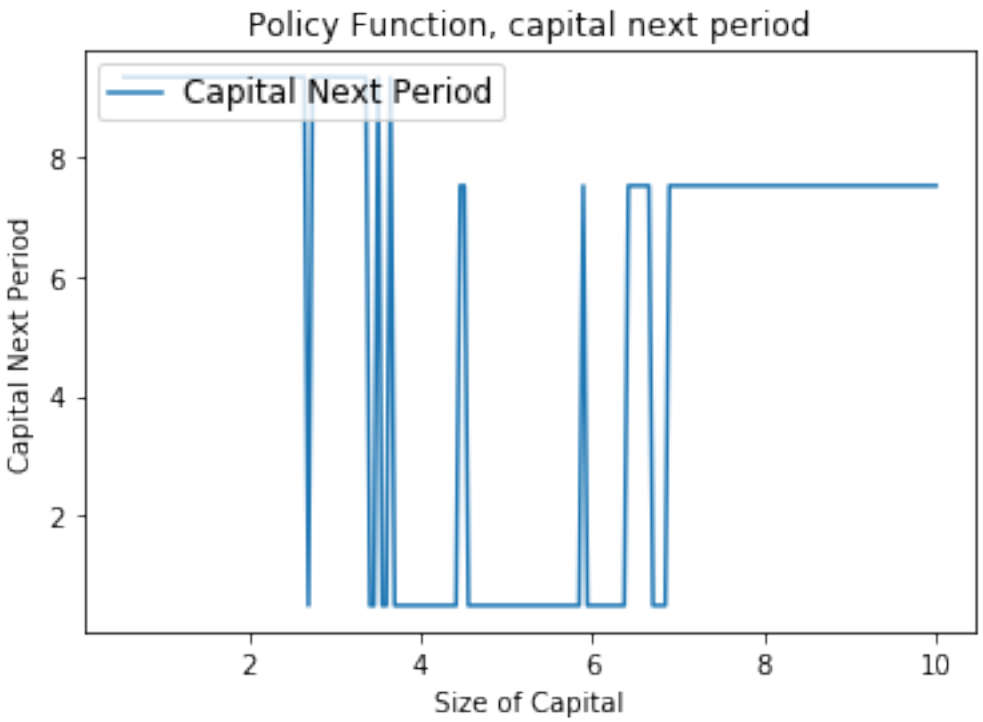
```
<Figure size 432x288 with 0 Axes>
```



Finally, we plot the **policy function** for the choice of capital next period:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optK[1:], label='Capital Next Period')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Capital Next Period')
plt.title('Policy Function, capital next period')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



Finally, Let us now repeat this process for one more set of values of the productivity shock v. Here we will choose 0.5.

```
'''
--------------------------------------------------------------------------
Create grid of current utility values
--------------------------------------------------------------------------
C         = matrix, current consumption
I         = matrix, current investment
Z         = matrix, productivity shocks
U         = matrix, current period utility value for all possible
            choices of k and k' (rows are k, columns k')
--------------------------------------------------------------------------
'''
C = np.zeros((size_k, size_k))
Y = np.zeros((size_k, size_k))
I = np.zeros((size_k, size_k))
Z = [0.5]

for i in range(size_k):
    Z.append(np.exp(rho * np.log(Z[i]) + np.random.normal(0,sigma_v)))

Z=np.array(Z)

print(Z)
```

```
        for i in range(size_k): # loop over k
            for j in range(size_k): # loop over k'
                Y[i,j] = Z[i] * k_grid[i] ** alpha # production function = stoch shock * capita
                I[i,j] = k_grid[j] - (1-delta) * k_grid[i] # investment, determined by the k'=(
                C[i, j] = Y[i,j] - I[i,j] # resource constraint

        # replace 0 and negative consumption with a tiny value
        # This is a way to impose non-negativity on cons
        C[C<=0] = 1e-15
        I[I<=0] = 1e-15
        if gamma == 1:
            U = np.log(C)
        else:
            U = (C ** (1-gamma)) / (1-gamma)
        U[C<0] = -1e10
```

```
[0.5        0.65268506 0.77633022 0.82446971 0.83397983 0.76672058
 0.857096   0.95315323 0.83605226 0.60294495 0.85269194 1.04259891
 0.88593523 0.70793405 0.63804864 0.67539487 0.68718868 0.63302556
 0.7428862  0.6903394  0.6196295  0.73075354 0.58973006 0.65676472
 0.65853808 0.70031422 0.82522524 0.99270302 1.20172153 1.72188884
 1.30041714 1.61834097 1.95953047 1.46381399 1.8733549  1.54370829
 1.54390833 1.6389801  1.25689501 1.81644731 1.69676941 1.69147037
 1.27659269 1.13551139 0.8067223  0.83182676 1.01695384 0.8156523
 0.99711197 0.78199372 0.572289   0.67690704 0.57722067 0.66213086
 0.59740702 0.77261507 0.73613974 0.4858471  0.558191   0.50529337
 0.54641178 0.70741176 0.65100006 0.89721371 1.17976144 1.40573414
 1.3412947  1.69297674 1.03318987 0.96174877 1.09466365 1.47021524
 1.52730844 1.66121068 2.21764779 2.36867832 2.53366792 2.01118982
 1.96768018 1.70539922 1.33003597 1.35968574 1.32865265 1.29361709
 1.18805287 1.11141256 1.4359761  1.65485126 0.94124843 1.01405323
 0.83555631 0.74098468 0.82382824 1.09773605 1.01688634 1.06049569
 0.78156699 0.75363169 1.17365217 1.19796354 1.60145056 1.15354114
 0.78870755 0.55160051 0.76010335 0.84891253 0.87762949 0.93057849
 0.71249996 0.81180691 0.79068698 1.16232594 1.5671783  1.38545646
 1.18176499 1.74306962 1.08644386 1.31060275 1.4507002  1.51393706
 1.13822828 0.68811611 0.71434236 0.73111968 0.65917487 0.59759103
 0.88170762 0.76967349 1.07913738 1.10517938 1.19626736 1.03387914
 0.94929938 0.61468271 0.66574311 0.66370436 0.71022452 0.74187253
 1.2374162  0.49456429 0.73826466 0.6564686  0.55178383 0.56364064
 0.80546493 0.94847988 1.00021973 1.3539668  1.77448586 1.44068971
 1.28940882 1.04466048 0.97980461 1.29474544 1.26255379 1.25847099
 1.19130585 0.79552735 0.83658489 0.74284037 0.58713532 0.75661723
 0.7218639  0.72882565 0.62332868 0.73612623 0.79579306 0.88401768
 0.86615893 1.39688145 1.3485901  1.58354425 2.31390613 2.45197923
 1.91304502 1.57650998 1.04009769 1.19646258 1.04517031 0.93834099
 0.89609384 1.0533015  0.93721761 1.06769642 1.06768139 0.97308936
```

```
1.15280845 1.2936793  1.0716797  0.86009577 0.81464883 0.65836559
0.64142715 0.62296904 0.49443235 0.37486781 0.32496543 0.43211695
0.40345643 0.49454743 0.66062145]
```

Now, this next step is virtually the same as we covered in class, where I apply a Bellman equation $V_{T+1} = u(c) + V_T$ looped over multiple iterations, recalculation the value function many times and corresponding policy functions.

```
'''
--------------------------------------------------------------------------
Value Function Iteration
--------------------------------------------------------------------------
VFtol     = scalar, tolerance required for value function to converge
VFdist    = scalar, distance between last two value functions
VFmaxiter = integer, maximum number of iterations for value function
V         = vector, the value functions at each iteration
Vmat      = matrix, the value for each possible combination of k and k'
Vstore    = matrix, stores V at each iteration
VFiter    = integer, current iteration number
TV        = vector, the value function after applying the Bellman operator
PF        = vector, indicies of choices of k' for all k
VF        = vector, the "true" value function
--------------------------------------------------------------------------
'''
VFtol = 1e-8
VFdist = 5.0
VFmaxiter = 3000
V = np.zeros(size_k) # initial guess at value function
Vmat = np.zeros((size_k, size_k)) # initialize Vmat matrix
Vstore = np.zeros((size_k, VFmaxiter)) #initialize Vstore array
VFiter = 1
while VFdist > VFtol and VFiter < VFmaxiter:
    for i in range(size_k): # loop over k
        for j in range(size_k): # loop over k'
            Vmat[i, j] = U[i, j] + beta * V[j]

    Vstore[:, VFiter] = V.reshape(size_k,) # store value function at each iteration for
    TV = Vmat.max(1) # apply max operator to Vmat (to get V(k))
    PF = np.argmax(Vmat, axis=1)
    VFdist = (np.absolute(V - TV)).max()  # check distance
    V = TV
    VFiter += 1



if VFiter < VFmaxiter:
    print('Value function converged after this many iterations:', VFiter)
```

```
        else:
            print('Value function did not converge')


        VF = V # solution to the functional equation
```

```
Value function converged after this many iterations: 491
```

There were 489 iterations.

This next part involves using the formula for the choice of consumption based off capital, organized in the order of the PF-selected values. I used the formula

$$\texttt{optC} = \texttt{Z}_{\texttt{PF}} \cdot \texttt{k\_grid}^{\texttt{alpha}} - (\texttt{optK} - (1 - \texttt{delta}) \cdot \texttt{k\_grid}),$$

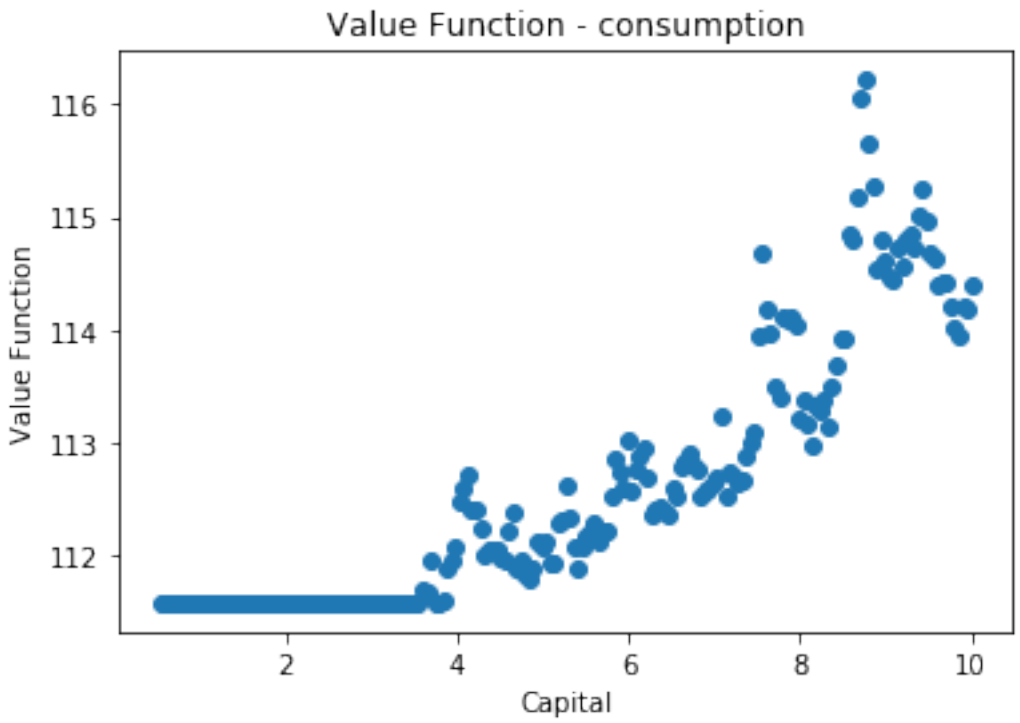
i.e.,

$$c = z \cdot k^{\alpha} - (k' - (1 - \delta) \cdot k).$$

```
        '''
        ------------------------------------------------------------------------
        Find consumption and savings policy functions
        ------------------------------------------------------------------------
        optK  = vector, the optimal choice of k' for each k
        optC  = vector, the optimal choice of c' for each c
        ------------------------------------------------------------------------
        '''
        optK = k_grid[PF] # tomorrow's optimal capital size (savings function)
        optC = Z[PF] * k_grid ** alpha - (optK - (1-delta) * k_grid) # optimal consumption, get
```

Plotting the **value function**, we have:

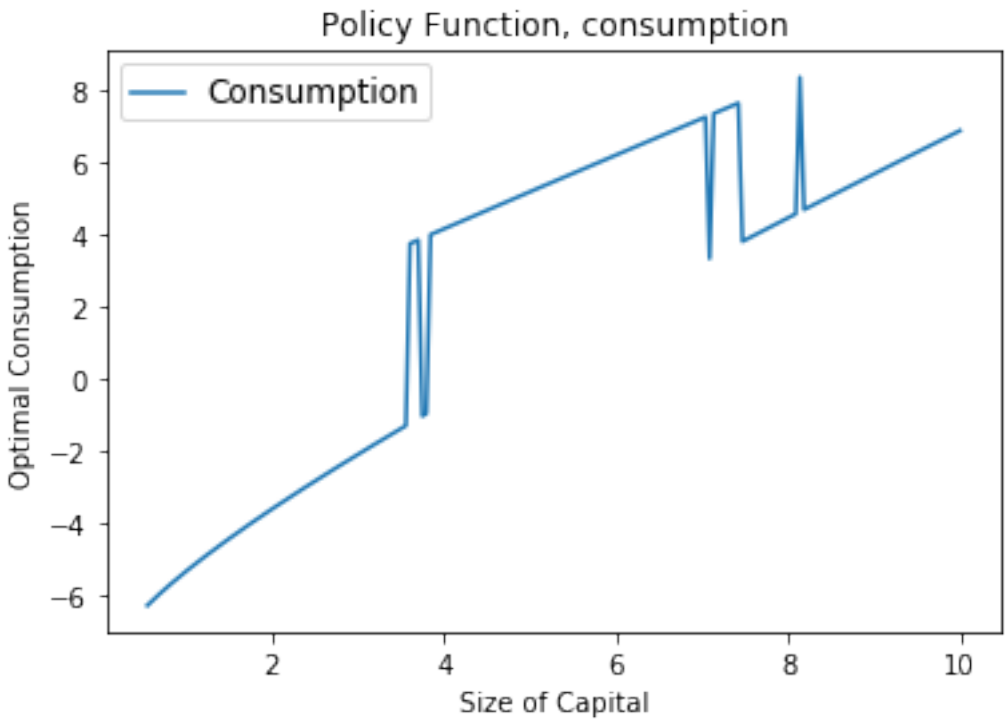
```
        # Plot value function
        plt.figure()
        # plt.plot(wvec, VF)
        plt.scatter(k_grid[1:], VF[1:])
        plt.xlabel('Capital')
        plt.ylabel('Value Function')
        plt.title('Value Function - consumption')
        plt.show()
```



Now we plot the **policy function** for the choice of consumption:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optC[1:], label='Consumption')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Optimal Consumption')
plt.title('Policy Function, consumption')
plt.show()
```

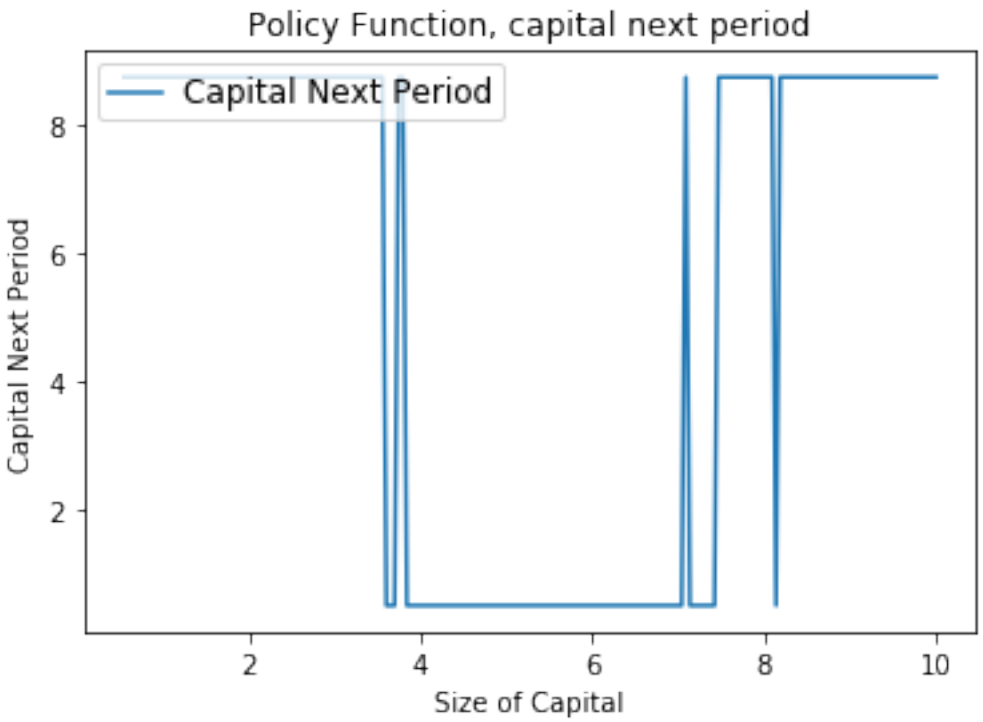
```
<Figure size 432x288 with 0 Axes>
```



Finally, we plot the **policy function** for the choice of capital next period:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optK[1:], label='Capital Next Period')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Capital Next Period')
plt.title('Policy Function, capital next period')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```


Policy Function, capital next period
Capital Next Period
Capital Next Period
Size of Capital
2
4
6
8
10

Code for Exercise 2.

For this problem, I am using the dynamic programming template created by Jason DeBacker.

First let's import the relevant modules:

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

Here, we set our parameters equal to those specified in the problem:

```
gamma = 0.5 # CRRA coefficient
beta = 0.96 # discount factor
delta = 0.05 # depreciation rate
alpha = 0.4 # curvature of production function
sigma_z = 0.2 # SD of productivity shocks
```

We now set up our state space grid:

```
'''
------------------------------------------------------------------------
Create Grid for State Space
------------------------------------------------------------------------
lb_k      = scalar, lower bound of capital grid
ub_k      = scalar, upper bound of capital grid
size_k    = integer, number of grid points in capital state space
k_grid    = vector, size_k x 1 vector of capital grid points
------------------------------------------------------------------------
'''
lb_k = .5
ub_k = 10
size_k = 200  # Number of grid points
k_grid = np.linspace(lb_k, ub_k, size_k)
```

Next, we run a Monte Carlo simulation with `numtrials` = `500` trials, each time setting matrices C, Y, and I to zeroes, and Z to an empty list. Each iteration, we re-populate Z with $\mathcal{N}\left(0, \sigma_z^2\right)$-distributed random variables, and then populate the C, Y, I matrices with their relevant calculations, based off the `k_grid` values. Next, we take the sample average over all the Monte Carlo simulations, define some regularity conditions, and set the CRRA utility function based off the consumption we found.

```
'''
------------------------------------------------------------------------
Create grid of current utility values
------------------------------------------------------------------------
C        = matrix, current consumption
I        = matrix, current investment
Z        = matrix, productivity shocks
U        = matrix, current period utility value for all possible
```

```
            choices of k and k' (rows are k, columns k')
numtrials= number of trials in Monte Carlo sim
allCs    = list of all the C's for numtrials different trials with different shocks
allIs    = analogous to allCs but for the I's
------------------------------------------------------------------------
'''
allCs = []
allIs = []
numtrials = 500
for n in range(numtrials):
    C = np.zeros((size_k, size_k))
    Y = np.zeros((size_k, size_k))
    I = np.zeros((size_k, size_k))
    Z = []
    for i in range(size_k):
        Z.append(np.exp(np.random.normal(0,sigma_z))) # productivity shocks, iid normal
    Z=np.array(Z)
    for i in range(size_k): # loop over k
        for j in range(size_k): # loop over k'
            Y[i,j] = Z[i] * k_grid[i] ** alpha # production function = stoch shock * cap
            I[i,j] = k_grid[j] - (1-delta) * k_grid[i] # investment, determined by the k
            C[i, j] = Y[i,j] - I[i,j] # resource constraint
    allCs.append(C)
    allIs.append(I)

C = np.zeros((size_k,size_k))
I = np.zeros((size_k,size_k))
for i in range(size_k):
        for j in range(size_k):
            avgCs = 0
            avgIs = 0
            for n in range(numtrials):
                avgCs += allCs[n][i,j]
                avgIs += allIs[n][i,j]
            avgCs /= numtrials
            avgIs /= numtrials
            C[i,j] = avgCs
            I[i,j] = avgIs

# replace 0 and negative consumption with a tiny value
# This is a way to impose non-negativity on cons
C[C<=0] = 1e-15
I[I<=0] = 1e-15
if gamma == 1:
    U = np.log(C)
else:
    U = (C ** (1-gamma)) / (1-gamma)
U[C<0] = -1e10
```

Now, this next step is virtually the same as we covered in class, where I apply a Bellman equation $V_{T+1}(k_0) = \max_{c_0,i_0}\{u(c_0)+V_T(c_1)\}$ (where $V_T(c_1)$ is already multiplied by β's) looped over multiple iterations, recalculation the value function many times and corresponding policy functions.

```
'''
------------------------------------------------------------------------
Value Function Iteration
------------------------------------------------------------------------
VFtol     = scalar, tolerance required for value function to converge
VFdist    = scalar, distance between last two value functions
VFmaxiter = integer, maximum number of iterations for value function
V         = vector, the value functions at each iteration
Vmat      = matrix, the value for each possible combination of k and k'
Vstore    = matrix, stores V at each iteration
VFiter    = integer, current iteration number
TV        = vector, the value function after applying the Bellman operator
PF        = vector, indicies of choices of k' for all k
VF        = vector, the "true" value function
------------------------------------------------------------------------
'''
VFtol = 1e-8
VFdist = 5.0
VFmaxiter = 3000
V = np.zeros(size_k) # initial guess at value function
Vmat = np.zeros((size_k, size_k)) # initialize Vmat matrix
Vstore = np.zeros((size_k, VFmaxiter)) #initialize Vstore array
VFiter = 1
while VFdist > VFtol and VFiter < VFmaxiter:
    for i in range(size_k): # loop over k
        for j in range(size_k): # loop over k'
            Vmat[i, j] = U[i, j] + beta * V[j]

    Vstore[:, VFiter] = V.reshape(size_k,) # store value function at each iteration for
    TV = Vmat.max(1) # apply max operator to Vmat (to get V(k))
    PF = np.argmax(Vmat, axis=1)
    VFdist = (np.absolute(V - TV)).max()  # check distance
    V = TV
    VFiter += 1



if VFiter < VFmaxiter:
    print('Value function converged after this many iterations:', VFiter)
else:
    print('Value function did not converge')


VF = V # solution to the functional equation
```

```
Value function converged after this many iterations: 484
```

There were 484 iterations.

This next part involves using the formula for the choice of consumption based off capital, organized in the order of the PF-selected values. I used the formula

$$\texttt{optC} = \texttt{Z}_{\texttt{PF}} \cdot \texttt{k\_grid}^{\texttt{alpha}} - (\texttt{optK} - (1 - \texttt{delta}) \cdot \texttt{k\_grid}),$$

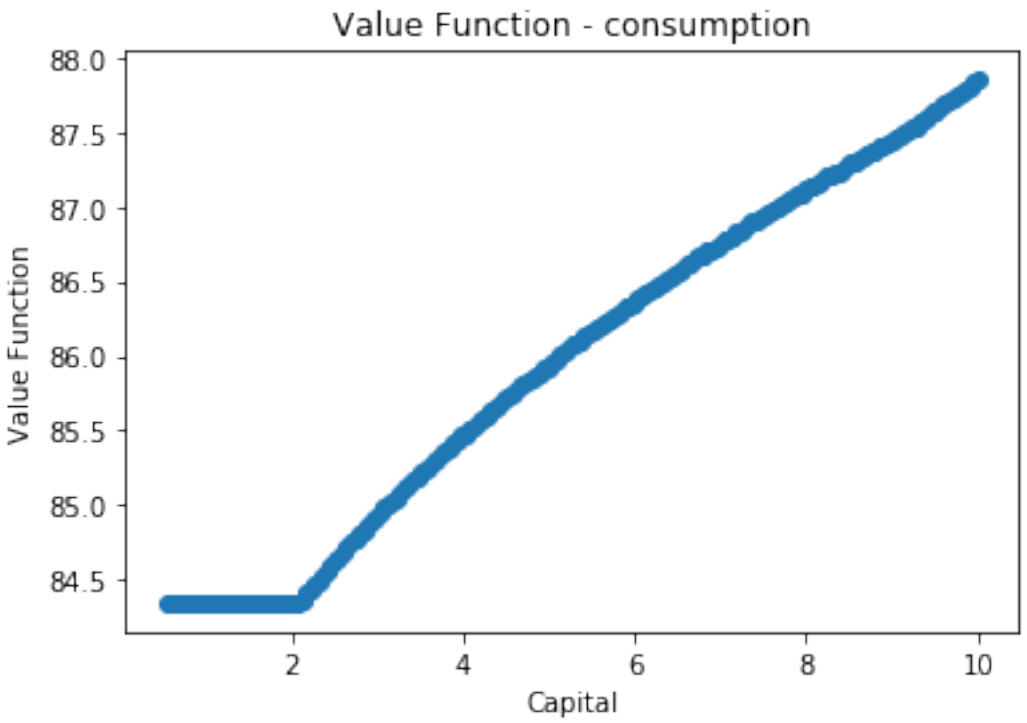
i.e.,

$$c = z \cdot k^{\alpha} - (k' - (1 - \delta) \cdot k).$$

```
'''
--------------------------------------------------------------------------
Find consumption and savings policy functions
--------------------------------------------------------------------------
optK  = vector, the optimal choice of k' for each k
optC  = vector, the optimal choice of c' for each c
--------------------------------------------------------------------------
'''
optK = k_grid[PF] # tomorrow's optimal capital size (savings function)
optC = Z[PF] * k_grid ** alpha - (optK - (1-delta) * k_grid) # optimal consumption, get
```

Plotting the **value function**, we have:

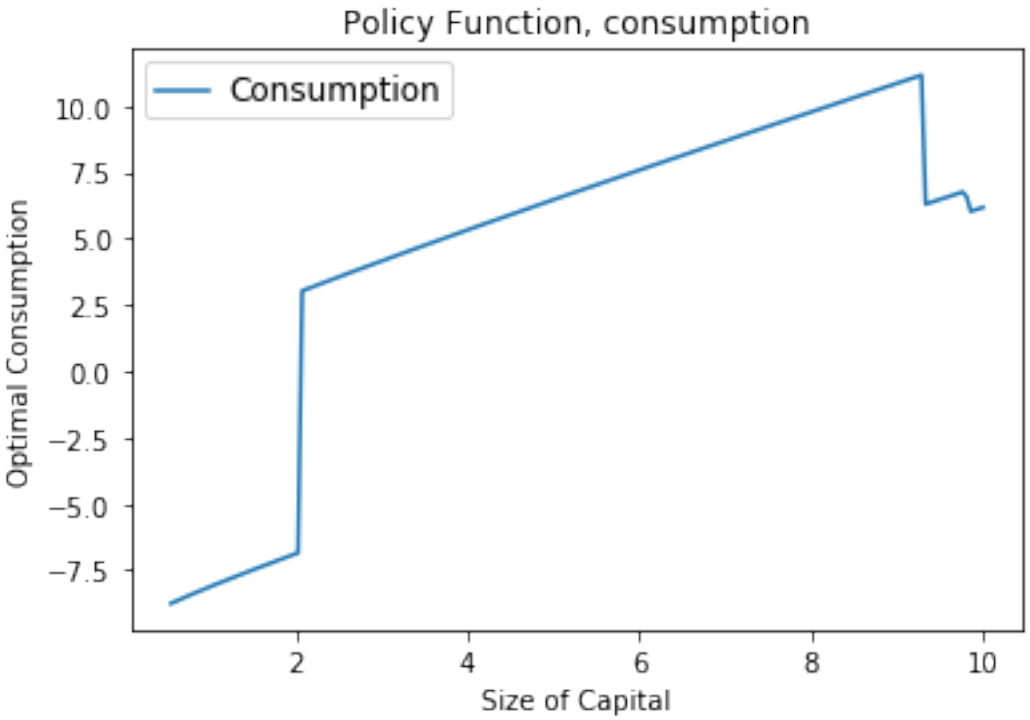
```
# Plot value function
plt.figure()
# plt.plot(wvec, VF)
plt.scatter(k_grid[1:], VF[1:])
plt.xlabel('Capital')
plt.ylabel('Value Function')
plt.title('Value Function - consumption')
plt.show()
```



Now we plot the **policy function** for the choice of consumption:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optC[1:], label='Consumption')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Optimal Consumption')
plt.title('Policy Function, consumption')
plt.show()
```

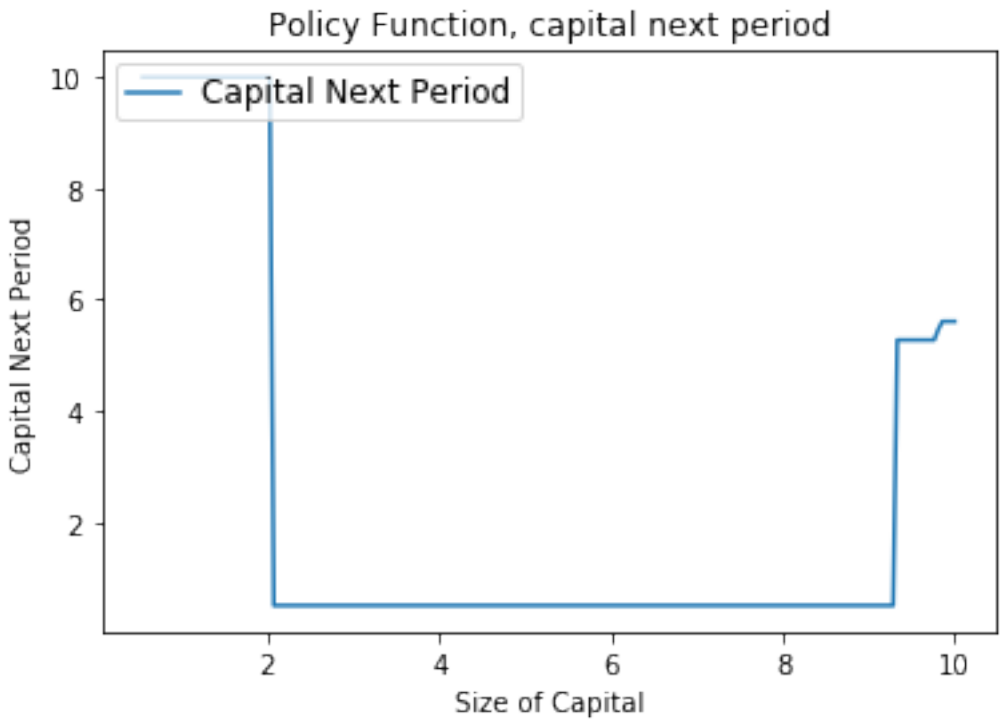
```
<Figure size 432x288 with 0 Axes>
```



Finally, we plot the **policy function** for the choice of capital next period:

```
#Plot optimal consumption rule as a function of capital size
plt.figure()
fig, ax = plt.subplots()
ax.plot(k_grid[1:], optK[1:], label='Capital Next Period')
# Now add the legend with some customizations.
legend = ax.legend(loc='upper left', shadow=False)
# Set the fontsize
for label in legend.get_texts():
    label.set_fontsize('large')
for label in legend.get_lines():
    label.set_linewidth(1.5)  # the legend line width
plt.xlabel('Size of Capital')
plt.ylabel('Capital Next Period')
plt.title('Policy Function, capital next period')
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```


Policy Function, capital next period
Capital Next Period
Capital Next Period
Size of Capital
10
8
6
4
2
2
4
6
8
10