dp_qlbs_oneset_m3_ex2_v3

June 20, 2020

0.1 The QLBS model for a European option

Welcome to your 2nd assignment in Reinforcement Learning in Finance. In this exercise you will arrive to an option price and the hedging portfolio via standard toolkit of Dynamic Pogramming (DP). QLBS model learns both the optimal option price and optimal hedge directly from trading data.

Instructions: - You will be using Python 3. - Avoid using for-loops and while-loops, unless you are explicitly told to do so. - Do not modify the (# GRADED FUNCTION [function name]) comment in some cells. Your work would not be graded if you change this. Each cell containing that comment should only contain one function. - After coding your function, run the cell right below it to check if your result is correct. - When encountering `# dummy code - remove` please replace this code with your own

After this assignment you will: - Re-formulate option pricing and hedging method using the language of Markov Decision Processes (MDP) - Setup foward simulation using Monte Carlo - Expand optimal action (hedge) $a_t^\star(X_t)$ and optimal Q-function $Q_t^\star(X_t, a_t^\star)$ in basis functions with time-dependend coefficients

Let's get started!

0.2 About iPython Notebooks

iPython Notebooks are interactive coding environments embedded in a webpage. You will be using iPython notebooks in this class. You only need to write code between the ### START CODE HERE ### and ### END CODE HERE ### comments. After writing your code, you can run the cell by either pressing "SHIFT"+"ENTER" or by clicking on "Run Cell" (denoted by a play symbol) in the upper bar of the notebook.

We will often specify "(X lines of code)" in the comments to tell you about how much code you need to write. It is just a rough estimate, so don't feel bad if your code is longer or shorter.

```
In [1]: #import warnings
        #warnings.filterwarnings("ignore")

        import numpy as np
        import pandas as pd
        from scipy.stats import norm
        import random
        import time
        import matplotlib.pyplot as plt
        import sys
```

```
        sys.path.append("..")
        import grading
```

```
In [2]: ### ONLY FOR GRADING. DO NOT EDIT ###
        submissions=dict()
        assignment_key="wLtf3SoiEeieSRL7rCBNJA"
        all_parts=["15mYc", "h1P6Y", "q9QW7","s7MpJ","Pa177"]
        ### ONLY FOR GRADING. DO NOT EDIT ###
```

```
In [3]: COURSERA_TOKEN = "N8iS6xLg8xolfTeu"  # the key provided to the Student under his/her ema
        COURSERA_EMAIL =  "chjaynagar@gmail.com"
        # the email
```

0.3 Parameters for MC simulation of stock prices

```
In [4]: S0 = 100       # initial stock price
        mu = 0.05      # drift
        sigma = 0.15   # volatility
        r = 0.03       # risk-free rate
        M = 1          # maturity

        T = 24         # number of time steps
        N_MC = 10000   # number of paths

        delta_t = M / T                # time interval
        gamma = np.exp(- r * delta_t)  # discount factor
```

0.3.1 Black-Sholes Simulation

Simulate N_{MC} stock price sample paths with T steps by the classical Black-Sholes formula.

$$dS_t = \mu S_t dt + \sigma S_t dW_t \qquad S_{t+1} = S_t e^{\left(\mu - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t} Z}$$

where Z is a standard normal random variable.
Based on simulated stock price S_t paths, compute state variable X_t by the following relation.

$$X_t = -\left(\mu - \frac{1}{2}\sigma^2\right) t\Delta t + \log S_t$$

Also compute

$$\Delta S_t = S_{t+1} - e^{r\Delta t} S_t \qquad \Delta \hat{S}_t = \Delta S_t - \Delta \bar{S}_t \qquad t = 0, ..., T-1$$

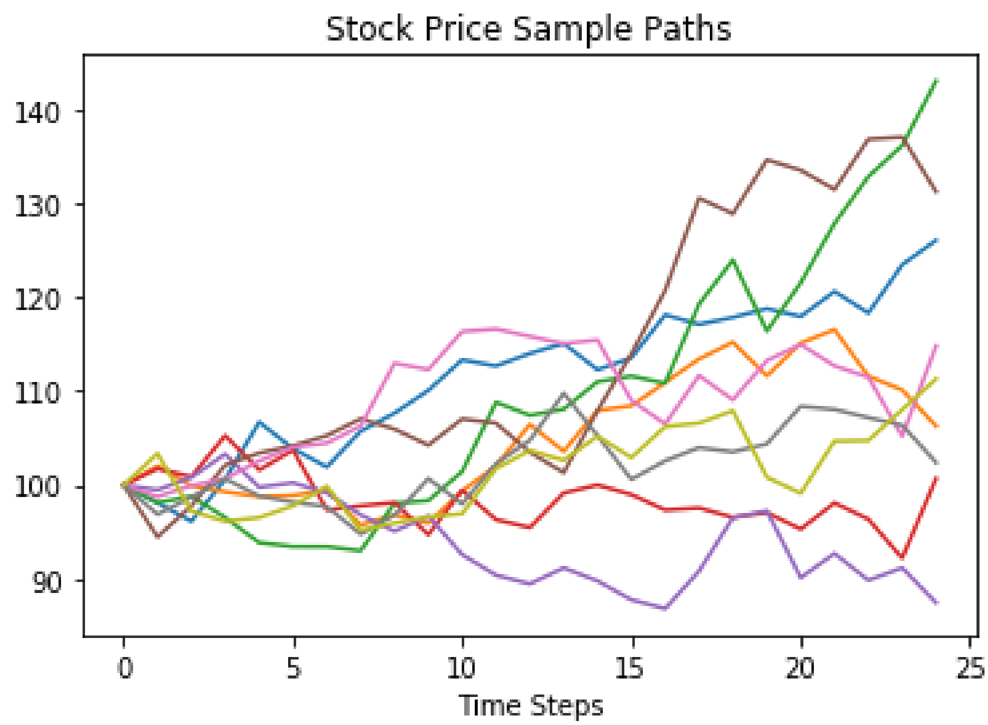
where $\Delta \bar{S}_t$ is the sample mean of all values of ΔS_t.
Plots of 5 stock price S_t and state variable X_t paths are shown below.

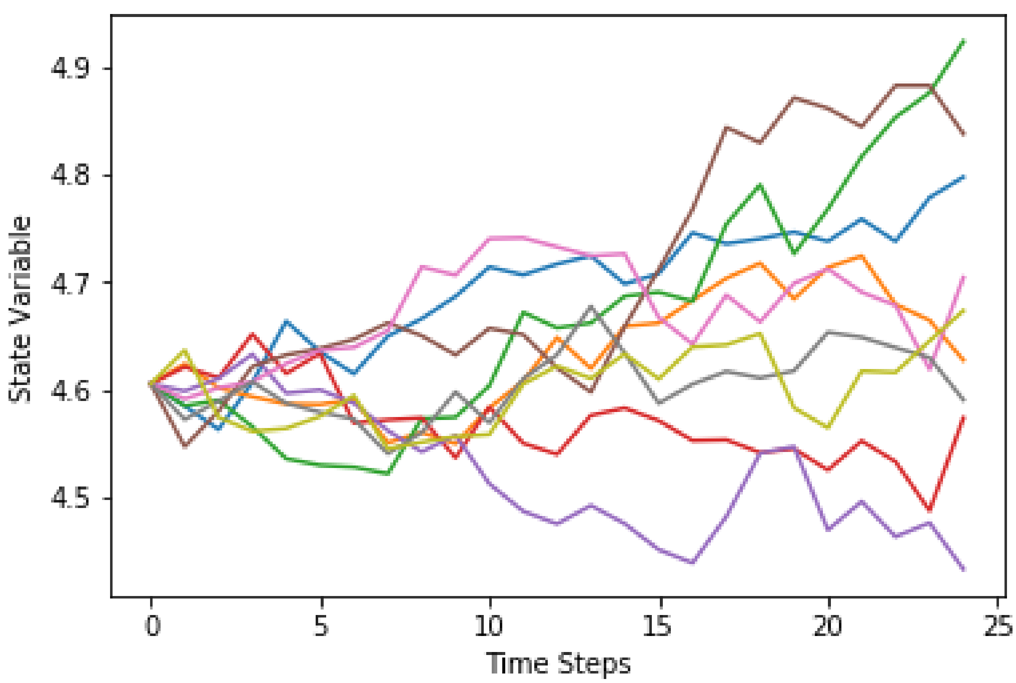
```
In [5]: # make a dataset
        starttime = time.time()
        np.random.seed(42)
```

```
    # stock price
    S = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
    S.loc[:,0] = S0

    # standard normal random numbers
    RN = pd.DataFrame(np.random.randn(N_MC,T), index=range(1, N_MC+1), columns=range(1, T+1)

    for t in range(1, T+1):
        S.loc[:,t] = S.loc[:,t-1] * np.exp((mu - 1/2 * sigma**2) * delta_t + sigma * np.sqrt

    delta_S = S.loc[:,1:T].values - np.exp(r * delta_t) * S.loc[:,0:T-1]
    delta_S_hat = delta_S.apply(lambda x: x - np.mean(x), axis=0)

    # state variable
    X = - (mu - 1/2 * sigma**2) * np.arange(T+1) * delta_t + np.log(S)   # delta_t here is d

    endtime = time.time()
    print('\nTime Cost:', endtime - starttime, 'seconds')
```

```
Time Cost: 0.2365262508392334 seconds
```

```
In [6]: # plot 10 paths
        step_size = N_MC // 10
        idx_plot = np.arange(step_size, N_MC, step_size)

        plt.plot(S.T.iloc[:,idx_plot])
        plt.xlabel('Time Steps')
        plt.title('Stock Price Sample Paths')
        plt.show()

        plt.plot(X.T.iloc[:,idx_plot])
        plt.xlabel('Time Steps')
        plt.ylabel('State Variable')
        plt.show()
```





Define function *terminal_payoff* to compute the terminal payoff of a European put option.

$$H_T(S_T) = \max(K - S_T, 0)$$

```
In [7]: def terminal_payoff(ST, K):
            # ST   final stock price
            # K    strike
            payoff = max(K - ST, 0)
            return payoff
```

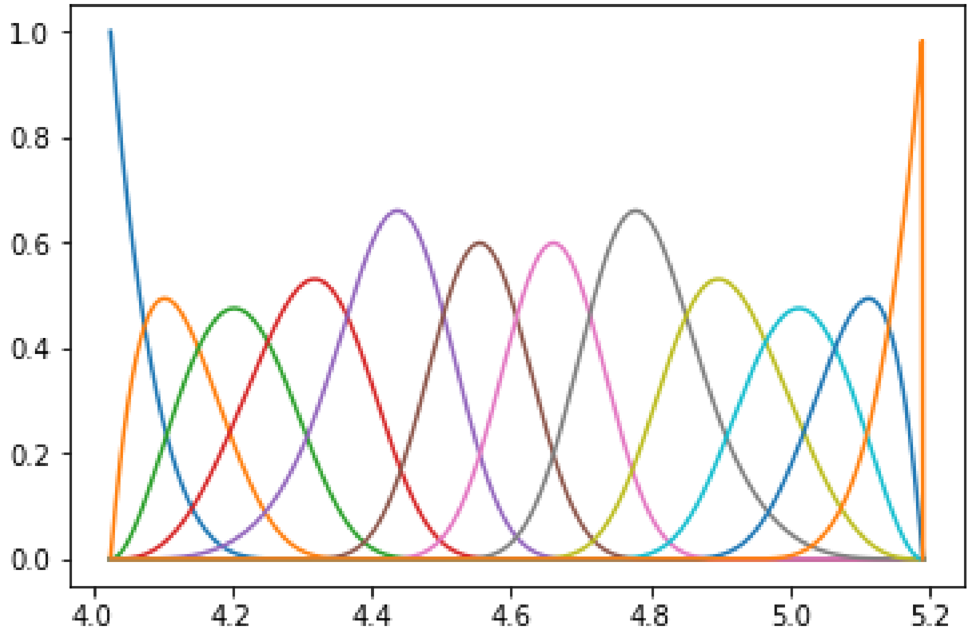
```
In [8]: type(delta_S)
```

```
Out[8]: pandas.core.frame.DataFrame
```

0.4 Define spline basis functions

```
In [9]: import bspline
        import bspline.splinelab as splinelab

        X_min = np.min(np.min(X))
        X_max = np.max(np.max(X))
        print('X.shape = ', X.shape)
        print('X_min, X_max = ', X_min, X_max)

        p = 4              # order of spline (as-is; 3 = cubic, 4: B-spline?)
        ncolloc = 12

        tau = np.linspace(X_min,X_max,ncolloc)  # These are the sites to which we would like to

        # k is a knot vector that adds endpoints repeats as appropriate for a spline of order p
        # To get meaninful results, one should have ncolloc >= p+1
        k = splinelab.aptknt(tau, p)

        # Spline basis of order p on knots k
        basis = bspline.Bspline(k, p)

        f = plt.figure()
        # B   = bspline.Bspline(k, p)     # Spline basis functions
        print('Number of points k = ', len(k))
        basis.plot()

        plt.savefig('Basis_functions.png', dpi=600)
```

```
X.shape =  (10000, 25)
X_min, X_max =  4.0249235249 5.19080277513
Number of points k =  17
```




```
In [10]: type(basis)
```

```
Out[10]: bspline.bspline.Bspline
```

```
In [11]: X.values.shape
```

```
Out[11]: (10000, 25)
```

0.4.1 Make data matrices with feature values

"Features" here are the values of basis functions at data points The outputs are 3D arrays of dimensions num_tSteps x num_MC x num_basis

```
In [12]: num_t_steps = T + 1
         num_basis =  ncolloc # len(k) #

         data_mat_t = np.zeros((num_t_steps, N_MC,num_basis ))
         print('num_basis = ', num_basis)
         print('dim data_mat_t = ', data_mat_t.shape)

         t_0 = time.time()
         # fill it
         for i in np.arange(num_t_steps):
             x = X.values[:,i]
             data_mat_t[i,:,:] = np.array([ basis(el) for el in x ])

         t_end = time.time()
         print('Computational time:', t_end - t_0, 'seconds')
```

```
num_basis =  12
dim data_mat_t =  (25, 10000, 12)
Computational time: 72.58780908584595 seconds
```

```
In [13]: # save these data matrices for future re-use
         np.save('data_mat_m=r_A_%d' % N_MC, data_mat_t)

In [14]: print(data_mat_t.shape)  # shape num_steps x N_MC x num_basis
         print(len(k))
```

```
(25, 10000, 12)
17
```

0.5 Dynamic Programming solution for QLBS

The MDP problem in this case is to solve the following Bellman optimality equation for the action-value function.

$$Q_t^\star\left(x,a\right)=\mathbb{E}_t\left[R_t\left(X_t,a_t,X_{t+1}\right)+\gamma\max_{a_{t+1}\in\mathcal{A}}Q_{t+1}^\star\left(X_{t+1},a_{t+1}\right)|X_t=x,a_t=a\right],t=0,...,T-1,\quad\gamma=e^{-r\Delta t}$$

where $R_t\left(X_t,a_t,X_{t+1}\right)$ is the one-step time-dependent random reward and $a_t\left(X_t\right)$ is the action (hedge).

Detailed steps of solving this equation by Dynamic Programming are illustrated below.

With this set of basis functions $\left\{\Phi_n\left(X_t^k\right)\right\}_{n=1}^N$, expand the optimal action (hedge) $a_t^\star\left(X_t\right)$ and optimal Q-function $Q_t^\star\left(X_t,a_t^\star\right)$ in basis functions with time-dependent coefficients.

$$a_t^\star\left(X_t\right)=\sum_n^N\phi_{nt}\Phi_n\left(X_t\right)\qquad Q_t^\star\left(X_t,a_t^\star\right)=\sum_n^N\omega_{nt}\Phi_n\left(X_t\right)$$

Coefficients ϕ_{nt} and ω_{nt} are computed recursively backward in time for $t=T1,...,0$.

Coefficients for expansions of the optimal action $a_t^\star\left(X_t\right)$ are solved by

$$\phi_t=\mathbf{A}_t^{-1}\mathbf{B}_t$$

where $\mathbf{A}_t$ and $\mathbf{B}_t$ are matrix and vector respectively with elements given by

$$A_{nm}^{(t)}=\sum_{k=1}^{N_{MC}}\Phi_n\left(X_t^k\right)\Phi_m\left(X_t^k\right)\left(\Delta\hat{S}_t^k\right)^2\qquad B_n^{(t)}=\sum_{k=1}^{N_{MC}}\Phi_n\left(X_t^k\right)\left[\hat{\Pi}_{t+1}^k\Delta\hat{S}_t^k+\frac{1}{2\gamma\lambda}\Delta S_t^k\right]$$

$$\Delta S_t=S_{t+1}-e^{-r\Delta t}S_t\quad t=T-1,...,0$$

where $\Delta\hat{S}_t$ is the sample mean of all values of ΔS_t.

Define function *function_A* and *function_B* to compute the value of matrix $\mathbf{A}_t$ and vector $\mathbf{B}_t$.

0.6 Define the option strike and risk aversion parameter

```
In [15]: risk_lambda = 0.001 # risk aversion
         K = 100             # option stike

         # Note that we set coef=0 below in function function_B_vec. This correspond to a pure r
```

0.6.1 Part 1 Calculate coefficients ϕ_{nt} of the optimal action $a_t^\star(X_t)$

Instructions: - implement function_A_vec() which computes $A_{nm}^{(t)}$ matrix - implement function_B_vec() which computes $B_n^{(t)}$ column vector

```
In [16]: # functions to compute optimal hedges
         def function_A_vec(t, delta_S_hat, data_mat, reg_param):
             """
             function_A_vec - compute the matrix A_{nm} from Eq. (52) (with a regularization!)
             Eq. (52) in QLBS Q-Learner in the Black-Scholes-Merton article

             Arguments:
             t - time index, a scalar, an index into time axis of data_mat
             delta_S_hat - pandas.DataFrame of dimension N_MC x T
             data_mat - pandas.DataFrame of dimension T x N_MC x num_basis
             reg_param - a scalar, regularization parameter

             Return:
             - np.array, i.e. matrix A_{nm} of dimension num_basis x num_basis
             """

             ### START CODE HERE ### ( 5-6 lines of code)
             # store result in A_mat for grading
             X_mat = data_mat[t, :, :]
             num_basis_funcs = X_mat.shape[1]
             this_dS = delta_S_hat.loc[:, t]
             hat_dS2 = (this_dS ** 2).reshape(-1, 1)
             A_mat = np.dot(X_mat.T, X_mat * hat_dS2) + reg_param * np.eye(num_basis_funcs)
             ### END CODE HERE ###
             return A_mat


         def function_B_vec(t,
                            Pi_hat,
                            delta_S_hat=delta_S_hat,
                            S=S,
                            data_mat=data_mat_t,
                            gamma=gamma,
                            risk_lambda=risk_lambda):
             """
             function_B_vec - compute vector B_{n} from Eq. (52) QLBS Q-Learner in the Black-Sch
```

```
    Arguments:
    t - time index, a scalar, an index into time axis of delta_S_hat
    Pi_hat - pandas.DataFrame of dimension N_MC x T of portfolio values
    delta_S_hat - pandas.DataFrame of dimension N_MC x T
    S - pandas.DataFrame of simulated stock prices of dimension N_MC x T
    data_mat - pandas.DataFrame of dimension T x N_MC x num_basis
    gamma - one time-step discount factor $exp(-r \delta t)$
    risk_lambda - risk aversion coefficient, a small positive number
    Return:
    np.array() of dimension num_basis x 1
    """
    # coef = 1.0/(2 * gamma * risk_lambda)
    # override it by zero to have pure risk hedge

    ### START CODE HERE ### ( 5-6 lines of code)
    # store result in B_vec for grading
    tmp = Pi_hat.loc[:,t+1] * delta_S_hat.loc[:, t]
    X_mat = data_mat[t, :, :]  # matrix of dimension N_MC x num_basis
    B_vec = np.dot(X_mat.T, tmp)
    ### END CODE HERE ###
    return B_vec
```

In [17]:
```
### GRADED PART (DO NOT EDIT) ###
reg_param = 1e-3
np.random.seed(42)

A_mat = function_A_vec(T-1, delta_S_hat, data_mat_t, reg_param)
idx_row = np.random.randint(low=0, high=A_mat.shape[0], size=50)

np.random.seed(42)
idx_col = np.random.randint(low=0, high=A_mat.shape[1], size=50)


part_1 = list(A_mat[idx_row, idx_col])
try:
    part1 = " ".join(map(repr, part_1))
except TypeError:
    part1 = repr(part_1)


submissions[all_parts[0]]=part1
grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:1],all_parts,s
A_mat[idx_row, idx_col]
### GRADED PART (DO NOT EDIT) ###
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:22: FutureWarning: reshape is depre
```

```
Submission successful, please check on the coursera grader page for the status
```

```
Out[17]: array([ 12261.42554869,   1259.28492179,    176.92982137,  11481.78830269,
                  6579.62177219,  12261.42554869,    628.29798339,    189.70711815,
                 12261.42554869,    176.92982137,    176.92982137,  11481.78830269,
                  6579.62177219,   1259.28492179,  11481.78830269,  11481.78830269,
                   189.70711815,  10408.62274335,   6579.62177219,     18.31727282,
                 11481.78830269,     32.94988345,  10408.62274335,     18.31727282,
                    32.94988345,   6579.62177219,     16.09789819,     32.94988345,
                   628.29798339,  10408.62274335,     32.94988345,   3275.69869791,
                    16.09789819,    176.92982137,    176.92982137,    628.29798339,
                    32.94988345,     32.94988345,    189.70711815,     32.94988345,
                 12261.42554869,   1259.28492179,   3275.69869791,    189.70711815,
                  6579.62177219,    189.70711815,  12261.42554869,   6579.62177219,
                  3275.69869791,  12261.42554869])
```

```
In [18]: ### GRADED PART (DO NOT EDIT) ###
         np.random.seed(42)
         risk_lambda = 0.001
         Pi = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
         Pi.iloc[:,-1] = S.iloc[:,-1].apply(lambda x: terminal_payoff(x, K))

         Pi_hat = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
         Pi_hat.iloc[:,-1] = Pi.iloc[:,-1] - np.mean(Pi.iloc[:,-1])
         B_vec = function_B_vec(T-1, Pi_hat, delta_S_hat, S, data_mat_t, gamma, risk_lambda)

         part_2 = list(B_vec)
         try:
             part2 = " ".join(map(repr, part_2))
         except TypeError:
             part2 = repr(part_2)


         submissions[all_parts[1]]=part2
         grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:2],all_parts,s

         B_vec
         ### GRADED PART (DO NOT EDIT) ###
```

```
Submission successful, please check on the coursera grader page for the status
```

```
Out[18]: array([  3.29073713e+01,  -2.95729027e+02,  -8.73272905e+02,
                 -3.31856654e+03,  -1.25928899e+04,  -1.14032852e+04,
                 -2.91636810e+03,  -3.38216415e+00,  -1.33830723e+02,
                 -1.36875328e+02,  -6.60942460e+01,  -3.07904971e+01])
```

0.7 Compute optimal hedge and portfolio value

Call *function_A* and *function_B* for $t = T-1, ..., 0$ together with basis function $\Phi_n(X_t)$ to compute optimal action $a_t^\star(X_t) = \sum_n^N \phi_{nt}\Phi_n(X_t)$ backward recursively with terminal condition $a_T^\star(X_T) = 0$.

Once the optimal hedge $a_t^\star(X_t)$ is computed, the portfolio value Π_t could also be computed backward recursively by

$$\Pi_t = \gamma\left[\Pi_{t+1} - a_t^\star \Delta S_t\right] \quad t = T-1, ..., 0$$

together with the terminal condition $\Pi_T = H_T(S_T) = \max(K - S_T, 0)$ for a European put option.

Also compute $\hat{\Pi}_t = \Pi_t - \bar{\Pi}_t$, where $\bar{\Pi}_t$ is the sample mean of all values of Π_t.

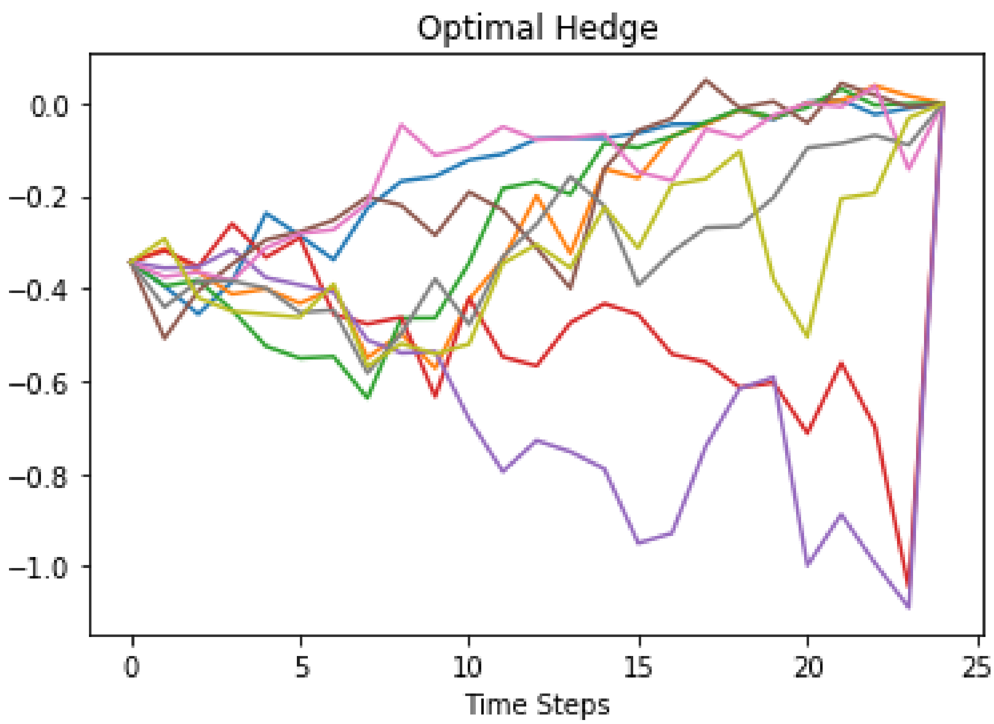
Plots of 5 optimal hedge $a_t^\star$ and portfolio value Π_t paths are shown below.

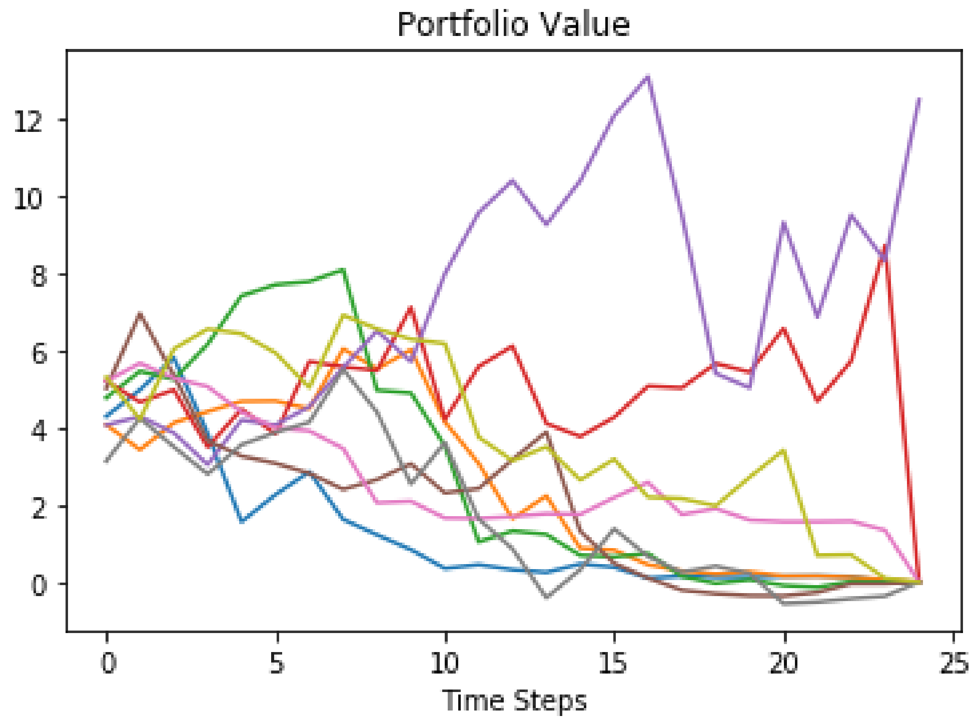
In [19]:
```
starttime = time.time()

# portfolio value
Pi = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
Pi.iloc[:,-1] = S.iloc[:,-1].apply(lambda x: terminal_payoff(x, K))

Pi_hat = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
Pi_hat.iloc[:,-1] = Pi.iloc[:,-1] - np.mean(Pi.iloc[:,-1])

# optimal hedge
a = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
a.iloc[:,-1] = 0

reg_param = 1e-3 # free parameter
for t in range(T-1, -1, -1):
    A_mat = function_A_vec(t, delta_S_hat, data_mat_t, reg_param)
    B_vec = function_B_vec(t, Pi_hat, delta_S_hat, S, data_mat_t, gamma, risk_lambda)
    # print ('t =  A_mat.shape = B_vec.shape = ', t, A_mat.shape, B_vec.shape)

    # coefficients for expansions of the optimal action
    phi = np.dot(np.linalg.inv(A_mat), B_vec)

    a.loc[:,t] = np.dot(data_mat_t[t,:,:],phi)
    Pi.loc[:,t] = gamma * (Pi.loc[:,t+1] - a.loc[:,t] * delta_S.loc[:,t])
    Pi_hat.loc[:,t] = Pi.loc[:,t] - np.mean(Pi.loc[:,t])

a = a.astype('float')
Pi = Pi.astype('float')
Pi_hat = Pi_hat.astype('float')

endtime = time.time()
print('Computational time:', endtime - starttime, 'seconds')
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:22: FutureWarning: reshape is depre
```

```
Computational time: 10.481941938400269 seconds
```

In [20]:
```
# plot 10 paths
plt.plot(a.T.iloc[:,idx_plot])
plt.xlabel('Time Steps')
plt.title('Optimal Hedge')
plt.show()

plt.plot(Pi.T.iloc[:,idx_plot])
plt.xlabel('Time Steps')
plt.title('Portfolio Value')
plt.show()
```





0.8 Compute rewards for all paths

Once the optimal hedge $a_t^\star$ and portfolio value Π_t are all computed, the reward function $R_t\left(X_t, a_t, X_{t+1}\right)$ could then be computed by

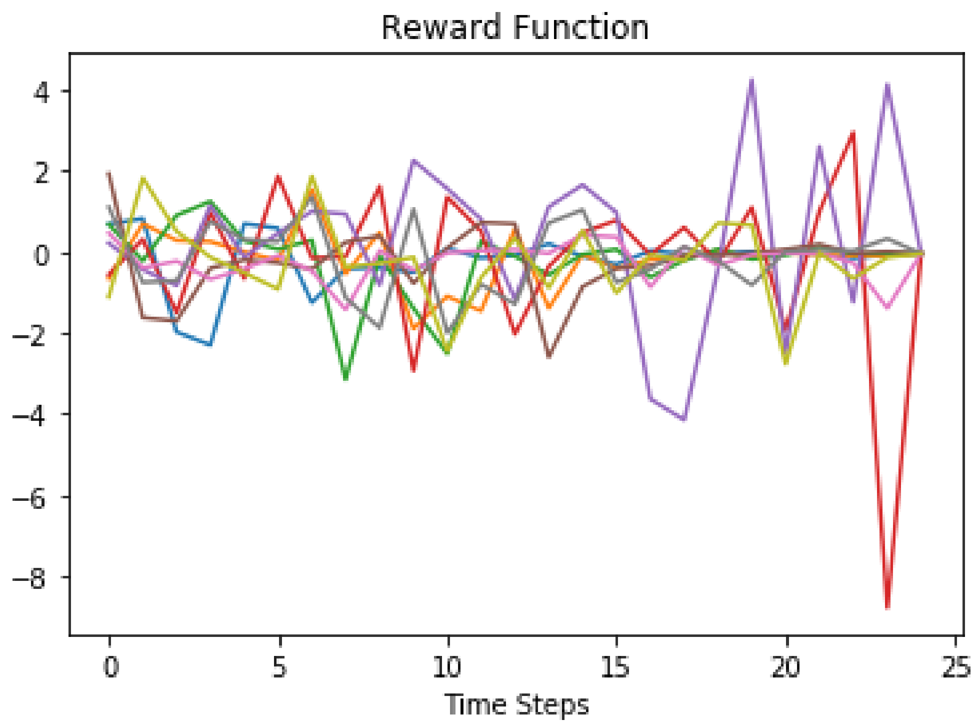
$$R_t\left(X_t, a_t, X_{t+1}\right) = \gamma a_t \Delta S_t - \lambda Var\left[\Pi_t | \mathcal{F}_t\right] \quad t = 0, ..., T-1$$

with terminal condition $R_T = -\lambda Var\left[\Pi_T\right]$.
Plot of 5 reward function R_t paths is shown below.

In [21]:
```
# Compute rewards for all paths
starttime = time.time()
# reward function
R = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
R.iloc[:,-1] = - risk_lambda * np.var(Pi.iloc[:,-1])

for t in range(T):
    R.loc[1:,t] = gamma * a.loc[1:,t] * delta_S.loc[1:,t] - risk_lambda * np.var(Pi.loc

endtime = time.time()
print('\nTime Cost:', endtime - starttime, 'seconds')

# plot 10 paths
plt.plot(R.T.iloc[:, idx_plot])
```

```
plt.xlabel('Time Steps')
plt.title('Reward Function')
plt.show()
```

```
Time Cost: 0.3799169063568115 seconds
```



0.9 Part 2: Compute the optimal Q-function with the DP approach

Coefficients for expansions of the optimal Q-function $Q_t^\star\left(X_t, a_t^\star\right)$ are solved by

$$\omega_t = \mathbf{C}_t^{-1}\mathbf{D}_t$$

where $\mathbf{C}_t$ and $\mathbf{D}_t$ are matrix and vector respectively with elements given by

$$C_{nm}^{(t)} = \sum_{k=1}^{N_{MC}} \Phi_n\left(X_t^k\right)\Phi_m\left(X_t^k\right) \qquad D_n^{(t)} = \sum_{k=1}^{N_{MC}} \Phi_n\left(X_t^k\right)\left(R_t\left(X_t, a_t^\star, X_{t+1}\right) + \gamma \max_{a_{t+1}\in\mathcal{A}} Q_{t+1}^\star\left(X_{t+1}, a_{t+1}\right)\right)$$

Define function *function_C* and *function_D* to compute the value of matrix $\mathbf{C}_t$ and vector $\mathbf{D}_t$.

Instructions: - implement function_C_vec() which computes $C_{nm}^{(t)}$ matrix - implement function_D_vec() which computes $D_n^{(t)}$ column vector

In [22]:
```
def function_C_vec(t, data_mat, reg_param):
    """
    function_C_vec - calculate C_{nm} matrix from Eq. (56) (with a regularization!)
    Eq. (56) in QLBS Q-Learner in the Black-Scholes-Merton article

    Arguments:
    t - time index, a scalar, an index into time axis of data_mat
    data_mat - pandas.DataFrame of values of basis functions of dimension T x N_MC x nu
    reg_param - regularization parameter, a scalar

    Return:
    C_mat - np.array of dimension num_basis x num_basis
    """
    ### START CODE HERE ### ( 5-6 lines of code)
    # your code here ....
    X_mat = data_mat[t, :, :]
    num_basis_funcs = X_mat.shape[1]
    C_mat = np.dot(X_mat.T, X_mat) + reg_param * np.eye(num_basis_funcs)
    # C_mat = your code here ...

    ### END CODE HERE ###
    return C_mat

def function_D_vec(t, Q, R, data_mat, gamma=gamma):
    """
    function_D_vec - calculate D_{nm} vector from Eq. (56) (with a regularization!)
    Eq. (56) in QLBS Q-Learner in the Black-Scholes-Merton article

    Arguments:
    t - time index, a scalar, an index into time axis of data_mat
    Q - pandas.DataFrame of Q-function values of dimension N_MC x T
    R - pandas.DataFrame of rewards of dimension N_MC x T
    data_mat - pandas.DataFrame of values of basis functions of dimension T x N_MC x nu
    gamma - one time-step discount factor $exp(-r \delta t)$

    Return:
    D_vec - np.array of dimension num_basis x 1
    """

    ### START CODE HERE ### ( 5-6 lines of code)
    # your code here ....
    X_mat = data_mat[t, :, :]
    D_vec = np.dot(X_mat.T, R.loc[:,t] + gamma * Q.loc[:, t+1])
    # D_vec = your code here ...

    ### END CODE HERE ###
    return D_vec
```

```
In [23]: ### GRADED PART (DO NOT EDIT) ###
         C_mat = function_C_vec(T-1, data_mat_t, reg_param)
         np.random.seed(42)
         idx_row = np.random.randint(low=0, high=C_mat.shape[0], size=50)

         np.random.seed(42)
         idx_col = np.random.randint(low=0, high=C_mat.shape[1], size=50)

         part_3 = list(C_mat[idx_row, idx_col])
         try:
             part3 = " ".join(map(repr, part_3))
         except TypeError:
             part3 = repr(part_3)


         submissions[all_parts[2]]=part3
         grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:3],all_parts,s

         C_mat[idx_row, idx_col]
         ### GRADED PART (DO NOT EDIT) ###
```

```
Submission successful, please check on the coursera grader page for the status
```

```
Out[23]: array([  1.09774699e+03,   2.10343651e+02,   4.56877655e+00,
                  8.41911156e+02,   8.69395802e+02,   1.09774699e+03,
                  3.30191775e+01,   3.53203253e+01,   1.09774699e+03,
                  4.56877655e+00,   4.56877655e+00,   8.41911156e+02,
                  8.69395802e+02,   2.10343651e+02,   8.41911156e+02,
                  8.41911156e+02,   3.53203253e+01,   1.10328718e+03,
                  8.69395802e+02,   4.35986560e+00,   8.41911156e+02,
                  1.04949534e+00,   1.10328718e+03,   4.35986560e+00,
                  1.04949534e+00,   8.69395802e+02,   2.34165631e+00,
                  1.04949534e+00,   3.30191775e+01,   1.10328718e+03,
                  1.04949534e+00,   1.99059232e+02,   2.34165631e+00,
                  4.56877655e+00,   4.56877655e+00,   3.30191775e+01,
                  1.04949534e+00,   1.04949534e+00,   3.53203253e+01,
                  1.04949534e+00,   1.09774699e+03,   2.10343651e+02,
                  1.99059232e+02,   3.53203253e+01,   8.69395802e+02,
                  3.53203253e+01,   1.09774699e+03,   8.69395802e+02,
                  1.99059232e+02,   1.09774699e+03])
```

```
In [24]: ### GRADED PART (DO NOT EDIT) ###
         Q = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
         Q.iloc[:,-1] = - Pi.iloc[:,-1] - risk_lambda * np.var(Pi.iloc[:,-1])
         D_vec = function_D_vec(T-1, Q, R, data_mat_t,gamma)
```

```
part_4 = list(D_vec)
try:
    part4 = " ".join(map(repr, part_4))
except TypeError:
    part4 = repr(part_4)


submissions[all_parts[3]]=part4
grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:4],all_parts,s

D_vec
### GRADED PART (DO NOT EDIT) ###
```

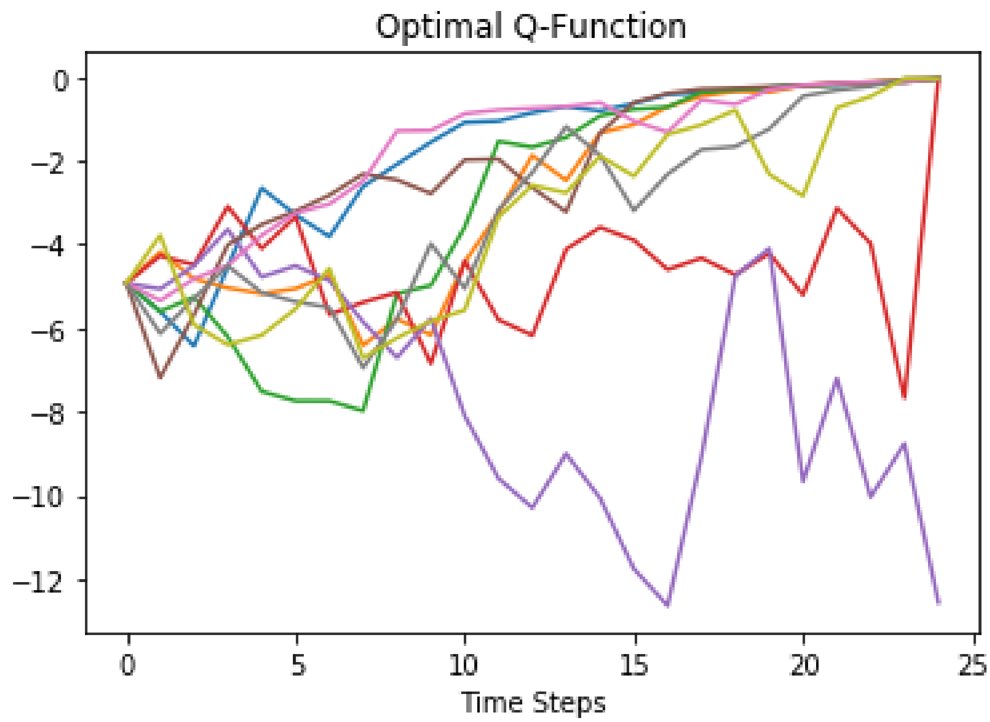
```
Submission successful, please check on the coursera grader page for the status
```

```
Out[24]: array([ -1.33721037e+02,  -5.99514760e+02,  -3.18661973e+03,
                 -1.02120353e+04,  -1.76323018e+04,  -7.20169691e+03,
                 -1.13250111e+03,  -1.66673355e+02,  -5.20254025e+01,
                 -1.55950276e+01,  -5.86197625e+00,  -4.96858215e+00])
```

Call *function_C* and *function_D* for $t = T-1, ..., 0$ together with basis function $\Phi_n(X_t)$ to compute optimal action Q-function $Q_t^\star(X_t, a_t^\star) = \sum_n^N \omega_{nt}\Phi_n(X_t)$ backward recursively with terminal condition $Q_T^\star(X_T, a_T = 0) = -\Pi_T(X_T) - \lambda Var[\Pi_T(X_T)]$.

```
In [25]: starttime = time.time()

         # Q function
         Q = pd.DataFrame([], index=range(1, N_MC+1), columns=range(T+1))
         Q.iloc[:,-1] = - Pi.iloc[:,-1] - risk_lambda * np.var(Pi.iloc[:,-1])

         reg_param = 1e-3
         for t in range(T-1, -1, -1):
             ######################
             C_mat = function_C_vec(t,data_mat_t,reg_param)
             D_vec = function_D_vec(t, Q,R,data_mat_t,gamma)
             omega = np.dot(np.linalg.inv(C_mat), D_vec)

             Q.loc[:,t] = np.dot(data_mat_t[t,:,:], omega)

         Q = Q.astype('float')
         endtime = time.time()
         print('\nTime Cost:', endtime - starttime, 'seconds')

         # plot 10 paths
         plt.plot(Q.T.iloc[:, idx_plot])
         plt.xlabel('Time Steps')
         plt.title('Optimal Q-Function')
         plt.show()
```

```
Time Cost: 8.962746143341064 seconds
```



The QLBS option price is given by $C_t^{(QLBS)}(S_t, ask) = -Q_t(S_t, a_t^\star)$

0.10 Summary of the QLBS pricing and comparison with the BSM pricing

Compare the QLBS price to European put price given by Black-Sholes formula.

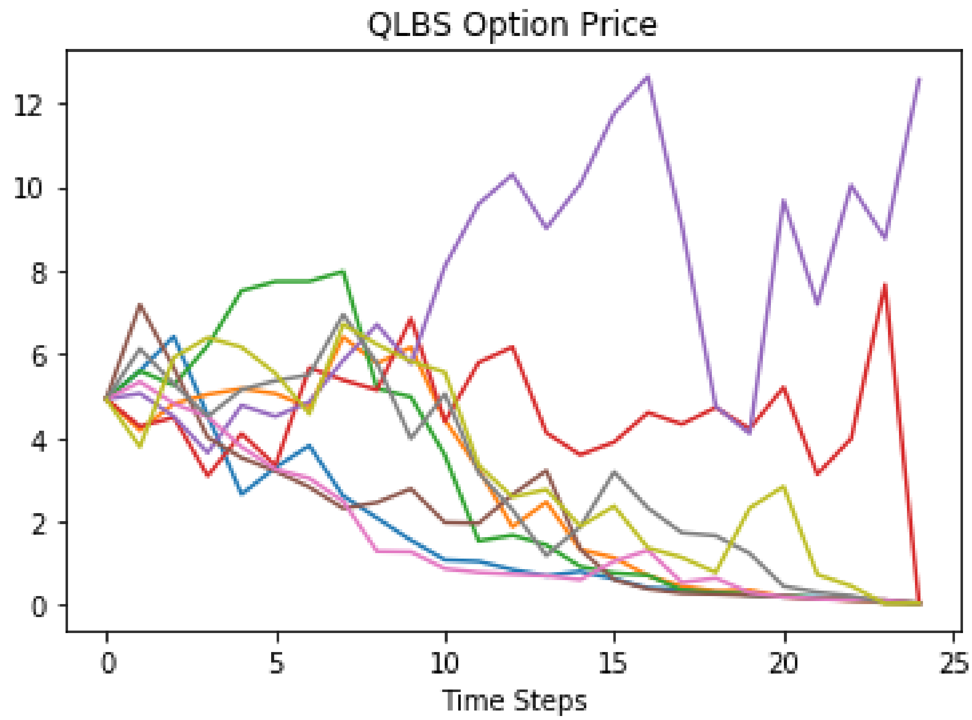
$$C_t^{(BS)} = Ke^{-r(T-t)}\mathcal{N}(-d_2) - S_t\mathcal{N}(-d_1)$$

```
In [26]: # The Black-Scholes prices
         def bs_put(t, S0=S0, K=K, r=r, sigma=sigma, T=M):
             d1 = (np.log(S0/K) + (r + 1/2 * sigma**2) * (T-t)) / sigma / np.sqrt(T-t)
             d2 = (np.log(S0/K) + (r - 1/2 * sigma**2) * (T-t)) / sigma / np.sqrt(T-t)
             price = K * np.exp(-r * (T-t)) * norm.cdf(-d2) - S0 * norm.cdf(-d1)
             return price

         def bs_call(t, S0=S0, K=K, r=r, sigma=sigma, T=M):
             d1 = (np.log(S0/K) + (r + 1/2 * sigma**2) * (T-t)) / sigma / np.sqrt(T-t)
             d2 = (np.log(S0/K) + (r - 1/2 * sigma**2) * (T-t)) / sigma / np.sqrt(T-t)
             price = S0 * norm.cdf(d1) - K * np.exp(-r * (T-t)) * norm.cdf(d2)
             return price
```

0.11 The DP solution for QLBS

In [27]:
```
# QLBS option price
C_QLBS = - Q.copy()

print('---------------------------------------------')
print('       QLBS Option Pricing (DP solution)      ')
print('---------------------------------------------\n')
print('%-25s' % ('Initial Stock Price:'), S0)
print('%-25s' % ('Drift of Stock:'), mu)
print('%-25s' % ('Volatility of Stock:'), sigma)
print('%-25s' % ('Risk-free Rate:'), r)
print('%-25s' % ('Risk aversion parameter: '), risk_lambda)
print('%-25s' % ('Strike:'), K)
print('%-25s' % ('Maturity:'), M)
print('%-26s %.4f' % ('\nQLBS Put Price: ', C_QLBS.iloc[0,0]))
print('%-26s %.4f' % ('\nBlack-Sholes Put Price:', bs_put(0)))
print('\n')

# plot 10 paths
plt.plot(C_QLBS.T.iloc[:,idx_plot])
plt.xlabel('Time Steps')
plt.title('QLBS Option Price')
plt.show()
```

```
---------------------------------------------
       QLBS Option Pricing (DP solution)
---------------------------------------------

Initial Stock Price:      100
Drift of Stock:           0.05
Volatility of Stock:      0.15
Risk-free Rate:           0.03
Risk aversion parameter:  0.001
Strike:                   100
Maturity:                 1

QLBS Put Price:           4.9261

Black-Sholes Put Price:   4.5296
```




```
In [28]: ### GRADED PART (DO NOT EDIT) ###

         part5 = str(C_QLBS.iloc[0,0])
         submissions[all_parts[4]]=part5
         grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:5],all_parts,s

         C_QLBS.iloc[0,0]
         ### GRADED PART (DO NOT EDIT) ###
```

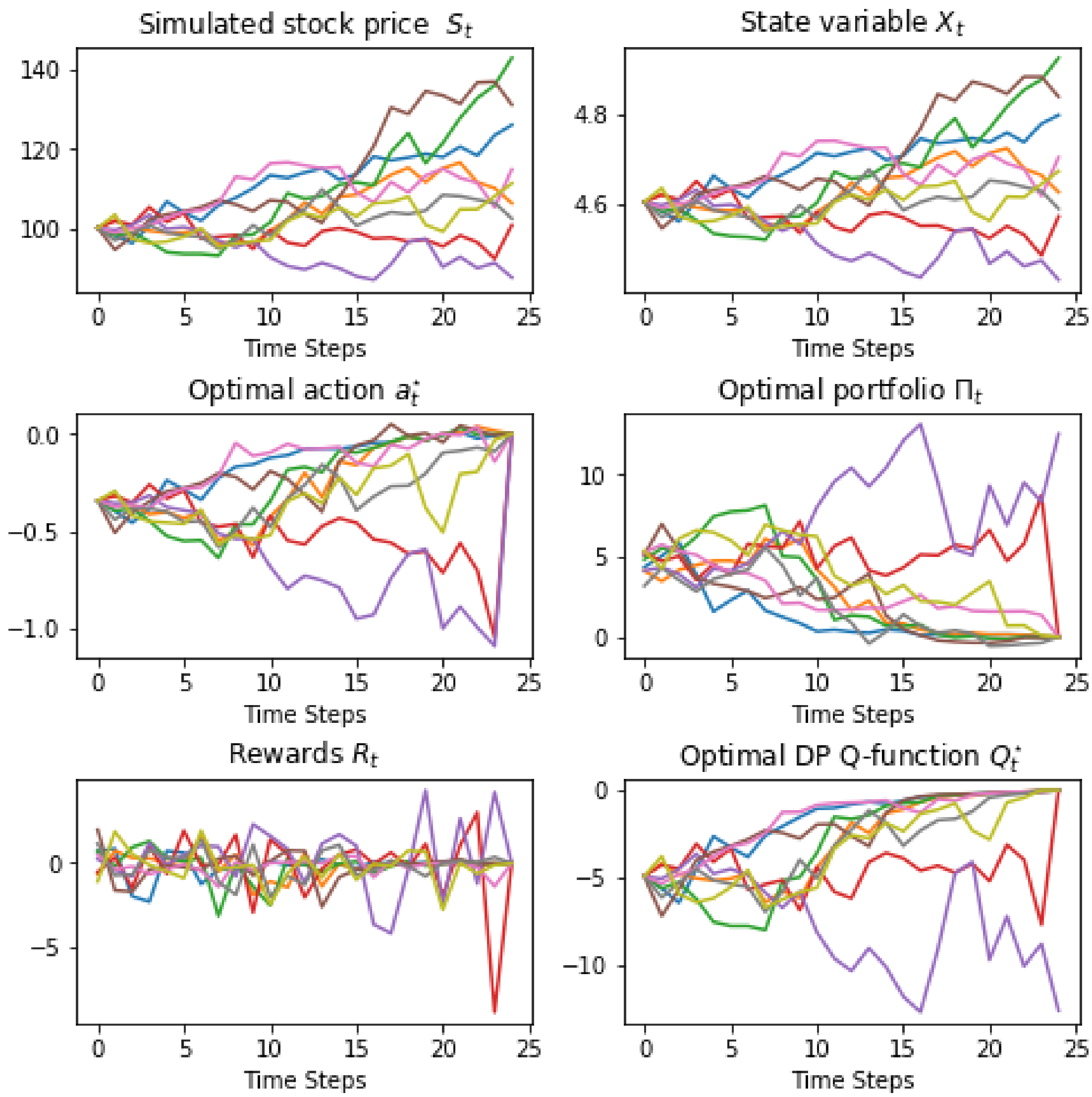
```
Submission successful, please check on the coursera grader page for the status
```

```
Out[28]: 4.9261254187274055
```

0.11.1 make a summary picture

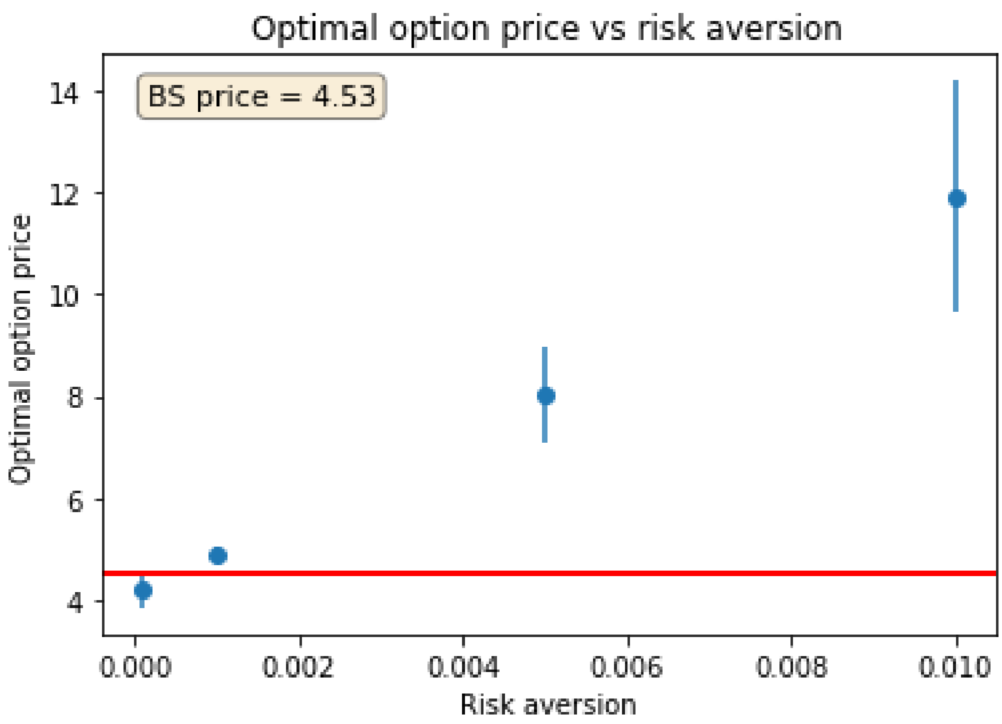
```
In [29]: # plot: Simulated S_t and X_t values
         # optimal hedge and portfolio values
         # rewards and optimal Q-function

         f, axarr = plt.subplots(3, 2)
         f.subplots_adjust(hspace=.5)
         f.set_figheight(8.0)
         f.set_figwidth(8.0)
```

```
axarr[0, 0].plot(S.T.iloc[:,idx_plot])
axarr[0, 0].set_xlabel('Time Steps')
axarr[0, 0].set_title(r'Simulated stock price  $S_t$')

axarr[0, 1].plot(X.T.iloc[:,idx_plot])
axarr[0, 1].set_xlabel('Time Steps')
axarr[0, 1].set_title(r'State variable $X_t$')

axarr[1, 0].plot(a.T.iloc[:,idx_plot])
axarr[1, 0].set_xlabel('Time Steps')
axarr[1, 0].set_title(r'Optimal action $a_t^{\star}$')

axarr[1, 1].plot(Pi.T.iloc[:,idx_plot])
axarr[1, 1].set_xlabel('Time Steps')
axarr[1, 1].set_title(r'Optimal portfolio $\Pi_t$')

axarr[2, 0].plot(R.T.iloc[:,idx_plot])
axarr[2, 0].set_xlabel('Time Steps')
axarr[2, 0].set_title(r'Rewards $R_t$')

axarr[2, 1].plot(Q.T.iloc[:,idx_plot])
axarr[2, 1].set_xlabel('Time Steps')
axarr[2, 1].set_title(r'Optimal DP Q-function $Q_t^{\star}$')


# plt.savefig('QLBS_DP_summary_graphs_ATM_option_mu=r.png', dpi=600)
# plt.savefig('QLBS_DP_summary_graphs_ATM_option_mu>r.png', dpi=600)
plt.savefig('QLBS_DP_summary_graphs_ATM_option_mu>r.png', dpi=600)

plt.show()
```




```
In [30]: # plot convergence to the Black-Scholes values

         # lam = 0.0001, Q = 4.1989 +/- 0.3612 # 4.378
         # lam = 0.001: Q = 4.9004 +/- 0.1206  # Q=6.283
         # lam = 0.005: Q = 8.0184 +/- 0.9484 # Q = 14.7489
         # lam = 0.01: Q = 11.9158 +/- 2.2846 # Q = 25.33

         lam_vals = np.array([0.0001, 0.001, 0.005, 0.01])
         # Q_vals =  np.array([3.77, 3.81, 4.57, 7.967,12.2051])
         Q_vals =  np.array([4.1989, 4.9004, 8.0184, 11.9158])
         Q_std =  np.array([0.3612,0.1206, 0.9484, 2.2846])

         BS_price = bs_put(0)
```

```
# f, axarr = plt.subplots(1, 1)
fig, ax = plt.subplots(1, 1)

f.subplots_adjust(hspace=.5)
f.set_figheight(4.0)
f.set_figwidth(4.0)

# ax.plot(lam_vals,Q_vals)
ax.errorbar(lam_vals, Q_vals, yerr=Q_std, fmt='o')

ax.set_xlabel('Risk aversion')
ax.set_ylabel('Optimal option price')
ax.set_title(r'Optimal option price vs risk aversion')
ax.axhline(y=BS_price,linewidth=2, color='r')
textstr = 'BS price = %2.2f'% (BS_price)
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
# place a text box in upper left in axes coords
ax.text(0.05, 0.95, textstr, fontsize=11,transform=ax.transAxes, verticalalignment='top
plt.savefig('Opt_price_vs_lambda_Markowitz.png')
plt.show()
```




```
In [ ]:
```