EEE I - Problem Set 2

Ozzy Houck

2024-11-08

Worked with Keisuke Ito and had chatgpt help revise code and improve style

Set up

```
import numpy as np
from scipy.sparse import csr_matrix
from scipy.stats import norm
import scipy.integrate as integrate
import altair as alt
import pandas as pd
from math import exp

import warnings
warnings.filterwarnings('ignore')
alt.renderers.enable("png")

# part (c)
def utility1(y: int) -> float:
    return 2 * y ** (1/2)
def marginal_utility1(y):
    return np.where(y > 0, y ** (-0.5), np.inf)

def utility2(y: int) -> float:
    return 5 * y - 0.05 * y ** 2
def marginal_utility2(y: int) -> float:
    return 5 - 0.1 * y

# Set up starting values
```

```
stock = 1000

discount_rate = 0.05
delta = 1 / (1 + discount_rate)

N = 501 # number of states
nA = 501 # number of actions
step_size = stock / (N - 1)
```

Question 1

```
# part a) and b)
# state space and action space are 0 to 1000 by 2
state_space = np.linspace(0, stock, N)
action_space = np.linspace(0, stock, nA)

state_space_matrix = np.tile(state_space.reshape(N, 1), (1, nA)) # N x nA
action_space_matrix = np.tile(action_space.reshape(1, nA), (N, 1)) # N x nA
feasible_actions = action_space_matrix <= state_space_matrix

# part (d)
utility_matrix1 = np.where(feasible_actions,
                            utility1(action_space_matrix), -np.inf)
utility_matrix2 = np.where(feasible_actions,
                                utility2(action_space_matrix), -np.inf)

utility_matrix1_flat = utility_matrix1.flatten() # N * nA 1D array to use in
↪ Bellman with trnasition matrix
utility_matrix2_flat = utility_matrix2.flatten()

# get next state_indices
update_state_matrix = state_space_matrix - action_space_matrix
next_state_indices = np.zeros((N, nA), dtype=int)

# part (e)
next_state_indices[feasible_actions] = (update_state_matrix[feasible_actions]
↪ / step_size).astype(int)

# part (f)
```

```
# create  (N * nA, N) matrix transition matrix
row_indices = []
col_indices = []
data = []

for i in range(N):  # state index
    for j in range(nA):  # action index
        if feasible_actions[i, j]:
            next_state_index = next_state_indices[i, j]  # state after action
            row_index = i * nA + j  # Correct row index for (state, action)
            col_index = next_state_index  # Next state index
            row_indices.append(row_index)
            col_indices.append(col_index)
            data.append(1)  # deterministic transition

transition_matrix = csr_matrix((data, (row_indices, col_indices)), shape=(N *
 ↪  nA, N))

# part (g)
def bellman(v, U_flat, transition_matrix, delta, N, nA):
    '''Evaluated RHS of bellman before max
    - v: current value function (N, )
    - U_flat: flattened utility matrix (N * nA, )
    - transition_matrix: (N, N * nA) sparse matrix
    - delta: dicsount factor
    - N: number of states
    - nA: number of actions
    '''
    v_next = transition_matrix.dot(v)  # N * nA, 1
    B_flat = U_flat + delta * v_next # (N * nA, 1)
    B_sa = B_flat.reshape(N, nA)  # (N, nA) sa for state action
    return B_sa

# Value function iteration function
max_iterations = 1000
np.random.seed(454)

# start with value function  of zeros
value_function = np.zeros(N)
utility_matrix = utility_matrix1
iteration = 0
tolerance = 1e-8
```

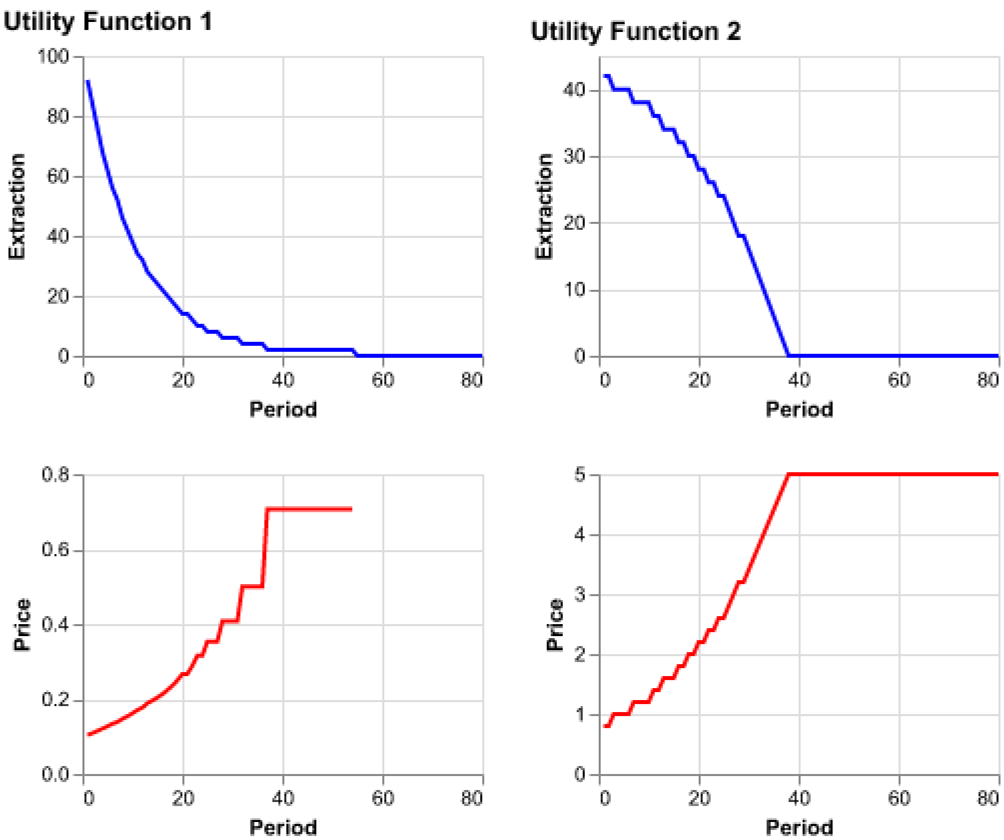
```python
def value_function_iteration(U_flat, transition_matrix, delta, N, nA, tolerance, max_iterations):
    """
    Performs value function iteration using the Bellman function.

    Parameters:
    - U: Utility matrix (N x nA array)
    - next_state_indices: Next state indices matrix (N x nA array)
    - delta: Discount factor
    - N: Number of states
    - nA: Number of actions
    - tolerance: Convergence tolerance
    - max_iterations: Maximum number of iterations

    Returns:
    - v: Value function (N x nA array)
    - policy: Optimal action indices for each state (N x 1 array)
    """
    v = value_function
    for iteration in range(max_iterations):
        B_sa = bellman(v, U_flat, transition_matrix, delta, N, nA)
        v_new = np.max(B_sa, axis=1) # find value of best action for each state
        policy = np.argmax(B_sa, axis=1) # find index of best action for each state
        diff = np.max(np.abs(v_new - v))
        v = v_new
        if diff < tolerance:
            print(f'Converged in {iteration + 1} iterations')
            break
    else:
        print('Did not converge within max iterations')
    return v, policy

def get_optimal_transition_matrix(N, C, next_state_indices):
    """
    Constructs the optimal transition matrix Topt.

    Parameters:
    - N: Number of states
    - C: Optimal action indices for each state (array of size N)
    - next_state_indices: Next state indices matrix (N x nA array)
```

```
    Returns:
    - Topt: Optimal transition matrix (N x N sparse matrix)
    """
    row_indices = np.arange(N)
    col_indices = next_state_indices[np.arange(N), C]
    data = np.ones(N)
    Topt = csr_matrix((data, (row_indices, col_indices)), shape=(N, N))
    return Topt

# Solve for utility function 1
print("Solving for utility function 1:")
v_u1, C_u1 = value_function_iteration(utility_matrix1_flat,
 ↪  transition_matrix, delta, N, nA, tolerance, max_iterations)

# Part (h) Find optimal transition matrix
Topt_u1 = get_optimal_transition_matrix(N, C_u1, next_state_indices)

# Solve for utility function 2
print("Solving for utility function 2:")
v_u2, C_u2 = value_function_iteration(utility_matrix2_flat,
 ↪  transition_matrix, delta, N, nA, tolerance, max_iterations)

# Part (h) Find optimal transition matrix
Topt_u2 = get_optimal_transition_matrix(N, C_u2, next_state_indices)

# Part (i) Simulate the model for t= 80 periods
T = 80
remaining_stock = stock
extraction_path = []
stock_path = []
price_path = []

def simulation(T, starting_stock, price_function, C, action_space,
 ↪  step_size):
    remaining_stock = starting_stock
    extraction_path = []
    stock_path = []
    price_path = []
    for t in range(T):
        stock_path.append(remaining_stock)
        current_state = int(remaining_stock / step_size)
```

```python
        if current_state >= N:
            print("Error: State index out of bounds")
            break

        action_index = C[current_state]
        action = action_space[action_index]
        remaining_stock -= action
        if remaining_stock < 0:
            print("Error: Negative stock")
            break

        extraction_path.append(action)
        price = price_function(action)
        price_path.append(price)
    return extraction_path, stock_path, price_path


def make_extraction_plot(extraction_path, stock_path, price_path, title):
    periods = np.arange(1, len(extraction_path) + 1)
    df = pd.DataFrame({
        'Period': periods,
        'Extraction': extraction_path,
        'Price': price_path,
        'Stock': stock_path
    })

    extraction_chart = alt.Chart(df).mark_line(color='blue').encode(
        x=alt.X('Period', axis=alt.Axis(title='Period')),
        y=alt.Y('Extraction', axis=alt.Axis(title='Extraction'))
    ).properties(
        width=200,
        height=150
    )

    price_chart = alt.Chart(df).mark_line(color='red').encode(
        x=alt.X('Period', axis=alt.Axis(title='Period')),
        y=alt.Y('Price', axis=alt.Axis(title='Price')),
    ).properties(
        width=200,
        height=150
    )
```

```python
    combined_chart = alt.vconcat(extraction_chart, price_chart).properties(
        title=title
    )
    return combined_chart


extraction_path, stock_path, price_path = simulation(T, stock,
 ↪  marginal_utility1, C_u1, action_space, step_size)

# this price path is ugly looking so round to 3 decimal places
price_path = np.round(price_path, 3)

chart_1 = make_extraction_plot(extraction_path, stock_path, price_path,
 ↪  "Utility Function 1")

extraction_path, stock_path, price_path = simulation(T, stock,
 ↪  marginal_utility2, C_u2, action_space, step_size)
chart_2 = make_extraction_plot(extraction_path, stock_path, price_path,
 ↪  "Utility Function 2")


chart_1 | chart_2
```

```
Solving for utility function 1:
Converged in 55 iterations
Solving for utility function 2:
Converged in 38 iterations
```



Question 2

```
# action space is the same as before
action_space = np.linspace(0, stock, nA)

# create statespace even over the squre root of the action space
state_space = np.linspace(0, stock ** 0.5, N) ** 2

# constant on the columns, state space on the rows
state_space_matrix = np.tile(state_space.reshape(N, 1), (1, nA)) # N x nA

# constant on the rows, action space on the columns
action_space_matrix = np.tile(action_space.reshape(1, nA), (N, 1)) # N x nA
feasible_actions = action_space_matrix <= state_space_matrix
```

```
utility_matrix1 = np.where(feasible_actions,
                           utility1(action_space_matrix), -1e10)
utility_matrix2 = np.where(feasible_actions,
                               utility2(action_space_matrix), -1e10)

utility_matrix1_flat = utility_matrix1.flatten() # N * nA 1D array to use in
↪ Bellman with trnasition matrix
utility_matrix2_flat = utility_matrix2.flatten()


update_state_matrix = state_space_matrix - action_space_matrix

# make sparse transition matrix
row_indices = []
col_indices = []
data = []

for i in range(N):  # state index
    for j in range(nA):  # action index
        if feasible_actions[i, j]:
            current_state = state_space[i]
            action = action_space[j]
            next_state = current_state - action

            row_index = i * nA + j  # row index for (state, action)

            if next_state in state_space:
                next_state_index = np.where(state_space == next_state)[0][0]
                col_indices.append(next_state_index)
                row_indices.append(row_index)
                data.append(1.0)
            else:
                if next_state <= state_space[0]:
                    next_state_index_low = 0
                    next_state_index_high = 0
                    weight_low = 1.0
                    weight_high = 0.0
                elif next_state >= state_space[-1]:
                    next_state_index_low = N - 1
                    next_state_index_high = N - 1
                    weight_low = 1.0
                    weight_high = 0.0
```

```
                else:
                    next_state_index_low = np.searchsorted(state_space,
 ↪  next_state, side='right') - 1
                    next_state_index_high = next_state_index_low + 1

                    s_low = state_space[next_state_index_low]
                    s_high = state_space[next_state_index_high]

                    weight_high = (next_state - s_low) / (s_high - s_low)
                    weight_low = 1.0 - weight_high

                row_indices.extend([row_index, row_index])
                col_indices.extend([next_state_index_low,
 ↪  next_state_index_high])
                data.extend([weight_low, weight_high])

transition_matrix = csr_matrix((data, (row_indices, col_indices)), shape=(N *
 ↪  nA, N))

# Bellman Function
def bellman(v, U_flat, transition_matrix, delta, N, nA):
    v_next = transition_matrix.dot(v)  # (N * nA,)
    B_flat = U_flat + delta * v_next   # (N * nA,)
    B_sa = B_flat.reshape(N, nA)       # (N, nA)
    return B_sa

# Value Function Iteration
def value_function_iteration(U_flat, transition_matrix, delta, N, nA,
 ↪  tolerance, max_iterations):
    v = np.zeros(N)
    for iteration in range(max_iterations):
        B_sa = bellman(v, U_flat, transition_matrix, delta, N, nA)
        v_new = np.max(B_sa, axis=1)
        policy = np.argmax(B_sa, axis=1)
        diff = np.max(np.abs(v_new - v))
        if diff < tolerance:
            print(f'Converged in {iteration + 1} iterations')
            break
        v = v_new
    else:
        print('Did not converge within max iterations')
    return v, policy
```

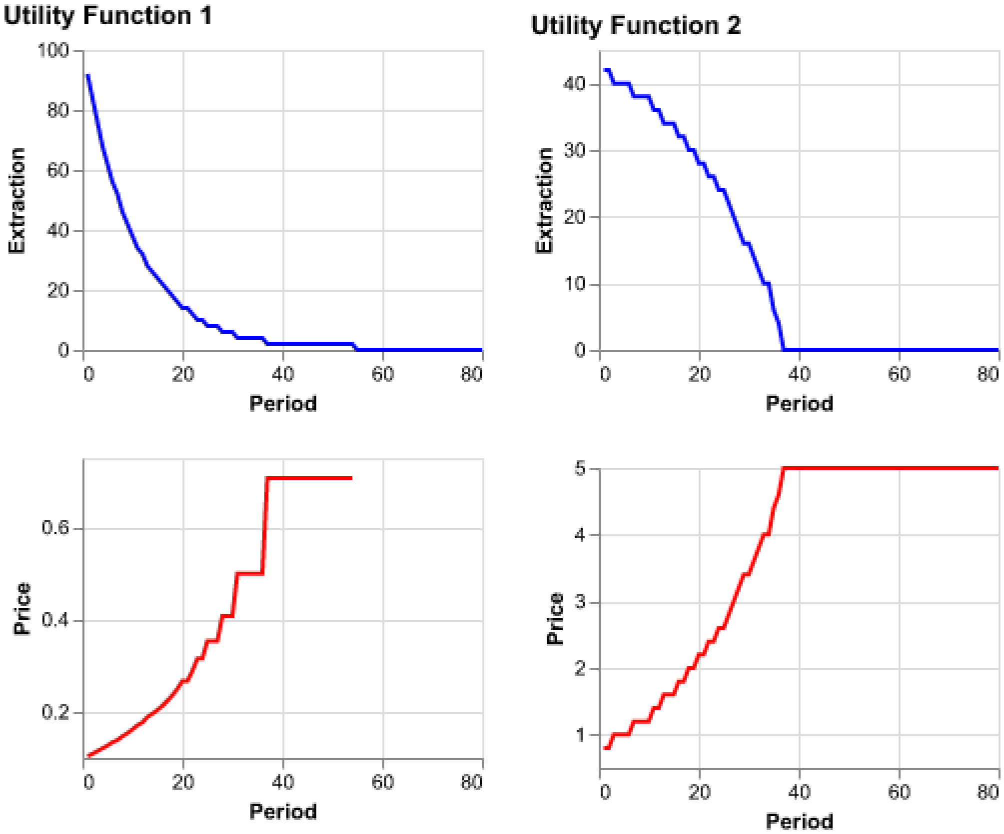
```
# Get Optimal Transition Matrix
def get_optimal_transition_matrix(N, nA, transition_matrix, C):
    row_indices = []
    col_indices = []
    data = []

    for i in range(N):
        optimal_action = C[i]
        row_index = i * nA + optimal_action
        row = transition_matrix.getrow(row_index)
        cols = row.indices
        probs = row.data
        for col, prob in zip(cols, probs):
            if prob > 0:
                row_indices.append(i)
                col_indices.append(col)
                data.append(prob)

    Topt = csr_matrix((data, (row_indices, col_indices)), shape=(N, N))
    return Topt

# Value function iteration function
max_iterations = 1000
np.random.seed(454)
# start with value function  of zeros
value_function = np.zeros(N)
tolerance = 1e-8

# Solve for utility function 1
print("Solving for utility function 1:")
v_u1, C_u1 = value_function_iteration(utility_matrix1_flat,
 ↪  transition_matrix, delta, N, nA, tolerance, max_iterations)

# Part (h) Find optimal transition matrix
Topt_u1 = get_optimal_transition_matrix(N, nA, transition_matrix, C_u1)


# Solve for utility function 2
print("Solving for utility function 2:")
v_u2, C_u2 = value_function_iteration(utility_matrix2_flat,
 ↪  transition_matrix, delta, N, nA, tolerance, max_iterations)
```

```
# Part (h) Find optimal transition matrix
Topt_u2 = get_optimal_transition_matrix(N, nA, transition_matrix, C_u2)

def simulation(T, starting_stock, price_function, C, action_space,
 ↪  state_space):
    remaining_stock = starting_stock
    extraction_path = []
    stock_path = []
    price_path = []
    for t in range(T):
        stock_path.append(remaining_stock)
        current_state = np.searchsorted(state_space, remaining_stock,
 ↪  side='right') - 1
        if current_state < 0 or current_state >= N:
            print("Error: State index out of bounds")
            break
        action_index = C[current_state]
        action = action_space[action_index]
        extraction_path.append(action)
        price = price_function(action)
        price_path.append(price)
        remaining_stock -= action
        if remaining_stock < 0:
            break
    return extraction_path, stock_path, price_path

def make_extraction_plot(extraction_path, stock_path, price_path, title):
    periods = np.arange(1, len(extraction_path) + 1)
    df = pd.DataFrame({
        'Period': periods,
        'Extraction': extraction_path,
        'Price': price_path,
        'Stock': stock_path
    })

    extraction_chart = alt.Chart(df).mark_line(color='blue').encode(
        x=alt.X('Period', axis=alt.Axis(title='Period')),
        y=alt.Y('Extraction', axis=alt.Axis(title='Extraction'))
    ).properties(
        width=200,
        height=150
```

```
    )

    price_chart = alt.Chart(df).mark_line(color='red').encode(
        x=alt.X('Period', axis=alt.Axis(title='Period')),
        y=alt.Y('Price', axis=alt.Axis(title='Price'),
 ↪  scale=alt.Scale(zero=False))
    ).properties(
        width=200,
        height=150
    )

    combined_chart = alt.vconcat(extraction_chart, price_chart).properties(
        title=title
    )
    return combined_chart

# Part (i) Simulate the model for t=80 periods
T = 80

# Simulation for Utility Function 1
extraction_path, stock_path, price_path = simulation(
    T, stock, marginal_utility1, C_u1, action_space, state_space
)

# round prices to 3 decimal places
price_path = np.round(price_path, 3)
chart_1 = make_extraction_plot(extraction_path, stock_path, price_path,
 ↪  "Utility Function 1")

# Simulation for Utility Function 2
extraction_path, stock_path, price_path = simulation(
    T, stock, marginal_utility2, C_u2, action_space, state_space
)
chart_2 = make_extraction_plot(extraction_path, stock_path, price_path,
 ↪  "Utility Function 2")

# Display the charts
chart_1 | chart_2
```

```
Solving for utility function 1:
Converged in 58 iterations
Solving for utility function 2:
```

```
Converged in 39 iterations
```



Question 3

```python
# Part (a): Define extraction functions
def extraction_function1(mu_0, t):
    """
    Computes the extraction rate y1 at time t given initial mu_0.
    """
    mu_t = mu_0 * exp(0.05 * t)
    return 1 / (mu_t ** 2)

def extraction_function2(mu_0, t):
    """
    Computes the extraction rate y2 at time t given initial mu_0.
```

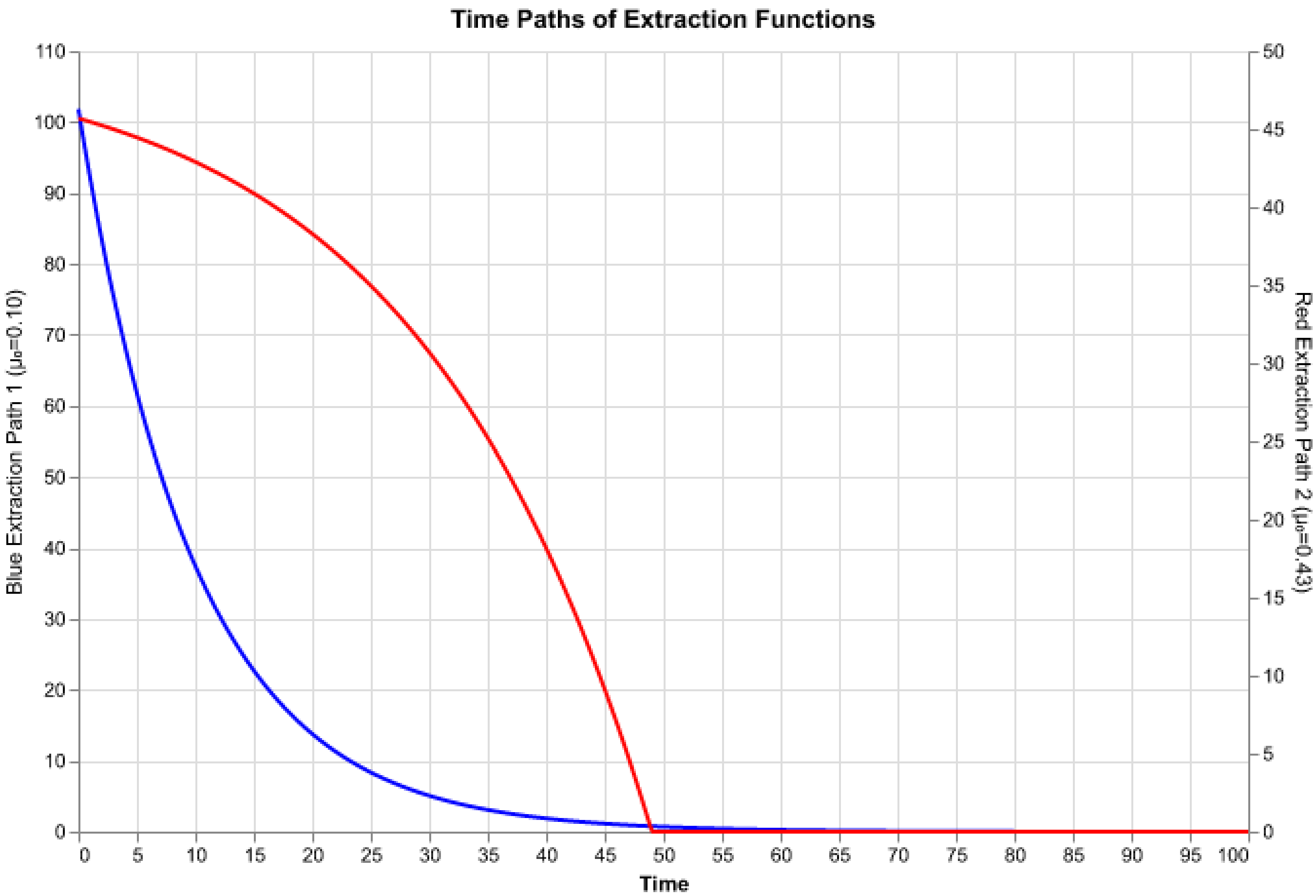
```
    """
    mu_t = mu_0 * exp(0.05 * t)
    return 50 - 10 * mu_t

# Part (b): Compute cumulative extraction over time

def cumulative_extraction1(mu_0, k):
    """
    Computes the cumulative extraction of y1 from time 0 to k.
    """
    result, _ = integrate.quad(lambda t: extraction_function1(mu_0, t), 0, k)
    return result

def cumulative_extraction2(mu_0, k):
    """
    Computes the cumulative extraction of y2 from time 0 to k.
    """
    result, _ = integrate.quad(lambda t: extraction_function2(mu_0, t), 0, k)
    return result

# Part (c): Find mu_0 that results in cumulative extraction close to 1000

def find_closest_mu(func_cumulative_extraction, target=1000,
 ↪  mu_values=np.linspace(0.5, 0.01, 100)):
    """
    Finds the value of mu_0 that brings cumulative extraction closest to the
 ↪  target.
    """
    closest_mu = min(mu_values, key=lambda mu:
 ↪  abs(func_cumulative_extraction(mu)[0] - target))
    return closest_mu

def cumulative_extraction_with_time1(mu_0):
    """
    Finds cumulative extraction and time k where cumulative extraction
 ↪  reaches or exceeds the target for y1.
    """
    for k in range(1001):
        cumulative_value = cumulative_extraction1(mu_0, k)
        if cumulative_value >= 1000:
            return cumulative_value, k
    return cumulative_value, 1000
```

```
def cumulative_extraction_with_time2(mu_0):
    """
    Finds cumulative extraction and time k where cumulative extraction
 ↪  reaches or exceeds the target for y2.
    """
    for k in range(1001):
        cumulative_value = cumulative_extraction2(mu_0, k)
        if cumulative_value >= 1000:
            return cumulative_value, k
    return cumulative_value, 1000

# Generate a range of mu_0 values to search
mu_values = np.linspace(0.5, 0.01, 100)

# Find the mu_0 that gives cumulative extraction closest to 1000
closest_mu1 = find_closest_mu(cumulative_extraction_with_time1,
 ↪  mu_values=mu_values)
closest_mu2 = find_closest_mu(cumulative_extraction_with_time2,
 ↪  mu_values=mu_values)

print(f"The value of mu_0 that gives cumulative extraction closest to 1000
 ↪  for extraction function 1 is: {closest_mu1}")
print(f"The value of mu_0 that gives cumulative extraction closest to 1000
 ↪  for extraction function 2 is: {closest_mu2}")

# Part (d): Plot the time paths of the extraction functions

def generate_extraction_paths(mu_0_1, mu_0_2, time_horizon=100,
 ↪  num_points=500):
    """
    Generates extraction paths for y1 and y2 over the specified time horizon.
    """
    t_values = np.linspace(0, time_horizon, num_points)
    y1_values = []
    y2_values = []

    for t in t_values:
        y1_val = extraction_function1(mu_0_1, t)
        y2_val = extraction_function2(mu_0_2, t)

        y1_values.append(max(y1_val, 0))  # Ensure non-negative values
```

```python
        y2_values.append(max(y2_val, 0))

    return t_values, y1_values, y2_values

# Generate the extraction paths
t_values, y1_values, y2_values = generate_extraction_paths(closest_mu1,
 ↪  closest_mu2)

# Create a DataFrame for plotting
df = pd.DataFrame({
    'Time': t_values,
    'Extraction Path 1': y1_values,
    'Extraction Path 2': y2_values
})

# Create the plots using Altair
def create_extraction_plot(df, mu_0_1, mu_0_2):
    """
    Creates an Altair plot showing the extraction paths of y1 and y2.
    """
    base = alt.Chart(df).encode(x='Time')

    y1_chart = base.mark_line(color='blue').encode(
        y=alt.Y('Extraction Path 1', axis=alt.Axis(title=f'Blue Extraction
 ↪  Path 1 ( ={mu_0_1:.2f})'))
    )

    y2_chart = base.mark_line(color='red').encode(
        y=alt.Y('Extraction Path 2', axis=alt.Axis(title=f'Red Extraction
 ↪  Path 2 ( ={mu_0_2:.2f})'))
    )

    combined_chart = alt.layer(y1_chart, y2_chart).resolve_scale(
        y='independent'
    ).properties(
        width=600,
        height=400,
        title='Time Paths of Extraction Functions'
    )
    return combined_chart

# Display the combined chart
```

```python
chart = create_extraction_plot(df, closest_mu1, closest_mu2)
chart.show()
```

```
The value of mu_0 that gives cumulative extraction closest to 1000 for
extraction function 1 is: 0.09909090909090912
The value of mu_0 that gives cumulative extraction closest to 1000 for
extraction function 2 is: 0.4307070707070707
```



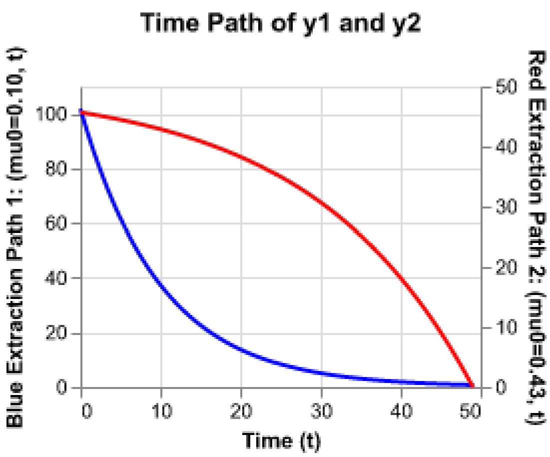

```python
from math import exp
import numpy as np
import scipy.integrate as integrate
import matplotlib.pyplot as plt

## (a)
def y1(mu_0, t):
```

```
    mu_t = mu_0 * exp(0.05 * t)
    return 1 / (mu_t ** 2)

def y2(mu_0, t):
    mu_t = mu_0 * exp(0.05 * t)
    return 50 - 10 * mu_t

## (b)
def cum1(mu_0, k):
    result, _ = integrate.quad(lambda t: y1(mu_0, t), 0, k)
    return result

def cum2(mu_0, k):
    result, _ = integrate.quad(lambda t: y2(mu_0, t), 0, k)
    return result

## (c)
def cum_mu_1(mu):
    for k in range(1001):  # Loop from k = 0 to k = 1000
        cumulative_value = cum1(mu, k)

        # Check if cumulative_value has reached or exceeded 1000
        if cumulative_value >= 1000:
            return cumulative_value, k

    # If we reach k = 1000 without hitting the threshold, return the last
     ↪  value
    return cumulative_value, 1000

def cum_mu_2(mu):
    for k in range(1001):  # Loop from k = 0 to k = 1000
        cumulative_value = cum2(mu, k)

        # Check if cumulative_value has reached or exceeded 1000
        if cumulative_value >= 1000:
            return cumulative_value, k

    # If we reach k = 1000 without hitting the threshold, return the last
     ↪  value
    return cumulative_value, 1000

# get a rough idea of the value of mu
```

```python
mu_values = np.linspace(0.5, 0.01, 100)

# find the value of l that gives the closest value to 1000
closest_mu1 = min(mu_values, key=lambda mu: abs(cum_mu_1(mu)[0] - 1000))
closest_mu2 = min(mu_values, key=lambda mu: abs(cum_mu_2(mu)[0] - 1000))

print(f"The value of mu that gives the closest value to 1000 for cum_mu_1 is:
 ↪  {closest_mu1}")
print(f"The value of mu that gives the closest value to 1000 for cum_mu_2 is:
 ↪  {closest_mu2}")

t_values = np.linspace(0, 100, 500)
y1_values = []
y2_values = []

for t in t_values:
    y1_val = y1(closest_mu1, t)
    y2_val = y2(closest_mu2, t)

    if y1_val < 0:
        break
    y1_values.append(y1_val)

    if y2_val < 0:
        break
    y2_values.append(y2_val)

# add zeros to the end of the shorter list
if len(y1_values) < len(y2_values):
    y1_values.extend([0] * (len(y2_values) - len(y1_values)))
elif len(y2_values) < len(y1_values):
    y2_values.extend([0] * (len(y1_values) - len(y2_values)))

# Create a DataFrame
df = pd.DataFrame({
    'Time': t_values[:len(y1_values)],
    'y1': y1_values,
    'y2': y2_values
})

# Create the y1 plot
y1_chart = alt.Chart(df).mark_line(color='blue').encode(
```

```
    x=alt.X('Time', axis=alt.Axis(title='Time (t)')),
    y=alt.Y('y1', axis=alt.Axis(title=f'Blue Extraction Path 1:
↪   (mu0={closest_mu1:.2f}, t)'))
).properties(
    width=200,
    height=150
)

# Create the y2 plot
y2_chart = alt.Chart(df).mark_line(color='red').encode(
    x=alt.X('Time', axis=alt.Axis(title='Time (t)')),
    y=alt.Y('y2', axis=alt.Axis(title=f'Red Extraction Path 2:
↪   (mu0={closest_mu2:.2f}, t)'))
).properties(
    width=200,
    height=150
)

# Combine the charts
combined_chart = alt.layer(y1_chart, y2_chart).resolve_scale(
    y='independent'
).properties(
    title='Time Path of y1 and y2'
)

combined_chart.show()
```

```
The value of mu that gives the closest value to 1000 for cum_mu_1 is:
0.09909090909090912
The value of mu that gives the closest value to 1000 for cum_mu_2 is:
0.4307070707070707
```



Question 4

```
# Part (a): Define price space and compute profit function

# Define the price space from 0 to 80 with a step size of 1
price_space = np.arange(0, 81, 1)  # Prices from 0 to 80

# Define the profit function
def profit_function(price):
    """
    Compute profit for a given price.

    Parameters:
    - price: Price value or array of prices

    Returns:
    - Profit corresponding to the price(s)
    """
    return price * 100000 - 3000000

# Compute profit for each price in the price space
profit_values = profit_function(price_space)

# Display the profit values
print(f"Profit values for price space:\n{profit_values}")
```

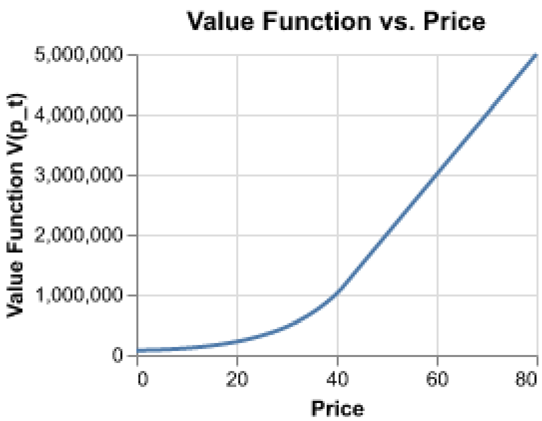
```
# Part (b): Construct the transition matrix

# Parameters for the transition
mean_increment = 0  # Mean of the price increment
std_increment = 4   # Standard deviation of the price increment

# Number of price states
num_price_states = len(price_space)

# Initialize the transition matrix (num_price_states x num_price_states)
transition_matrix = np.zeros((num_price_states, num_price_states))

# Define cutoffs for state transitions
price_cutoffs = np.arange(-0.5, num_price_states, 1)  # Cutoffs between
 ↪  states

# Populate the transition matrix
for current_state_index in range(num_price_states):
    for next_state_index in range(num_price_states):
        if next_state_index == 0:
            lower_bound = -np.inf  # For the first state, lower bound is
 ↪  -infinity
        else:
            lower_bound = price_cutoffs[next_state_index]

        if next_state_index == num_price_states - 1:
            upper_bound = np.inf  # For the last state, upper bound is
 ↪  infinity
        else:
            upper_bound = price_cutoffs[next_state_index + 1]

        # Calculate the probability of transitioning from current state to
         ↪  next state
        probability = norm.cdf(upper_bound,
 ↪  loc=price_space[current_state_index], scale=std_increment) - \
                      norm.cdf(lower_bound,
 ↪  loc=price_space[current_state_index], scale=std_increment)
        transition_matrix[current_state_index, next_state_index] =
 ↪  probability

# Verify that each row of the transition matrix sums to 1
row_sums = transition_matrix.sum(axis=1)
```

```
print("Transition Matrix:")
print(transition_matrix)
print("\nRow sums (should be 1):")
print(row_sums)

# Part (c): Solve for the value function using value function iteration

def value_function_iteration(transition_matrix, profit_values,
↪ discount_factor, tolerance, max_iterations):
    """
    Performs value function iteration to solve for the value function V(p_t).

    Parameters:
    - transition_matrix: State transition probability matrix (num_states x
↪ num_states)
    - profit_values: Profit function values for each state (num_states,)
    - discount_factor: Discount factor
    - tolerance: Convergence tolerance
    - max_iterations: Maximum number of iterations

    Returns:
    - value_function: Value function V(p_t) (num_states,)
    - policy: Optimal decision for each state (num_states,), 1 if choosing
↪ profit, 0 if waiting
    """
    num_states = len(profit_values)
    value_function = np.zeros(num_states)
    policy = np.zeros(num_states, dtype=int)

    for iteration in range(max_iterations):
        value_function_old = value_function.copy()

        # Compute option value (expected continuation value)
        option_value = discount_factor * np.dot(transition_matrix,
↪ value_function_old)

        # Update value function and policy
        value_function = np.maximum(profit_values, option_value)
        policy = (value_function == profit_values).astype(int)

        # Check for convergence
        diff = np.max(np.abs(value_function - value_function_old))
```

```
        if diff < tolerance:
            print(f'Converged in {iteration + 1} iterations')
            break
    else:
        print('Did not converge within max iterations')

    return value_function, policy

# Parameters for value function iteration
discount_factor = delta  # Discount factor
tolerance = 1e-6
max_iterations = 1000

# Solve for the value function and optimal policy
value_function, optimal_policy = value_function_iteration(
    transition_matrix, profit_values, discount_factor, tolerance,
 ↪  max_iterations)

# Find the trigger price (first price where it's optimal to act)
trigger_price = price_space[optimal_policy == 1][0]
print(f"The trigger price is: {trigger_price}")

# Part (d): Plot the value function

def plot_value_function(price_space, value_function):
    """
    Plot the value function V(p_t) against price states.

    Parameters:
    - price_space: Array of price states
    - value_function: Array of value function values
    """
    df = pd.DataFrame({'Price': price_space, 'Value Function':
 ↪  value_function})
    chart = alt.Chart(df).mark_line().encode(
        x=alt.X('Price', axis=alt.Axis(title='Price')),
        y=alt.Y('Value Function', axis=alt.Axis(title='Value Function
 ↪  V(p_t)'))
    ).properties(
        width=200,
        height=150,
        title='Value Function vs. Price'
```

```
    )
    return chart

# Create and display the plot
chart = plot_value_function(price_space, value_function)
chart.show()
```

```
Profit values for price space:
[-3000000 -2900000 -2800000 -2700000 -2600000 -2500000 -2400000 -2300000
 -2200000 -2100000 -2000000 -1900000 -1800000 -1700000 -1600000 -1500000
 -1400000 -1300000 -1200000 -1100000 -1000000  -900000  -800000  -700000
  -600000  -500000  -400000  -300000  -200000  -100000        0   100000
   200000   300000   400000   500000   600000   700000   800000   900000
  1000000  1100000  1200000  1300000  1400000  1500000  1600000  1700000
  1800000  1900000  2000000  2100000  2200000  2300000  2400000  2500000
  2600000  2700000  2800000  2900000  3000000  3100000  3200000  3300000
  3400000  3500000  3600000  3700000  3800000  3900000  4000000  4100000
  4200000  4300000  4400000  4500000  4600000  4700000  4800000  4900000
  5000000]
Transition Matrix:
[[5.49738225e-01 9.64315418e-02 8.78447043e-02 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [4.50261775e-01 9.94764497e-02 9.64315418e-02 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [3.53830233e-01 9.64315418e-02 9.94764497e-02 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 ...
 [6.27307599e-84 7.75594078e-82 9.07800780e-80 ... 9.94764497e-02
  9.64315418e-02 3.53830233e-01]
 [4.72885399e-86 6.22578745e-84 7.75594078e-82 ... 9.64315418e-02
  9.94764497e-02 4.50261775e-01]
 [3.34932479e-88 4.69536075e-86 6.22578745e-84 ... 8.78447043e-02
  9.64315418e-02 5.49738225e-01]]

Row sums (should be 1):
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Converged in 399 iterations
The trigger price is: 41
```


Value Function vs. Price
Value Function V(p_t)
5,000,000
4,000,000
3,000,000
2,000,000
1,000,000
0
0
20
40
60
80
Price