Computational Economics: Problem Set 1

S M Sajid Al Sanai

May 1, 2019

Contents

1 Infinite Horizon Ramsey Model

1.1 Value Function Approximation

1.1.1 Bellman Equation

Given our Maximisation Problem,

$$max_{\{K_t\}_{t=1}^{\infty}} \sum_{t=0}^{\infty} \beta^t u(C_t) \tag{1}$$

Where our Utility Function is specified,

$$u(C_t) = \ln(C_t) \tag{2}$$

Where our Production Function is specified,

$$F(K_t) = AK_t^{\alpha} \tag{3}$$

Given K_0 and Non-Negativity Constraints,

$$C_t, K_{t+1} \geq 0 \tag{4}$$

Given Law of Motion of Capital,

$$I_t = K_{t+1} - (1-\delta)K_t \tag{5}$$

Given Budget Constraint limited by the Production Function,

$$F(K_t) \geq C_t + I_t \tag{6}$$

Derive our Consumption and summarise our Production,

$$C_t = F(K_t) - K_{t+1} + (1-\delta)K_t \tag{7}$$

$$f(K_t) = F(K_t) + (1-\delta)K_t \tag{8}$$

$$\implies C_t = f(K_t) - K_{t+1} \tag{9}$$

Substituting Consumption into our Maximisation Problem allows us to derive our Bellman Equation,

$$V(K_t) = max_{\{K_{t+1}\}} u(f(K_t) - K_{t+1}) + \beta V(K_{t+1}) \tag{10}$$

$$V(K_t) = max_{\{K_{t+1}\}} \ln(AK_t^{\alpha} - (1-\delta)K_t - K_{t+1}) + \beta V(K_{t+1}) \tag{11}$$

1.1.2 Value Function Iteration

Value Function Iteration conducted over the discretised space for Capital assuming $\beta = 0.6$, $A = 20$, $\alpha = 0.3$, and $\delta = 0.5$ yields the following graph visualising output,

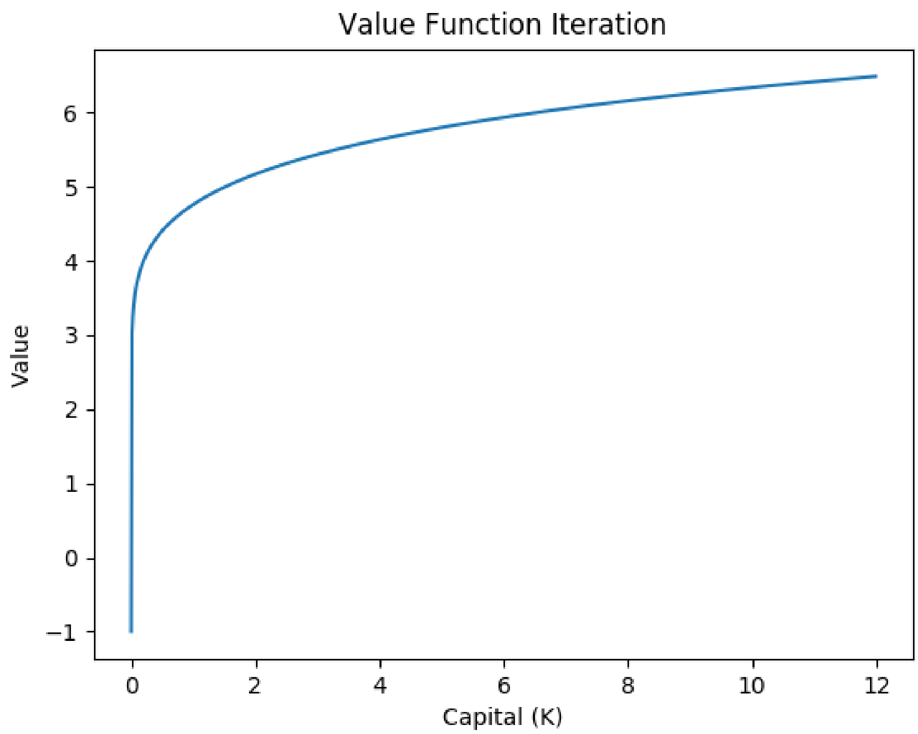

Figure 1: Value Function Iteration over Discretised Capital

1.2 Customised Value Function Approximation

1.2.1 Using Iteration

Value Function Iteration conducted over the discretised space for Capital assuming $\beta = 0.6,\ A = 1,\ \alpha = 0.3$, and $\delta = 1$ yields the following graph visualising output,

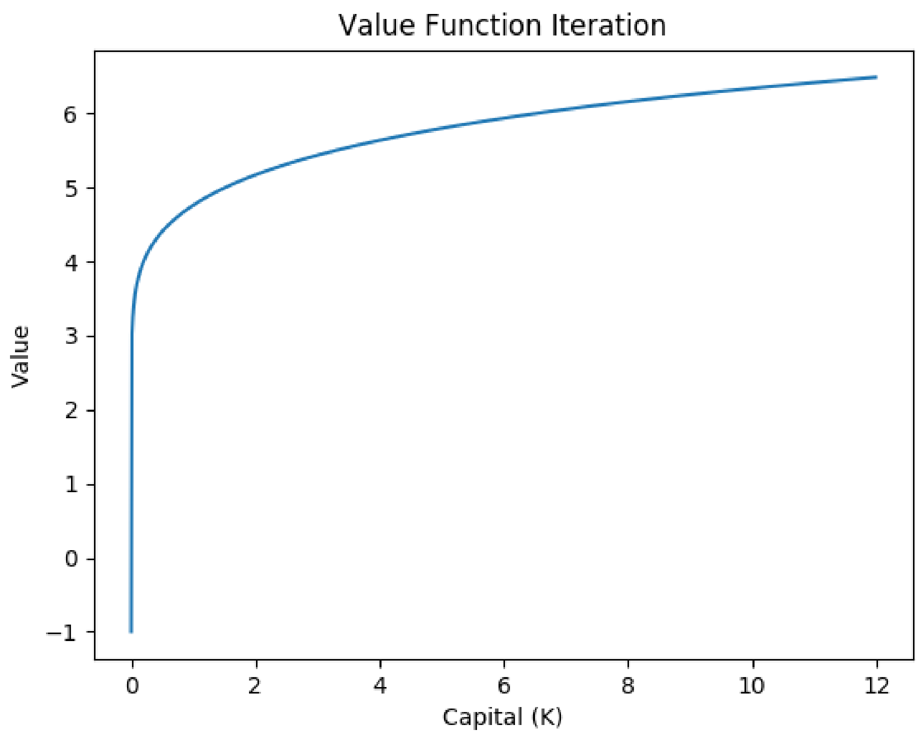

Figure 2: Value Function Iteration over Discretised Capital

1.2.2 Using Analytical Form

Value Function Iteration conducted over the discretised space for Capital assuming $\beta = 0.6,\ A = 1,\ \alpha = 0.3$, and $\delta = 1$ is superimposed with the Analytical Form for comparison of output,

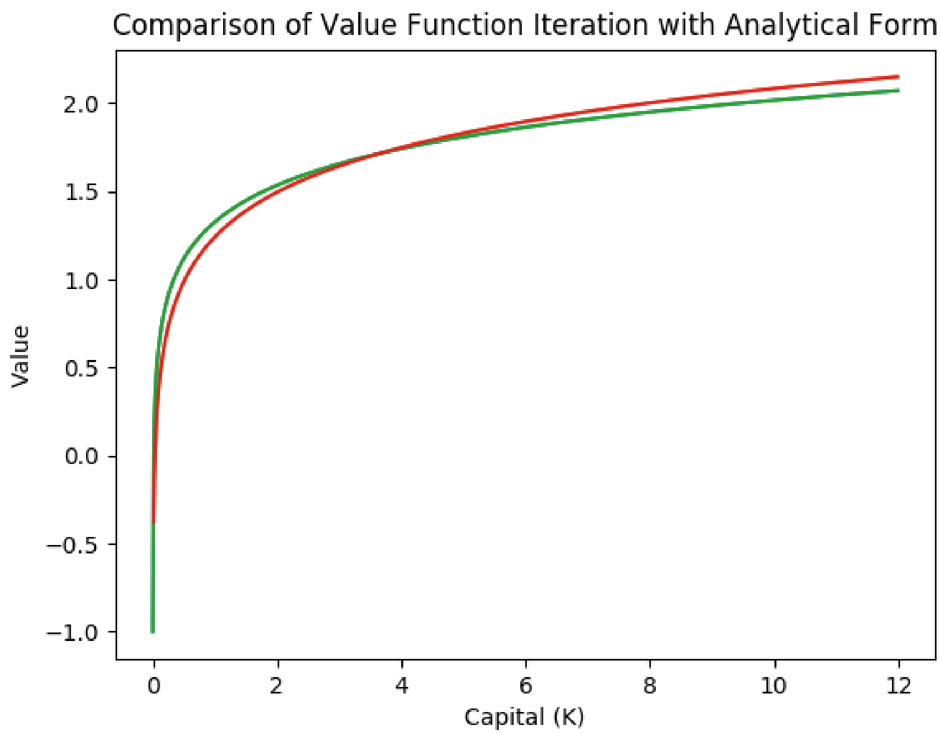

Figure 3: Comparison of Value Function Iteration with Analytical Form

With our initial guess of the functional form of the analytical form being,

$$V(K_t) = A + B \ln K_t \tag{12}$$

$$\implies A = \frac{\alpha\beta}{1-\alpha\beta} \ln \alpha\beta \tag{13}$$

$$\implies B = \frac{\alpha}{1-\alpha\beta} \tag{14}$$

2 Rust Model

```
Estimation of Parameters and respective Standard Errors:
Beta[0]:        [25.36425337]; (0.11561192456479381) 0.013366117101575576
Beta[1]:        [-1.48315299]; (3.004018606720816) 9.024127789524872
Beta[2]:        [0.18493143]; (0.04011846630681128) 0.001609491338810752
Alpha:          [-8.37344604]; (0.003586413934865264) 1.2862364912195745e-05
Sigma Alpha:    [1.075]; (1.265210529517815) 1.6007576840027502
```

3 Appendix

3.1 Source

3.1.1 Source: Question 1

```
#!/usr/bin/python3

# Import Python System Libraries
import sys
import time

# Import relevant Python Libraries
import matplotlib.pyplot as plt
import numpy as np

# Defining Helper Functions
def cout(text):
    t_cout = sys.stdout
    sys.stdout = open('pset1_output.txt', 'a')
    print(text)
    sys.stdout = t_cout
    print(text)
    return True

# Defining Estimation Loops
def RecursiveLoop(M_V0):
    # Constructing Utility Matrix
    tM_Vc1 = Production(M_Ki, v_parameter[0], v_parameter[1])
    tM_Vc2 = (1 - v_parameter[3]) * M_Ki
    tM_Vc3 = Utility( tM_Vc1 + tM_Vc2 - M_Kj )

    # Constructing Value Matrix
    tM_V = tM_Vc3 + v_parameter[2] * M_V0

    # Constructing Value Vectors (argmax)
    v_V0 = np.reshape( np.nanmax(M_V0, axis=1), (i_n, 1) )
    tv_V = np.reshape( np.nanmax(tM_V, axis=1), (i_n, 1) )

    M_Values = np.zeros( (1, i_n) )

    # Diagnose Value Matrix
    #cout( np.transpose(v_V0) )
    #cout( np.transpose(tv_V) )
    cout("")

    # Test for Convergence
    cout( "Convergence Test: " + str(np.linalg.norm(v_V0-tv_V)) + " < " + str(v_parameter[4

    if np.linalg.norm(v_V0-tv_V) > v_parameter[4]:
        # Enter Recursion
        cout("Convergence Failed ...")
```

```
        cout("Entering Recursion ...")
        M_Values = RecursiveLoop(tM_V)
    else:
        cout("Convergence Successful ...")

    cout("Collapsing Recursion ...")

    # Append Result to Matrix of Value Iterations
    return np.append(M_Values, np.transpose(tv_V), axis=0)

# Defining Utility Function
def Utility(x):
    tM_U = np.log(x)
    tM_U[x <= 0] = -1
    return tM_U

# Defining Production Function
def Production(x, coefficient, power):
    return coefficient * np.power(x, power * np.ones(x.shape))

# Discretisation
i_n = 1000
# Capital is discretised into finite bins

# Declaring Capital Vectors
v_Ki = np.arange(0, i_n, 1)
v_Ki = np.reshape(v_Ki, (i_n, 1))
v_K = v_Ki
v_Ki[0] = 12 / i_n
v_KP = np.arange(0, 12, 12 / i_n)
v_Kj = np.transpose(v_Ki)

# Capital Matrices
M_Ki = np.repeat( v_Ki, i_n, axis=1 )
M_Kj = np.repeat( v_Kj, i_n, axis=0 )
cout(M_Ki)
cout(M_Kj)

# Declaring Initial Guess for Values
M_InitialValues = np.zeros( (i_n, i_n) )

# Declaring Container for Iterated Values
M_IteratedValues = np.zeros( (1, i_n) )

cout("######")

# Set-up Output File Header [pset1_output.txt]
cout("[OUTPUT] Problem Set 1: Question 1 - S M Sajid Al Sanai")
t_time = time.asctime( time.localtime(time.time()) )
cout(t_time)
cout("")
```

```
# Question 1, Part A
cout("Question 1. (a)")
cout("i. A=20, alpha=0.3, beta=0.6, delta=0.5, epsilon=0.01")
cout("")

# Declaring Parameters
v_parameter = np.zeros( (5,) )
v_parameter[0] = 20     # A
v_parameter[1] = 0.3    # alpha
v_parameter[2] = 0.6    # beta
v_parameter[3] = 0.5    # delta
v_parameter[4] = 0.01   # epsilon

# Call Recursive Loop
M_IteratedValues = RecursiveLoop(M_InitialValues)
M_IteratedValues = np.delete(M_IteratedValues, 0, axis=0)
cout("")

# Display Matrix of Iterated Values
cout(M_IteratedValues)
cout("")

# Generate Plots
cout("Generating Plots ...")
cout("")
plt.plot(v_KP, M_IteratedValues[M_IteratedValues.shape[0]-1, :])
plt.title("Value Function Iteration")
plt.xlabel("Capital (K)")
plt.ylabel("Value")
plt.show()

# Question 1, Part B
cout("Question 1. (b)")
cout("i. Value Function Iteration")
cout("A=1, [alpha=0.3], [beta=0.6], delta=1, epsilon=0.01")
cout("")

# Declaring Parameters
v_parameter[0] = 1      # A
v_parameter[1] = 0.3    # alpha
v_parameter[2] = 0.6    # beta
v_parameter[3] = 1      # delta
v_parameter[4] = 0.01   # epsilon

# Call Recursive Loop
M_IteratedValues = RecursiveLoop(M_InitialValues)
M_IteratedValues = np.delete(M_IteratedValues, 0, axis=0)
cout("")

# Display Matrix of Iterated Values
cout(M_IteratedValues)
cout("")
```

```
# Generate Plots
cout("Generating Plots ...")
cout("")
plt.plot(v_KP, M_IteratedValues[M_IteratedValues.shape[0]-1, :])
plt.title("Value Function Iteration")
plt.xlabel("Capital (K)")
plt.ylabel("Value")
plt.show()

cout("ii. Analytical Form")
cout("A=1, [alpha=0.3], [beta=0.6], delta=1, epsilon=0.01")
cout("")

f_ab = v_parameter[1] * v_parameter[2]
f_B  = v_parameter[1] / (1 - f_ab)
f_A  = v_parameter[2] * f_B * np.log(f_ab)
v_VA = f_A * np.ones( (i_n, 1) ) + f_B * np.log(v_Ki)

#cout(v_VA)
#cout("")

# Generate Plots
cout("Generating Plots ...")
cout("")
plt.plot(v_KP, M_IteratedValues[M_IteratedValues.shape[0]-1, :])
plt.plot(v_KP, v_VA)
plt.plot(v_KP, M_IteratedValues[M_IteratedValues.shape[0]-1, :], v_KP, v_VA)
plt.title("Comparison of Value Function Iteration with Analytical Form")
plt.xlabel("Capital (K)")
plt.ylabel("Value")
plt.show()
```

3.1.2 Source: Question 2

```
#!/usr/bin/python3

# Import Python System Libraries
import sys
import time

# Import relevant Python Libraries
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
from scipy.linalg import lu, lu_factor, lu_solve
from scipy import optimize

# Defining Helper Functions
def cout(text):
    t_cout = sys.stdout
    sys.stdout = open('pset1_output.txt', 'a')
    print(text)
    sys.stdout = t_cout
    print(text)
    return True

# Defining Estimation Loops
def RecursiveLoop_ChoiceSpecific(M_V0, parameter):
    # Constructing Utility Vector
    tM_V = np.append( Utility(0, v_x, parameter), Utility(1, v_x, parameter), axis=1 )

    # Constructing Temporary Value Vector
    tc1 = v_parameter[6] * np.ones( (i_n, i_d) )
    tc2 = np.log(np.exp(M_V0[:, 0]) + np.exp(M_V0[:, 1]))
    tc2 = np.repeat(np.reshape(tc2, (i_n, 1)), 2, axis=1)
    tc2[:, 0] = np.matmul( M_TransitionProbability0, tc2[:, 0])
    tc2[:, 1] = np.matmul( M_TransitionProbability1, tc2[:, 1])
    tc2 *= parameter[1]
    tM_V += tc1 + tc2

    # Constructing Iteration Value Vector
    M_Values = tM_V
    cout("")

    # Test for Convergence
    cout( "Convergence Test: " + str(np.linalg.norm(M_V0.flatten()-tM_V.flatten())) + " < "

    if np.linalg.norm(M_V0.flatten()-tM_V.flatten()) > parameter[2]:
        # Enter Recursion
        cout("Convergence Failed ...")
        cout("Entering Recursion ...")
        M_Values = RecursiveLoop_ChoiceSpecific(tM_V, parameter)
    else:
        cout("Convergence Successful ...")
```

```
    cout("Collapsing Recursion ...")
    return M_Values

def RecursiveLoop_Integrated(v_PR0, v_V0, parameter):
    # Declare Probability of Replacement across Realisable State Variable
    tv_ProbabilityReplacement = v_PR0

    # Define Errors by Replacement Decision
    tM_Errors = np.zeros( (i_n, i_d) )
    tM_Errors[:, 0] = np.reshape(parameter[6] * np.ones( (i_n, 1) ) - np.log(np.ones( (i_n,
    tM_Errors[:, 1] = np.reshape(parameter[6] * np.ones( (i_n, 1) ) - np.log(tv_ProbabilityR

    # Constructing Temporary Value Vector
    tuc0 = Utility(0, v_x, parameter) + tM_Errors[:, 0]
    tuc1 = Utility(1, v_x, parameter) + tM_Errors[:, 1]
    tv_R = np.multiply(np.ones( (i_n, 1) ) - tv_ProbabilityReplacement, tuc0) + np.multiply
    tv_R = np.reshape(np.diag(tv_R), (i_n, 1))
    tM_G = np.multiply(tv_ProbabilityReplacement, M_TransitionProbability1) + np.multiply((

    # Calculate Iterated Values
    # (Do not use Inverse or you will not graduate)
    #tM_V = np.linalg.inv(np.identity(i_n) - parameter[1] * tM_G)
    #tM_V = np.matmul(tM_V, tv_R)
    tM_V = lu_solve(lu_factor(np.identity(i_n) - parameter[1] * tM_G), tv_R)

    # Update Probabilities and Errors
    tv_ProbabilityReplacement = ConditionalChoice(Utility(0, v_x, parameter) + parameter[1]

    # Constructing Iteration Value Vector
    M_Values = tM_V
    cout("")

    cout( "Convergence Test: " + str(np.linalg.norm(v_V0.flatten()-tM_V.flatten())) + " < "

    if np.linalg.norm(v_V0.flatten()-tM_V.flatten()) > parameter[2]:
        # Enter Recursion
        cout("Convergence Failed ...")
        cout("Entering Recursion ...")
        M_Values, tv_ProbabilityReplacement = RecursiveLoop_Integrated(tv_ProbabilityReplace
    else:
        cout("Convergence Successful ...")

    cout("Collapsing Recursion ...")
    return M_Values, tv_ProbabilityReplacement

def ForwardSimulation_ChoiceSpecific(M_PR0, M_Errors, binomial_seed):
    np.random.seed(binomial_seed)
    tv_PathDecision = np.zeros( (i_t, 1) )
    tv_PathPolicy   = np.zeros( (i_t, 1) )
    for t in range(i_t):
        tx = tv_PathPolicy[t].astype(int)
```

```
        t_EstimatedDifference = np.log(M_PR0[tx, 1]) - np.log(M_PR0[tx, 0])
        if t_EstimatedDifference > M_Errors[t, 1] - M_Errors[t, 0]:
            tv_PathDecision[t] = 1
            if t < i_t - 1:
                tv_PathPolicy[t+1] = 0
        else:
            tv_PathDecision[t] = 0
            if t < i_t - 1:
                tv_PathPolicy[t+1] = min(tv_PathPolicy[t] + np.random.binomial(1, v_paramet
    return tv_PathDecision, tv_PathPolicy

# Defining Conditional Choice Probability Function
def ConditionalChoice(Value0, Value1, i):
    t_denominator = np.exp(Value0) + np.exp(Value1)
    t_numerator   = (1 - i) * np.exp(Value0) + i * np.exp(Value1)
    return t_numerator / t_denominator

# Defining Utility Function
def Utility(i, x, parameter):
    uc1 = -Cost(x, parameter)
    uc2 = -parameter[5] * np.ones( (x.shape[0], 1) )
    uc1 = np.reshape( uc1, (x.shape[0], 1) )
    uc2 = np.reshape( uc2, (x.shape[0], 1) )
    return (1 - i) * uc1 + i * uc2

# Defining Cost Function
def Cost(x, parameter):
    return parameter[3] * x + parameter[4] * np.power(x, 2)

# Defining Transition Probability Matrix Generation Function
def TransitionProbability(parameter, replacement):
    p = np.array( (1 - v_parameter[0], v_parameter[0], 0, 0, 0, 0, 0, 0, 0, 0, 0) )
    P = np.reshape(np.tile( p, (1, i_n) ), (i_n, i_n))
    if replacement != 1:
        for i in range(i_n):
            P[i, :] = np.roll(P[i, :], i)
        P[i_n -1, i_n -1] = 1
        P[i_n -1, 0] = 0
    return P

# Discretisation
i_n = 11
i_d = 2
i_t = 5000

# Declaring State Vector
v_x = np.reshape(np.arange(0, i_n, 1), (i_n, 1))

# Declaring Time Vector
v_t = np.reshape(np.arange(0, i_t, 1), (i_t, 1))

'''
```

```
'''

cout("######")

# Set-up Output File Header [pset1_output.txt]
cout("[OUTPUT] Problem Set 1: Question 2 - S M Sajid Al Sanai")
t_time = time.asctime( time.localtime(time.time()) )
cout(t_time)
cout("")

# Question 2, Part B
cout("Question 2. (b) i. Choice Specific Value Function")
cout("")

# Declaring Parameters
v_parameter = np.zeros( (7,) )
v_parameter[0] = 0.8     # lambda
v_parameter[1] = 0.95    # beta
v_parameter[2] = 0.001   # epsilon
v_parameter[3] = 0.3     # theta1
v_parameter[4] = 0.0     # theta2
v_parameter[5] = 4.0     # theta3 R replacement cost
v_parameter[6] = 0.5772  # euler constant

# Declaring Initial Guess for Values
M_InitialValues = np.zeros( (i_n, i_d) )

# Declaring Container for Iterated Values
M_IteratedValues = np.zeros( (i_n, i_d) )

# Declaring Transition Probabilities
M_TransitionProbability0 = TransitionProbability(v_parameter, 0)
M_TransitionProbability1 = TransitionProbability(v_parameter, 1)

# Declaring Conditional Choice Probabilities
M_CCProbability = np.zeros( (i_n, i_d) )

# Call Recursive Loop
M_IteratedValues = RecursiveLoop_ChoiceSpecific(M_InitialValues, v_parameter)
cout("")

# Display Vector of Iterated Values
cout(M_IteratedValues)
cout("")

# Generate Plots
cout("Generating Plots ...")
cout("")
plt.plot(v_x, M_IteratedValues[:, 0], v_x, M_IteratedValues[:, 1])
plt.title("Choice Specific Value Function Iteration")
plt.xlabel("Mileage Realisations")
plt.ylabel("Value")
```

```
plt.show()

# Generate Conditional Choice Probabilities
cout("Conditional Choice Probabilities:")
M_CCProbability[:, 0] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1], 0
M_CCProbability[:, 1] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1], 1
cout("[[ x; Pr(i=0|x,theta); Pr(i=1|x,theta); ]]")
cout(np.append(v_x, M_CCProbability, axis=1))
cout("")

'''
'''

cout("Question 2. (b) ii. Integrated Value Function")
cout("")

# Declaring Initial Guess for Probabilities
v_InitialProbabilities = 0.1 * np.ones( (i_n, 1) )

# Declaring Initial Guess for Values
M_InitialValues = np.zeros( (i_n, 1) )

# Call Recursive Loop
M_IteratedValues, M_CCProbability = RecursiveLoop_Integrated(v_InitialProbabilities, M_Init
cout("")

# Display Vector of Iterated Values
cout(M_IteratedValues)
cout("")

# Generate Plots
cout("Generating Plots ...")
cout("")
plt.plot(v_x, M_IteratedValues, v_x, Utility(1, v_x, v_parameter) + v_parameter[1] * np.mat
plt.title("Integrated Value Function Iteration")
plt.xlabel("Mileage Realisations")
plt.ylabel("Value")
plt.show()

# Generate Conditional Choice Probabilities
cout("Conditional Choice Probabilities:")
M_CCProbability = np.append(np.ones( (i_n, 1) ) - M_CCProbability, M_CCProbability, axis=1)
cout("[[ x; Pr(i=0|x,theta); Pr(i=1|x,theta); ]]")
cout(np.append(v_x, M_CCProbability, axis=1))
cout("")

'''
'''

# Question 2, Part C
cout("Question 2. (c) Forward Simulation")
cout("")
```

```
# Load Uniformly Distributed Errors as Type I Extreme Value
M_TypeIErrors = np.loadtxt('./draw.out')
M_TypeIErrors = np.log( -np.log( M_TypeIErrors ) )

# Declaring Initial Guess for Values
M_InitialValues = np.zeros( (i_n , i_d) )

# Declaring Container for Iterated Values
M_IteratedValues = np.zeros( (i_n , i_d) )

# Declaring Transition Probabilities
M_TransitionProbability0 = TransitionProbability(v_parameter , 0)
M_TransitionProbability1 = TransitionProbability(v_parameter , 1)

# Declaring Conditional Choice Probabilities
M_CCProbability = np.zeros( (i_n , i_d) )

# Call Recursive Loop
M_IteratedValues = RecursiveLoop_ChoiceSpecific(M_InitialValues , v_parameter)
cout("")

# Generate Conditional Choice Probabilities
M_CCProbability[:, 0] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1], 0
M_CCProbability[:, 1] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1], 1

# Generate Policy and Decision Paths
v_PathDecision , v_PathPolicy = ForwardSimulation_ChoiceSpecific(M_CCProbability , M_TypeIErr

v_Decision = np.zeros( (i_n , 1) )
for t in range(i_t):
    if v_PathDecision[t] == 1:
        v_Decision[int(v_PathPolicy[t])] += 1

# Generate Plots
cout("Generating Plots ...")
cout("")

plt.bar(["Do not replace","Replace"], [i_t - np.sum(v_PathDecision), np.sum(v_PathDecision)
plt.title("Simulated Frequencies of Replacement Decisions")
plt.xlabel("Replacement Decision")
plt.ylabel("Frequency")
plt.show()

plt.plot(v_Decision)
plt.title("Simulated Mileage Before Replacement")
plt.xlabel("Mileage")
plt.ylabel("Frequency of Replacements")
plt.show()

# Diagnose Simulated Path
#cout(np.append(v_PathPolicy , v_PathDecision , axis=1))
```

```
#cout("")

'''
'''

# Question 2, Part D
cout("Question 2. (d) Maximum Likelihood Estimation")
cout("")

def RecursiveLoop_Inner(M_V0, parameter):
        # Constructing Utility Vector
    tM_V = np.append( Utility(0, v_x, parameter), Utility(1, v_x, parameter), axis=1 )

    # Constructing Temporary Value Vector
    tc1 = v_parameter[6] * np.ones( (i_n, i_d) )
    tc2 = np.log(np.exp(M_V0[:, 0]) + np.exp(M_V0[:, 1]))
    tc2 = np.repeat(np.reshape(tc2, (i_n, 1)), 2, axis=1)
    tc2[:, 0] = np.matmul(M_TransitionProbability0, tc2[:, 0])
    tc2[:, 1] = np.matmul(M_TransitionProbability1, tc2[:, 1])
    tc2 *= parameter[1]
    tM_V += tc1 + tc2

    # Constructing Iteration Value Vector
    M_Values = tM_V
    #cout("")

    # Test for Convergence
    #cout( "Convergence Test: " + str(np.linalg.norm(M_V0.flatten()-tM_V.flatten())) + " < 

    if np.linalg.norm(M_V0.flatten()-tM_V.flatten()) > parameter[2]:
        # Enter Recursion
        #cout("Convergence Failed ...")
        #cout("Entering Recursion ...")
        M_Values = RecursiveLoop_Inner(tM_V, parameter)
    else:
        cout("Convergence Successful ...")

    #cout("Collapsing Recursion ...")
    return M_Values

def RecursiveLoop_OuterNFPA(theta, p_lambda, p_beta, p_epsilon, p_euler_constant, decision,
    # Obtain Expected Values
    parameter = np.zeros((7, 1))
    parameter[0] = p_lambda
    parameter[1] = p_beta
    parameter[2] = p_epsilon
    parameter[3] = theta[0]
    parameter[4] = theta[1]
    parameter[5] = theta[2]
    parameter[6] = p_euler_constant
    M_EV = RecursiveLoop_Inner(np.zeros((i_n, i_d)), parameter)
```

```
    # Declare Log Likelihood Loop
    f_LogLikelihood = 0

    # Call Log Likelihood Loop
    for t in range(1, i_t):
        # Determine Mileage Transition Probability
        i_delta_mileage = policy[t] - policy[t-1]
        if i_delta_mileage == 1:
            f_probability_mileage = p_lambda
        elif i_delta_mileage == 0 and decision[t-1] == 0:
            f_probability_mileage = 1 - p_lambda
        elif policy[t] == 0:
            f_probability_mileage = 1 - p_lambda
        elif policy[t] == 1 and decision[t-1] == 1:
            f_probability_mileage = p_lambda

        # Determine Replacement Probability
        f_probability_replacement = ProbabilityReplacement(M_EV, parameter)
        t_probability_replacement =f_probability_replacement[0, int(policy[t])]
        if decision[t] == 1:
            f_probability_replacement = t_probability_replacement
        else:
            f_probability_replacement = 1 - t_probability_replacement

        if f_probability_replacement <= 0:
            f_probability_mileage = 0.0001

        # Sum over Log Likelihoods
        f_LogLikelihood += np.log(f_probability_replacement) + np.log(f_probability_mileage

    # Return Sum of Log Likelihood
    cout("Generated Log Likelihood ... " + str(-f_LogLikelihood))
    return -f_LogLikelihood

def ProbabilityReplacement(M_EV, parameter):
    V0 = np.exp(Utility(0, v_x, parameter) + parameter[1] * M_EV[:, 0])
    V1 = np.exp(Utility(1, v_x, parameter) + parameter[1] * M_EV[:, 1])
    return V1 / (V0 + V1)

# Declaring Initial Guess for Parameters before MLE
v_InitialTheta = np.zeros( (3,) )
v_InitialTheta[0] = 0.3
v_InitialTheta[1] = 0.0
v_InitialTheta[2] = 4.0

cout("Initial Guess for Theta: " + str(v_InitialTheta))
cout("")

cout("Running Minimisation Routine:")
v_OutputTheta = sp.optimize.minimize(RecursiveLoop_OuterNFPA, x0=v_InitialTheta, args=(v_pa
#v_OutputTheta = np.append(0.3, np.append(0, 4.0))
cout("")
```

```
cout("Minimised Theta: " + str(v_OutputTheta))
cout("")

# Simplex Nealder-Mead is preferable to BFGS which goes to negative values

'''
'''

# Question 2, Part E
cout("Question 2. (e) i. Forward Simulation with Minimised Parameters")
cout("")

# Declaring Container for Iterated Values
M_IteratedValues = np.zeros( (i_n, i_d) )

# Declaring Conditional Choice Probabilities
M_CCProbability0 = np.zeros( (i_n, i_d) )

# Call Recursive Loop
M_IteratedValues = RecursiveLoop_ChoiceSpecific(M_InitialValues, np.append(v_parameter[0:2]
cout("")

# Generate Conditional Choice Probabilities
cout("Conditional Choice Probabilities:")
M_CCProbability0[:, 0] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1],
M_CCProbability0[:, 1] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1],
cout("[[ x; Pr(i=0|x,theta); Pr(i=1|x,theta); ]]")
cout(np.append(v_x, M_CCProbability0, axis=1))
cout("")

# Generate Policy and Decision Paths
v_PathDecision, v_PathPolicy = ForwardSimulation_ChoiceSpecific(M_CCProbability0, M_TypeIEr

v_Decision = np.zeros( (i_n, 1) )
v_States   = np.zeros( (i_n, 1) )
for t in range(i_t):
    if v_PathDecision[t] == 1:
        v_Decision[int(v_PathPolicy[t])] += 1
    v_States[int(v_PathPolicy[t])] += 1

# Long Run Replacement Probabilities
cout("Long Run Replacement Probabilities:")
cout(v_Decision)
cout(v_States)
v_LRReplacementProbability0 = np.divide(v_Decision, v_States)
cout(v_LRReplacementProbability0)
cout("[[ x; Pr(i=0|x,theta); Pr(i=1|x,theta); ]]")
cout(np.append(v_x, np.append(np.ones((i_n, 1)) - v_LRReplacementProbability0, v_LRReplacem
cout("")

# Generate Plots
```

```
cout("Generating Plots ...")
cout("")

plt.bar(["Do not replace","Replace"], [i_t - np.sum(v_PathDecision), np.sum(v_PathDecision)
plt.title("Simulated Frequencies of Replacement Decisions")
plt.xlabel("Replacement Decision")
plt.ylabel("Frequency")
plt.show()

plt.plot(v_Decision)
plt.title("Simulated Mileage Before Replacement")
plt.xlabel("Mileage")
plt.ylabel("Frequency of Replacements")
plt.show()

cout("Question 2. (e) ii. Long Run Replacement Probabilities (SS)")

cout("Question 2. (e) iii. Counterfactual with Subsidy on Minimised Parameters")

# Declaring Container for Iterated Values
M_IteratedValues = np.zeros( (i_n, i_d) )

# Declaring Conditional Choice Probabilities
M_CCProbability1 = np.zeros( (i_n, i_d) )

# Call Recursive Loop
M_IteratedValues = RecursiveLoop_ChoiceSpecific(M_InitialValues, np.append(v_parameter[0:2]
cout("")

# Generate Conditional Choice Probabilities
M_CCProbability1[:, 0] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1],
M_CCProbability1[:, 1] = ConditionalChoice(M_IteratedValues[:, 0], M_IteratedValues[:, 1],

# Generate Policy and Decision Paths
v_PathDecision, v_PathPolicy = ForwardSimulation_ChoiceSpecific(M_CCProbability1, M_TypeIEr

v_Decision = np.zeros( (i_n, 1) )
v_States   = np.zeros( (i_n, 1) )
for t in range(i_t):
    if v_PathDecision[t] == 1:
        v_Decision[int(v_PathPolicy[t])] += 1
    v_States[int(v_PathPolicy[t])] += 1

# Long Run Replacement Probabilities
cout("(10% Subsidy) Long Run Replacement Probabilities:")
v_LRReplacementProbability1 = np.divide(v_Decision, v_States)
cout("[[ x; Pr(i=0|x,theta); Pr(i=1|x,theta); ]]")
cout(np.append(v_x, np.append(np.ones((i_n, 1)) - v_LRReplacementProbability1, v_LRReplacem
cout("")

cout("(Differential) Long Run Replacement Probabilities:")
cout("[[ x; Pr(i=0|x,theta); Pr(i=1|x,theta); ]]")
```

```
cout(np.append(v_x, v_LRReplacementProbability1 - v_LRReplacementProbability0, axis=1))
cout("")
```

3.2 Output

```
Minimised Theta:        fun: 5319.407700566876
 hess_inv: array([[ 4.16487948e-08, -9.90443077e-10, -1.04954367e-07],
       [-9.90443077e-10,  2.32738993e-10,  3.50993346e-09],
       [-1.04954367e-07,  3.50993346e-09,  2.71644050e-07]])
      jac: array([ 0.00787354, -0.00042725,  0.00372314])
  message: 'Desired error not necessarily achieved due to precision loss.'
     nfev: 595
      nit: 19
     njev: 115
   status: 2
  success: False
        x: array([ 0.82498424, -0.03871543,  2.14309007])

Question 2. (e) i. Forward Simulation with Minimised Parameters
[[   0    0    0 ...    0    0    0]
 [   1    1    1 ...    1    1    1]
 [   2    2    2 ...    2    2    2]
 ...
 [997 997 997 ... 997 997 997]
 [998 998 998 ... 998 998 998]
 [999 999 999 ... 999 999 999]]
[[   0    1    2 ... 997 998 999]
 [   0    1    2 ... 997 998 999]
 [   0    1    2 ... 997 998 999]
 ...
 [   0    1    2 ... 997 998 999]
 [   0    1    2 ... 997 998 999]
 [   0    1    2 ... 997 998 999]]
######
[OUTPUT] Problem Set 1: Question 1 - S M Sajid Al Sanai
Fri Jun  7 14:38:41 2019

Question 1. (a)
i. A=20, alpha=0.3, beta=0.6, delta=0.5, epsilon=0.01


Convergence Test: 183.32179637054315 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 109.9930778223259 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 65.99584669339553 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 39.59750801603732 < 0.01
Convergence Failed ...
```

```
Entering Recursion ...

Convergence Test: 23.75850480962239 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 14.255102885773432 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 8.553061731464059 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 5.131837038878435 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 3.079102223327062 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 1.8474613339962376 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 1.1084768003977425 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.6650860802386442 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.3990516481431881 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.23943098888591174 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.14365859333154649 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.08619515599892967 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.051717093599356105 < 0.01
Convergence Failed ...
```

```
Entering Recursion ...

Convergence Test: 0.031030256159614477 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.01861815369576841 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.01117089221746202 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.00670253533047651 < 0.01
Convergence Successful ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...

[[-2.49994516  7.55089657  8.10853718 ... 16.2197008  16.22178344
  16.22386422]
 [-2.4999086   7.55078614  8.10841859 ... 16.21946359 16.2215462
  16.22362695]
 [-2.49984766  7.55060208  8.10822095 ... 16.21906824 16.22115079
  16.22323149]
 ...
 [-1.96        5.92003278  6.35723261 ... 12.71652439 12.71815722
  12.71978858]
 [-1.6         4.83267982  5.18957764 ... 10.38083624 10.38216916
  10.38350088]
 [-1.          3.02042489  3.24348602 ...  6.48802265  6.48885572
   6.48968805]]
```

```
Generating Plots ...

Question 1. (b)
i. Value Function Iteration
A=1, [alpha=0.3], [beta=0.6], delta=1, epsilon=0.01


Convergence Test: 56.831450641975245 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 34.098870385185144 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 20.459322231111088 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 12.275593338666653 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 7.365356003199991 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 4.419213601919994 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 2.651528161151997 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 1.5909168966911977 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.9545501380147192 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.5727300828088309 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.34363804968529915 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.206182829811179 < 0.01
```

```
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.1237096978867079 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.07422581873202484 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.044535491239214665 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.026721294743528783 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.016032776846116908 < 0.01
Convergence Failed ...
Entering Recursion ...

Convergence Test: 0.009619666107670408 < 0.01
Convergence Successful ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...
Collapsing Recursion ...

[[-2.4997461   0.          0.51980759 ...  5.17803714  5.17878895
   5.17954   ]
 [-2.49957683  0.          0.51977239 ...  5.17768652  5.17843827
   5.17918927]
 [-2.49929472  0.          0.51971373 ...  5.17710215  5.17785382
   5.17860473]
 ...
 [-1.96         0.          0.40757054 ...  4.05999345  4.06058293
```

```
   4.06117181]
 [-1.6          0.           0.33271065 ...  3.31428037  3.31476157
   3.31524229]
 [-1.           0.           0.20794415 ...  2.07142523  2.07172598
   2.07202643]]

Generating Plots ...

ii. Analytical Form
A=1, [alpha=0.3], [beta=0.6], delta=1, epsilon=0.01

Generating Plots ...
```