# EECS 498: Reinforcement Learning
# Homework 4 Responses

Tejas Jha
tjha

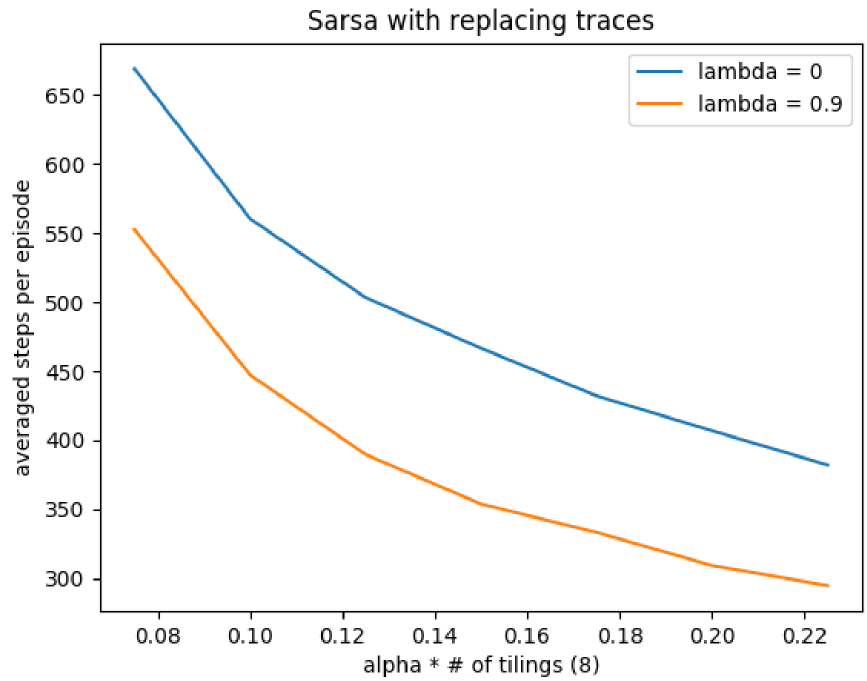
November 27, 2018

This document includes my responses to Homework 4 questions. Responses that involved the use of coding will provide references to specific lines of code to provide a better overview of how the problem was approached. The code can either be referenced in the Appendix or in the accompanied python script submitted with this assignment.

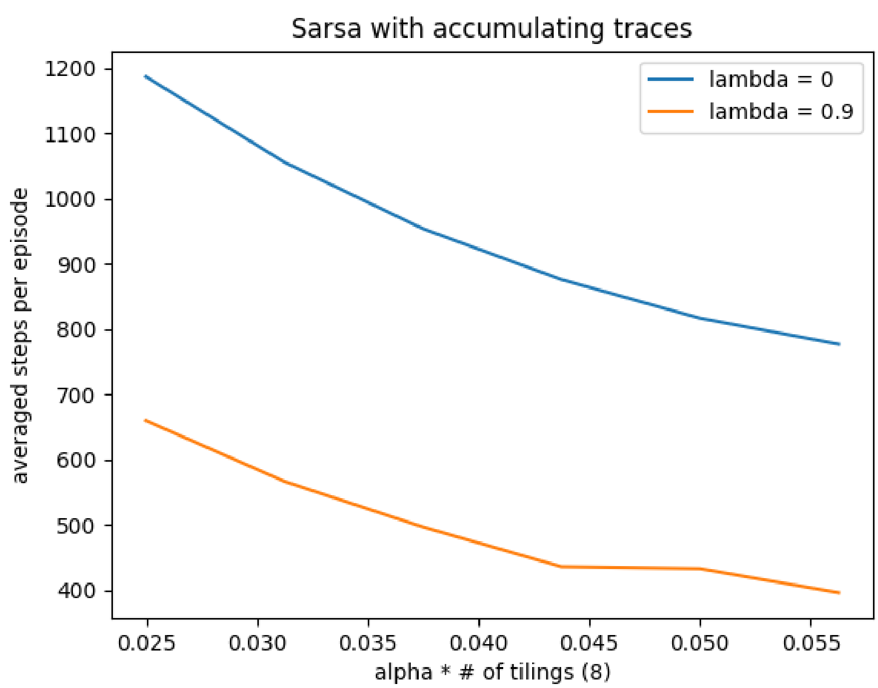
## Question 1

(a) In this specific problem, we are able to set epsilon to 0 because we begin with optimistic initial values. We covered optimistic initial values in the bandit setting, but similarly in this specific problem, these values will encourage initial exploration until values are brought down to a point where the estimated values approach the expected values for the greedy actions. The optimistic initial value we use for the values is 0, which is greater than the true values since a reward of -1 is received for every time step where the car is not at the terminal state.

(b) Below is a generated plot similar to the left part of Figure 12.10 from the textbook for the requested trace and alpha values using replacing traces.




(c) Below is a generated plot similar to the left part of Figure 12.10 from the textbook for the requested trace and alpha values using accumulating traces.

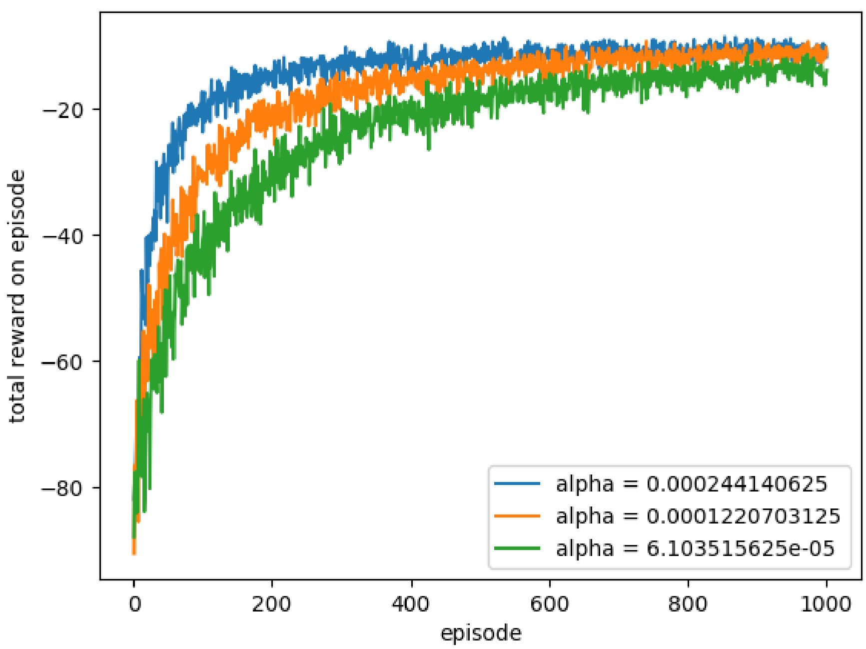


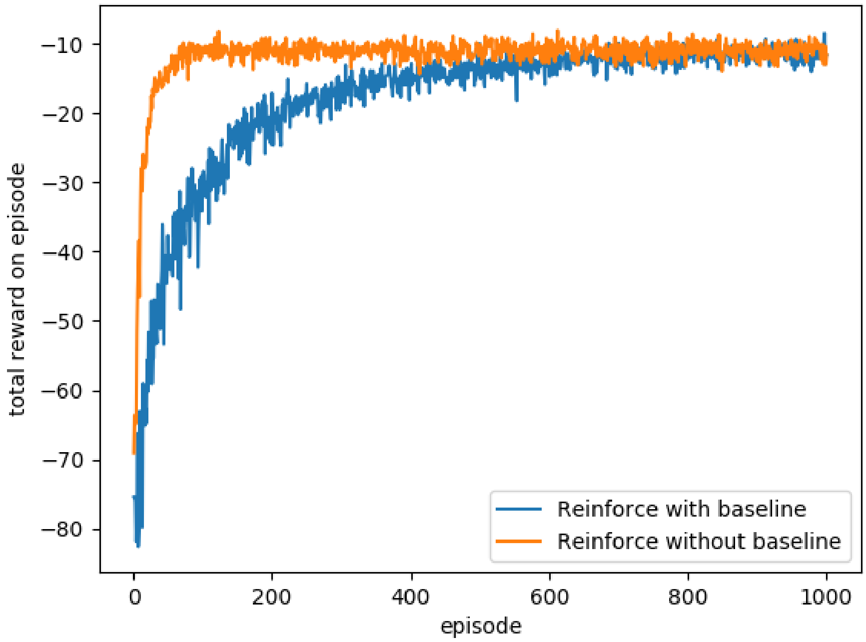

(d) If we change the range of the alphas to be same as that which we used to generate the plot in part (b) for accumulating traces, we can expect a plot with lower average steps than the plot in part (c) where accumulating traces was also used. This will be due to the larger alpha values being looked at which will now focus at the points on the curve that are lower. However, if we were to compare the plot in part (b) that used replacing traces , we will see that the curve for 0 lambda would be the same but the curve for 0.9 lambda would be lower for the accumulating traces. This result would suggest that using accumulating traces for this range of alphas seems to perform better as the averaged step per episode is lower, indicating a more optimal policy. While some searching online seems to indicate that replacing traces generally perform better than accmulating traces (http://www.incompleteideas.net/book/ebook/node80.html) as accumulating traces could end up selecting the ”wrong” action more times resulting in a larger trace corresponding to the ”wrong” action. As a result of this, accumulating traces generally take longer to learn to account for the larger trace for the ”wrong” action. However, from the given alpha values, accumulating traces seems to work better in this situation, indicating that the optimal action could have been chosen earlier and then greedily selected more often with accumulating traces.

## Question 2

Below are the reproduced Figures 13.1 and 13.2 from the textbook. The corresponding code used to genereate these igures can be found in the appendix.

For fig1: Note that the blue curve is for $2^{-12}$, the orange curve is for $2^{-13}$ and the green curve is for $2^{-14}$. Also the data has been averaged over 100 trials.






## Question 3

Below are the attempts made to recreate Figures 4 and 5 from the Options paper. The corresponding code used to genereate the plots for these figures can be found in the appendix.

## Appendix: Relevant Code - tjha_hw4.py

```
# Tejas Jha
# EECS 498 Reinforcement Learning HW 4
# ##########################################################################
# This python script contains all of the relevant code needed to recreate the desired
# plots on the assignment. Comments will spearate the code to better indcated which
# protions corresponding to which parts of the assignment.

import gym
from gym.envs.registration import register
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

```
from tqdm import tqdm
from math import floor, log
from itertools import zip_longest
from tiles3 import IHT

# Environment Generation
register(
    id = 'MountainCar-v1',
    entry_point = 'gym.envs.classic_control:MountainCarEnv',
    max_episode_steps = 5000
)

env = gym.make('MountainCar-v1')
# ################################################################################
#
# cp code
# all possible actions
ACTION_REVERSE = -1
ACTION_ZERO = 0
ACTION_FORWARD = 1
# order is important
ACTIONS = [ACTION_REVERSE, ACTION_ZERO, ACTION_FORWARD]

# bound for position and velocity
POSITION_MIN = -1.2
POSITION_MAX = 0.5
VELOCITY_MIN = -0.07
VELOCITY_MAX = 0.07

# discount is always 1.0 in these experiments
DISCOUNT = 1.0

# use optimistic initial value, so it's ok to set epsilon to 0
EPSILON = 0

# maximum steps per episode
STEP_LIMIT = 5000

# take an @action at @position and @velocity
# @return: new position, new velocity, reward (always -1)
def step(position, velocity, action):
    new_velocity = velocity + 0.001 * action - 0.0025 * np.cos(3 *
        position)
    new_velocity = min(max(VELOCITY_MIN, new_velocity), VELOCITY_MAX)
    new_position = position + new_velocity
    new_position = min(max(POSITION_MIN, new_position), POSITION_MAX)
```

```python
    reward = -1.0
    if new_position == POSITION_MIN:
        new_velocity = 0.0
    return new_position, new_velocity, reward

# accumulating trace update rule
# @trace: old trace (will be modified)
# @activeTiles: current active tile indices
# @lam: lambda
# @return: new trace for convenience
def accumulating_trace(trace, active_tiles, lam):
    trace *= lam * DISCOUNT
    trace[active_tiles] += 1
    return trace

# replacing trace update rule
# @trace: old trace (will be modified)
# @activeTiles: current active tile indices
# @lam: lambda
# @return: new trace for convenience
def replacing_trace(trace, activeTiles, lam):
    active = np.in1d(np.arange(len(trace)), activeTiles)
    trace[active] = 1
    trace[~active] *= lam * DISCOUNT
    return trace

# wrapper class for Sarsa(lambda)
class Sarsa:
    # @maxSize: the maximum # of indices
    def __init__(self, step_size, lam, trace_update=accumulating_trace,
        num_of_tilings=8, max_size=2048):
        self.max_size = max_size
        self.num_of_tilings = num_of_tilings
        self.trace_update = trace_update
        self.lam = lam

        # divide step size equally to each tiling
        self.step_size = step_size / num_of_tilings

        self.hash_table = IHT(max_size)

        # weight for each tile
        self.weights = np.zeros(max_size)

        # trace for each tile
        self.trace = np.zeros(max_size)

```

```
        # position and velocity needs scaling to satisfy the tile software
        self.position_scale = self.num_of_tilings / (POSITION_MAX - POSITION_MIN)
        self.velocity_scale = self.num_of_tilings / (VELOCITY_MAX - VELOCITY_MIN)

    # get indices of active tiles for given state and action
    def get_active_tiles(self, position, velocity, action):
        # I think positionScale * (position - position_min) would be a good normalization.
        # However positionScale * position_min is a constant, so it's ok to ignore it.
        active_tiles = tiles(self.hash_table, self.num_of_tilings,
                             [self.position_scale * position, self.velocity_scale * velocity],
                             [action])
        return active_tiles

    # estimate the value of given state and action
    def value(self, position, velocity, action):
        if position == POSITION_MAX:
            return 0.0
        active_tiles = self.get_active_tiles(position, velocity, action)
        return np.sum(self.weights[active_tiles])

    # learn with given state, action and target
    def learn(self, position, velocity, action, target):
        active_tiles = self.get_active_tiles(position, velocity, action)
        estimation = np.sum(self.weights[active_tiles])
        delta = target - estimation
        if self.trace_update == accumulating_trace or self.trace_update == replacing_trace:
            self.trace_update(self.trace, active_tiles, self.lam)
        else:
            raise Exception('Unexpected Trace Type')
        self.weights += self.step_size * delta * self.trace

    # get # of steps to reach the goal under current state value function
    def cost_to_go(self, position, velocity):
        costs = []
        for action in ACTIONS:
            costs.append(self.value(position, velocity, action))
        return -np.max(costs)

```

```python
# get action at @position and @velocity based on epsilon greedy policy and @valueFunction
def get_action(position, velocity, valueFunction):
    if np.random.binomial(1, EPSILON) == 1:
        return np.random.choice(ACTIONS)
    values = []
    for action in ACTIONS:
        values.append(valueFunction.value(position, velocity, action))
    return np.argmax(values) - 1

# play Mountain Car for one episode based on given method @evaluator
# @return: total steps in this episode
def play(evaluator):
    position = np.random.uniform(-0.6, -0.4)
    velocity = 0.0
    action = get_action(position, velocity, evaluator)
    steps = 0
    while True:
        next_position, next_velocity, reward = step(position, velocity, action)
        next_action = get_action(next_position, next_velocity, evaluator)
        steps += 1
        target = reward + DISCOUNT * evaluator.value(next_position, next_velocity, next_action)
        evaluator.learn(position, velocity, action, target)
        position = next_position
        velocity = next_velocity
        action = next_action
        if next_position == POSITION_MAX:
            break
        if steps >= STEP_LIMIT:
            print('Step Limit Exceeded!')
            break
    return steps

# end cp code
# ##############################################################################

# Question 1 plots
def q1plots():
    runs = 5
    episodes = 50
    lambdas = [0,0.9]

    # Part (b) -    Generation of plot using replacing traces
```

```
    alphas = np.arange(0.6,2.0,0.2) / 8.0
    steps = np.zeros((len(lambdas), len(alphas), runs, episodes))
    for lambdaIdx, lam in enumerate(lambdas):
        for alphaIdx, alpha in enumerate(alphas):
            for run in tqdm(range(runs)):
                evaluator = Sarsa(alpha, lam, replacing_trace, max_size
                    =4096)
                for ep in range(episodes):
                    step = play(evaluator)
                    steps[lambdaIdx, alphaIdx, run, ep] = step

    # average over episodes
    steps = np.mean(steps, axis=3)
    # average over runs
    steps = np.mean(steps, axis=2)

    for lamdaIdx, lam in enumerate(lambdas):
        plt.plot(alphas, steps[lamdaIdx, :], label='lambda = %s' % (str
            (lam)))
    plt.xlabel('alpha * # of tilings (8)')
    plt.ylabel('averaged steps per episode')
    plt.title('Sarsa with replacing traces')
    #plt.ylim([180, 300])
    plt.legend()

    plt.savefig('replacing_traces.png')
    plt.close()

    print("Completed Problem 1 Part (b)")

    # Part (c) -    Generation of plot using accumulating traces
    alphas = np.arange(0.2,0.5,0.05) / 8.0
    steps = np.zeros((len(lambdas), len(alphas), runs, episodes))
    for lambdaIdx, lam in enumerate(lambdas):
        for alphaIdx, alpha in enumerate(alphas):
            for run in tqdm(range(runs)):
                evaluator = Sarsa(alpha, lam, accumulating_trace,
                    max_size=4096)
                for ep in range(episodes):
                    step = play(evaluator)
                    steps[lambdaIdx, alphaIdx, run, ep] = step

    # average over episodes
    steps = np.mean(steps, axis=3)
    # average over runs
    steps = np.mean(steps, axis=2)

    for lamdaIdx, lam in enumerate(lambdas):
```

```
        plt.plot(alphas, steps[lamdaIdx, :], label='lambda = %s' % (str(lam)))
    plt.xlabel('alpha * # of tilings (8)')
    plt.ylabel('averaged steps per episode')
    plt.title('Sarsa with accumulating traces')
    #plt.ylim([180, 300])
    plt.legend()

    plt.savefig('accumulating_traces.png')
    plt.close()

    print("Completed Problem 1 Part (c)")

# ###############################################################################

#
# Work for Q2

class ShortCorridor:
    """
    Short corridor environment, see Example 13.1
    """
    def __init__(self):
        self.reset()

    def reset(self):
        self.state = 0

    def step(self, go_right):
        """
        Args:
            go_right (bool): chosen action
        Returns:
            tuple of (reward, episode terminated?)
        """
        if self.state == 0 or self.state == 2:
            if go_right:
                self.state += 1
            else:
                self.state = max(0, self.state - 1)
        else:
            if go_right:
                self.state -= 1
            else:
                self.state += 1

        if self.state == 3:
```

```python
            # terminal state
            return 0, True
        else:
            return -1, False

def softmax(x):
    t = np.exp(x - np.max(x))
    return t / np.sum(t)

class ReinforceAgent:
    """
    ReinforceAgent that follows algorithm
    'REINFORNCE Monte-Carlo Policy-Gradient Control (episodic)'
    """
    def __init__(self, alpha, gamma):
        # set values such that initial conditions correspond to left-
            epsilon greedy
        self.theta = np.array([-1.47, 1.47])
        self.alpha = alpha
        self.gamma = gamma
        # first column - left, second - right
        self.x = np.array([[0, 1],
                           [1, 0]])
        self.rewards = []
        self.actions = []

    def get_pi(self):
        h = np.dot(self.theta, self.x)
        t = np.exp(h - np.max(h))
        pmf = t / np.sum(t)
        # never become deterministic,
        # guarantees episode finish
        imin = np.argmin(pmf)
        epsilon = 0.05

        if pmf[imin] < epsilon:
            pmf[:] = 1 - epsilon
            pmf[imin] = epsilon

        return pmf

    def get_p_right(self):
        return self.get_pi()[1]

    def choose_action(self, reward):
        if reward is not None:
            self.rewards.append(reward)

```

```python
        pmf = self.get_pi()
        go_right = np.random.uniform() <= pmf[1]
        self.actions.append(go_right)

        return go_right

    def episode_end(self, last_reward):
        self.rewards.append(last_reward)

        # learn theta
        G = np.zeros(len(self.rewards))
        G[-1] = self.rewards[-1]

        for i in range(2, len(G) + 1):
            G[-i] = self.gamma * G[-i + 1] + self.rewards[-i]

        gamma_pow = 1

        for i in range(len(G)):
            j = 1 if self.actions[i] else 0
            pmf = self.get_pi()
            grad_ln_pi = self.x[:, j] - np.dot(self.x, pmf)
            update = self.alpha * gamma_pow * G[i] * grad_ln_pi

            self.theta += update
            gamma_pow *= self.gamma

        self.rewards = []
        self.actions = []

class ReinforceBaselineAgent(ReinforceAgent):
    def __init__(self, alpha, gamma, alpha_w):
        super(ReinforceBaselineAgent, self).__init__(alpha, gamma)
        self.alpha_w = alpha_w
        self.w = 0

    def episode_end(self, last_reward):
        self.rewards.append(last_reward)

        # learn theta
        G = np.zeros(len(self.rewards))
        G[-1] = self.rewards[-1]

        for i in range(2, len(G) + 1):
            G[-i] = self.gamma * G[-i + 1] + self.rewards[-i]

        gamma_pow = 1

```

```
        for i in range(len(G)):
            self.w += self.alpha_w * gamma_pow * (G[i] - self.w)

            j = 1 if self.actions[i] else 0
            pmf = self.get_pi()
            grad_ln_pi = self.x[:, j] - np.dot(self.x, pmf)
            update = self.alpha * gamma_pow * (G[i] - self.w) *
                grad_ln_pi

            self.theta += update
            gamma_pow *= self.gamma

        self.rewards = []
        self.actions = []

def trial(num_episodes, agent_generator):
    env = ShortCorridor()
    agent = agent_generator()

    rewards = np.zeros(num_episodes)
    for episode_idx in range(num_episodes):
        rewards_sum = 0
        reward = None
        env.reset()

        while True:
            go_right = agent.choose_action(reward)
            reward, episode_end = env.step(go_right)
            rewards_sum += reward

            if episode_end:
                agent.episode_end(reward)
                break

        rewards[episode_idx] = rewards_sum

    return rewards

def q2plot1():
    num_trials = 100
    num_episodes = 1000
    alphas = [2**(-12), 2**(-13), 2**(-14)]
    gamma = 1

    for alpha in alphas:

        print(alpha)

```

```
        rewards = np.zeros((num_trials, num_episodes))
        agent_generator = lambda : ReinforceAgent(alpha=alpha, gamma=gamma)

        for i in tqdm(range(num_trials)):
            reward = trial(num_episodes, agent_generator)
            rewards[i, :] = reward

        plt.plot(np.arange(num_episodes) + 1, rewards.mean(axis=0), label='alpha = %s' % (str(alpha)))

    #plt.plot(np.arange(num_episodes) + 1, -11.6 * np.ones(num_episodes), ls='dashed', color='red', label='-11.6')
    #plt.plot(np.arange(num_episodes) + 1, rewards.mean(axis=0), color='blue')
    plt.ylabel('total reward on episode')
    plt.xlabel('episode')
    plt.legend(loc='lower right')

    plt.savefig('fig1.png')
    plt.close()

def q2plot2():
    num_trials = 100
    num_episodes = 1000
    alpha = 2**(-13)
    gamma = 1
    agent_generators = [lambda : ReinforceAgent(alpha=alpha, gamma=gamma),
                        lambda : ReinforceBaselineAgent(alpha=2**(-9), gamma=gamma, alpha_w=2**(-6))]
    labels = ['Reinforce with baseline',
              'Reinforce without baseline']

    rewards = np.zeros((len(agent_generators), num_trials, num_episodes))

    for agent_index, agent_generator in enumerate(agent_generators):
        for i in tqdm(range(num_trials)):
            reward = trial(num_episodes, agent_generator)
            rewards[agent_index, i, :] = reward

    #plt.plot(np.arange(num_episodes) + 1, -11.6 * np.ones(num_episodes), ls='dashed', color='red', label='-11.6')
    for i, label in enumerate(labels):
        plt.plot(np.arange(num_episodes) + 1, rewards[i].mean(axis=0), label=label)
    plt.ylabel('total reward on episode')
```

```
    plt.xlabel('episode')
    plt.legend(loc='lower right')

    plt.savefig('fig2.png')
    plt.close()

if __name__ == '__main__':
    # Ensuring environment is reset to begin
    env.reset()
    # ____________________________________________________________________________________________
    # Question 1 - Implementation of replacing traces and accumulating traces
    #              to produce plots similar to that of Figure 12.10 in the textbook
    # ____________________________________________________________________________________________
    #q1plots()

    # ____________________________________________________________________________________________
    # Quesiton 2 - Reproduction of Figures 13.1 and 13.2 from the textbook.
    #              Plots are generated using example 13.1
    # ____________________________________________________________________________________________
    # Figure 13.1 Reproduciton
    #q2plot1()
    # Figure 13.2 Reproduction
    q2plot2()

    # ____________________________________________________________________________________________
    # Quesiton 3 - Attempts to reproduce the work behind Figures 4 and 5 in the
    #              Options paper. Heatmap figure will be created to represent the results
    # ____________________________________________________________________________________________
    #


```

```
    # Ensuring environment is closed at the end to avoid compilation issues
    env.close()
```