Problem Set 2 Question 3

Each time period, Harold Zurcher chooses to perform maintenance on the bus, or to replace the engine. His flow utilities are given by the following function:

$$u\left(x_t, d_t\right) + \epsilon_{a,t} \;=\; \begin{cases} -\theta_1 x_t - \theta_2 \left(\frac{x_t}{100}\right)^2 + \epsilon_{0,t} & \text{if } d_t = 0 \\ -\theta_3 + \epsilon_{1,t} & \text{if } d_t = 1 \end{cases}$$

where x_t is the current mileage of the bus, d_t is the choice of Harold Zurcher, and θ is a vector of parameters. Each choice also contains unobserved utility $\epsilon_{a,t}$ that are distributed independent T1EV.

Harold Zurcher maximizes his lifetime discounted utility, discounted by β, where the state x_t evolves according to

$$p\left(x_{t+1} \mid x_t, d_t\right) \;=\; \begin{cases} g\left(x_{t+1} - 0\right) & \text{if } d_t = 1 \\ g\left(x_{t+1} - x_t\right) & \text{if } d_t = 0 \end{cases}$$

That is, replacing the engine regenerates the mileage to 0.

Part 1

We can detect engine replacements from the data when the mileage decreases from one period to the next.

In [1]:
```
import numpy as np
import scipy.stats as stats
import pandas as pd
import torch
from torch.autograd import Variable
from torch import logsumexp
import torch.optim as optim
import matplotlib.pyplot as plt
import statsmodels.api as sm
import warnings

warnings.filterwarnings("ignore")
data = pd.read_csv('ps2_ex3.csv')
```

In [2]:
```
data.rename(columns={'milage': 'mileage'}, inplace=True)
data['mileage_old'] = data['mileage'].shift(periods=1, fill_value=0)

data['replace'] = (data['mileage_old'] - data['mileage'] > 0) * 1

x_max = data['mileage'].max()
```

Part 2

From the state transition process, we see that $\left(x_{t+1} \perp \epsilon_{a,t}, \epsilon_{a,t+1}\right) \mid x_t, d_t$. This implies that

$$\begin{aligned} p\left(x_{t+1}, \epsilon_{a,t+1} \mid x_t, d_t, \epsilon_{a,t}\right) &= p\left(x_{t+1} \mid x_t, d_t, \epsilon_{a,t}\right) \cdot p\left(x_{t+1}, \epsilon_{a,t+1} \mid x_t, d_t, \epsilon_{a,t}\right) \\ &= p\left(x_{t+1} \mid x_t, d_t, \epsilon_{a,t}\right) \cdot p\left(\epsilon_{a,t+1} \mid x_t, d_t, \epsilon_{a,t}\right) \\ &= p\left(x_{t+1} \mid x_t, d_t\right) \cdot p\left(\epsilon_{a,t+1} \mid \epsilon_{a,t}\right) \\ &= p\left(x_{t+1} \mid x_t, d_t\right) \cdot g(\epsilon_{t+1}) \end{aligned}$$

Part 3

The following function creates the transition matrix for any level of discretization of the state space. We use the trick from Rust's paper to set $\mathbb{P}(x'|x,1) = \mathbb{P}(x'|0,0)$.

In [3]:
```
def generate_trans_mat(dataset, length_inter):

    """
    This function estimates the transition matrix pertaining to the evolution of the observed state variable x_t.
```

```
    :param dataset: The data with mileage information.
    :param length_inter: The length of chunk to be used to discretize the domain of the state variable.
    :return: A transition matrix of size (number_of_intervals, number_of_intervals, 2)
    """

    k = length_inter
    x_max_round = round(x_max / k) + 1
    trans_mat = np.zeros((x_max_round, x_max_round, 2)
)
    # Create transition matrix for a = 0
    i = 0

    # Extract data with the same decision: replace (1) or not replace (0).
    data_x = dataset.query('replace == {}'.format(i))

    # Rounding mileage using the interval size provided.
    data_x['mileage'] = k * round(data_x['mileage'] / k)
    data_x['mileage_old'] = k * round(data_x['mileage_old'] / k)

    # Constructing the counts of transition from each value of mileage_old to mileage.
    data_x = data_x.groupby(['mileage_old', 'mileage'])['mileage'].count().rename('count').reset_index()
    data_x = data_x.merge(data_x.groupby(['mileage_old'])['count'].sum().rename('total').reset_index(), on='mileag

    # Construct transition probabilities from the counts of transitions.
    data_x['prob'] = data_x['count']/data_x['total']

    # Encode probability data into a transition matrix.
    trans_mat[(data_x['mileage_old']/k).astype(int),
              (data_x['mileage']/k).astype(int), i] = data_x['prob']

    # Add 1s in appropriate cells to ensure that the Markov transition matrices are valid.
    for inter in range(x_max_round):

        if trans_mat[inter, :, i].sum() == 0:

            if i == 0 and inter < x_max_round - 1: # Don't replace engine, mileage goes to next state
                trans_mat[inter, inter + 1, i] = 1

            elif i == 0 and inter == x_max_round - 1:  # Don't replace, boundary
                trans_mat[inter, inter, i] = 1

    # Set the transition matrix for a = 1 to values from Pr(x'|0, 0)
    trans_mat[:, :, 1] = np.repeat(trans_mat[0, :, 0][None, :],
                                   repeats=x_max_round, axis=0)
    # Need to replace zeros with infinitesimal values to not run into
    # overflow issues.
    trans_mat = trans_mat + 1e-6
    trans_mat = trans_mat / trans_mat.sum(axis=1)[:, None, :]

    return data_x, torch.tensor(trans_mat, dtype=torch.double)
```

In [4]:
```
K = 20
```

In [5]:
```
_, trans_matrix = generate_trans_mat(dataset=data, length_inter=K)
```

Part 4

The Bellman equation for this problem is given by

$$V(x,\epsilon) \;=\; \max_{a\in\{0,1\}} \left\{ u(x,a) + \epsilon_j + \beta \int_{x'} \int_{\epsilon'} V\left(x',\epsilon'\right) p\left(\epsilon'\right)\, \mathrm{d}\epsilon' f(x' \mid x,a)\, \mathrm{d}x' \right\}$$

We can define the expected value function $w(x) \;\equiv\; \int_{\epsilon'} V\left(x,\epsilon'\right) p\left(\epsilon'\right)\, \mathrm{d}\epsilon'$. Taking the expected value with respect to ϵ on both sides of the Bellman, we get that

$$
\begin{aligned}
w(x) &= \int_{\epsilon} \max_{a \in \{0,1\}} \left\{ u(x,a) + \epsilon + \beta \int_{x'} \int_{\epsilon'} V\left(y, \epsilon'\right) p\left(\epsilon'\right) \, \mathrm{d}\epsilon' f(x' \mid x, a) \, \mathrm{d}x' \right\} p\left(\epsilon\right) \, \mathrm{d}\epsilon \\
&= \int_{\epsilon} \max_{a \in \{0,1\}} \left\{ u(x,a) + \epsilon_a + \beta \int_{x'} w(x') f(x' \mid x, a) \, \mathrm{d}x' \right\} p\left(\epsilon\right) \, \mathrm{d}\epsilon \\
&= \log \sum_{a=0}^{1} \exp\left\{ u(x,a) + \beta \int_{x'} w(x') f(x' \mid x, a) \, \mathrm{d}x' \right\}
\end{aligned}
$$

We can then take an expectation with respect to x for a given choice d to arrive at the choice-specific value function, defined as $\mathcal{V}(x,d) \equiv \int_{x'} w(x') f(x' \mid x, a) \, \mathrm{d}x'$.

$$
\mathcal{V}(y,d) = \int_{x} \log \sum_{a=0}^{1} \exp\{u(x,a) + \beta \mathcal{V}(x,a)\} f(x \mid y, d) \, \mathrm{d}x
$$

Since we have discretized the state space, we can write that

$$
\mathcal{V}(y,d) = \sum_{x} \left(\log \sum_{a=0}^{1} \exp\{u(x,a) + \beta \mathcal{V}(x,a)\} \right) p(x \mid y, d)
$$

Note that $p(x \mid y, d)$ comes from the transition matrix derived in Part 3, and that $\mathcal{V}$ is a function $\boldsymbol{\theta}$.

Part 5

The conditional choice probability for replacing the engine is given by

$$
\mathbb{P}(a = 1 \mid x; \boldsymbol{\theta}) = \mathbb{P}\left[\mathcal{V}(x,1) + \epsilon_1 > \mathcal{V}(x,0) + \epsilon_0\right] = \frac{\exp(\mathcal{V}(x,1))}{\exp(\mathcal{V}(x,1)) + \exp(\mathcal{V}(x,0))}
$$

Likewise, we can derive that

$$
\mathbb{P}\left(a = 1 \mid x; \boldsymbol{\theta}\right) = \frac{\exp(\mathcal{V}(x,0))}{\exp(\mathcal{V}(x,1)) + \exp(\mathcal{V}(x,0))}
$$

Part 6-8

In [6]:

```
def generate_utility(theta, length_inter):

    k = length_inter
    x_max_round = round(x_max / k) + 1
    state_mat = torch.tensor(np.arange(0, k * x_max_round), dtype=torch.double)

    utility_mat = torch.ones((x_max_round * k, 2), dtype=torch.double) * - theta[2]

    utility_mat[:, 0] = - theta[0] * state_mat - theta[1] * (state_mat ** 2)/10000

    utility_mat = torch.reshape(utility_mat, shape=(x_max_round, k, 2))
    utility_mat = utility_mat.mean(axis=1)

    return utility_mat
```

In [7]:

```
def solve_bellman(theta, beta, trans_mat, length_inter, delta, tol):

    """
    Function that derives the fixed-point solution to the Bellman equation characterizing the
    behavior of the choice specific value function.
    :param theta: Parameters in the utility functions.
    :param beta: Discount factor.
    :param trans_mat: Markov transition matrix corresponding to the states.
    :param length_inter: Discretization level used to characterize the state space.
    :param tol: Tolerance used to determine the solution of the fixed point problem.
    :param delta: Relaxation parameter to arrive at the fixed point solution.
    :return: The fixed-point solution to the EVF Bellman, the CCPs associated with the state space,
     and the number of iterations taken to arrive at the fixed point.
    """
```

```
    evf = torch.ones((trans_matrix.shape[0], 2), dtype=torch.double)
    utility_mat = generate_utility(theta=theta, length_inter=length_inter)

    error, count, max_count = 1, 0, 100000

    while error > tol and count < max_count:

        evf2 = torch.einsum('j, ijk -> ik', logsumexp(input=utility_mat + beta * evf, dim=1), trans_mat)

        error = torch.norm(evf2 - evf)

        evf = delta * evf2 + (1 - delta) * evf

        count = count + 1

        if count % 10000 == 0:
            print('[{}]: error = {}'.format(count, error.detach().numpy()))

    ccp = torch.exp(evf - logsumexp(input=evf, dim=1)[:, None])

    return evf, ccp, count
```

In [8]:
```
# EVF, CCP, _ = solve_bellman(theta=np.array([-0.1, 10, -1]), beta=0.999, trans_mat=trans_matrix, length_inter=K,
```

Part 9

The MLE chooses the parameter set that maximizes the likelihood associated with the data.

$$
\begin{aligned}
\widehat{\boldsymbol{\theta}} &= \arg\max_{\boldsymbol{\theta}} \prod_{t=2}^{T} \left\{ \prod_{j \in \{0,1\}} \mathbb{P}(a_t = j \mid x_t; \boldsymbol{\theta})^{\mathbf{1}\{a_t = j\}} \right\} \mathbb{P}\left(x_t \mid x_{t-1}, a_{t-1}\right) \\
&= \arg\max_{\boldsymbol{\theta}} \sum_{t=2}^{T} \left\{ \sum_{j \in \{0,1\}} \mathbf{1}\left\{a_t = j\right\} \ln \mathbb{P}\left(a_t = j \mid x_t; \boldsymbol{\theta}\right) \right\} + \sum_{t=2}^{T} \ln \mathbb{P}\left(x_t \mid x_{t-1}, a_{t-1}\right)
\end{aligned}
$$

The following function `evaluate_likelihood` first solves the Bellman equation for the expected value function at the given parameter values $\boldsymbol{\theta}$ for a specified discount factor and then evaluates the log-likelihood associated with the data using the conditional choice probabilities derived from the fixed point solution to the Bellman equation as well as the provided Markov matrices for state transition probabilities associated with each possible choice.

In [9]:
```
def evaluate_likelihood(dataset, theta, beta, trans_mat, length_inter, delta, tol):
    """
    A function that computes the loglikelihood associated with the provided choice data based on
    conditional choice probabilities derived from the fixed point of the Bellman equation involving
    the expected value function.

    :param dataset: The data with mileage and choice information.
    :param theta: Parameters in the utility functions.
    :param beta: Discount factor.
    :param trans_mat: Markov transition matrix corresponding to the states.
    :param length_inter: Discretization level used to characterize the state space.
    :param tol: Tolerance used to determine the solution of the fixed point problem.
    :param delta: Relaxation parameter to arrive at the fixed point solution.
    :return: The likelihood value associated with
    """

    k = length_inter

    # CCP solution derived from the solution to the Bellman equation.
    _, ccp, _ = solve_bellman(theta=theta, beta=beta, trans_mat=trans_mat,
                              length_inter=length_inter, delta=delta, tol=tol)

    # Converting mileage values to indices used in the CCP and transition matrices.
    mileage_index = np.array((dataset['mileage'] / k).astype(int)[1:])
    mileage_old_index = np.array((dataset['mileage_old'] / k).astype(int)[1:])
    replace = np.array((dataset['replace']).astype(int)[1:])

    # Likelihood contribution from the conditional choice probabilities.
    ll_term_1 = torch.sum(torch.log(ccp[mileage_index, replace]))

    # Likelihood contribution from the Markov transition probabilities.
    ll_term_2 = torch.sum(torch.log(trans_mat[mileage_index, mileage_old_index, replace]))
```

```
    return ll_term_1 + ll_term_2
```

Part 10

The following function takes an initial guess for the model parameters and estimates them from the dataset using the NFXP + MLE approach.

In [10]:
```
def run_mle(init_guess, dataset, beta, trans_mat, length_inter):

    # Initialize the optimization variables for the ML procedure.
    theta_hat = Variable(init_guess, requires_grad=True)

    init_guess2 = init_guess.detach().clone().numpy()

    # Define the adaptive gradient descent optimizer used to find the estimates.
    opt = optim.Adam([theta_hat], lr=0.1)

    # Define the objective function.
    log_lik = lambda x: evaluate_likelihood(dataset=dataset, theta=x, beta=beta, trans_mat=trans_mat, length_inter

    # Loss vector
    loss_list = list()

    for epoch in range(1000):

        opt.zero_grad() # Reset gradient inside the optimizer

        # Compute the objective at the current parameter values.
        loss = - log_lik(theta_hat)
        loss.backward() # Gradient computed.
        opt.step()

        # Store value of loss.
        loss_list.append(loss.detach().clone())

    return {'init': init_guess2,
            'final': theta_hat.detach().clone().numpy(),
             'loss': loss_list}
```

In [11]:
```
# Create grid of initial conditions used in ML estimation

init_vec = np.linspace(start=-5, stop=10, num=21)
input_grid = np.array(np.meshgrid(init_vec, init_vec, init_vec)).T.reshape(-1, 3)
```

In [ ]:
```
results_list = []


# Run ML estimation for each initial condition.
for ix in range(input_grid.shape[0]):

    result_dict = run_mle(init_guess=torch.tensor(input_grid[ix, :]),
                          dataset=data, beta=0.999,
                          trans_mat=trans_matrix, length_inter=K)

    results_list.append(result_dict)
```

We chose the estimates with the lowest negative log-likelihood. The ML estimates for $\boldsymbol{\theta}$ are:

$$\begin{pmatrix} \hat{\theta}_1 \\ \hat{\theta}_2 \\ \hat{\theta}_3 \end{pmatrix} = \begin{pmatrix} 0.8122 \\ 9.9459 \\ -2.2145 \end{pmatrix}$$

In [19]:
```
theta_hat
```

Out[19]:
```
tensor([ 0.8122,  9.9459, -2.2145], requires_grad=True)
```

In [ ]:
```

```