dp_qlbs_oneset_m3_ex3_v4

October 29, 2018

0.1 Fitted Q-iteration

Welcome to your 3rd assignment in Reinforcement Learning in Finance. In this exercise you will take the most popular extension of Q-Learning to a batch RL setting called Fitted Q-Iteration.

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: - Setup inputs for batch-RL model - Implement Fitted Q-Iteration

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 numpy as np
        import pandas as pd
        from scipy.stats import norm
        import random

        import sys

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

        import time
        import matplotlib.pyplot as plt
```

```
In [2]: ### ONLY FOR GRADING. DO NOT EDIT ###
        submissions=dict()
        assignment_key="0jn7tioiEeiBAA49aGvLAg"
        all_parts=["wrZFS","yqg6m","KY5p8","BsRWi","pWxky"]
        ### ONLY FOR GRADING. DO NOT EDIT ###
```

```
In [3]: COURSERA_TOKEN = 'mxzwbbOi9yVinyJa' # the key provided to the Student under his/her em
        COURSERA_EMAIL = 'cilsya@yahoo.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 = 6          # number of time steps

        N_MC = 10000 # 10000 # 50000    # 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

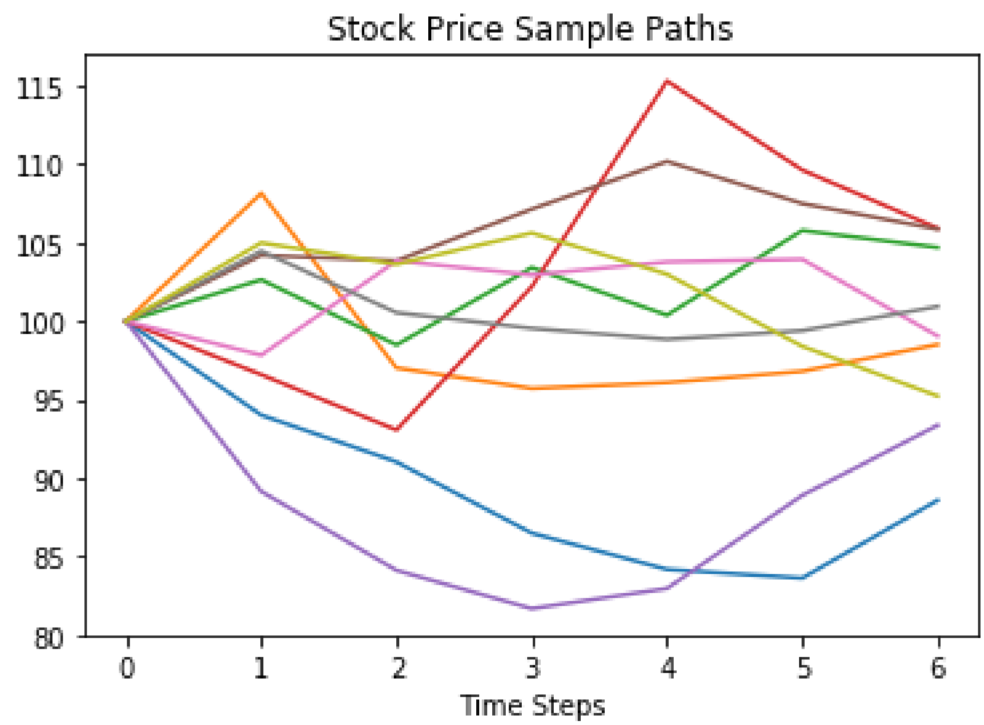
$$\Delta S_t = S_{t+1} - e^{r\Delta t} S_t \qquad \Delta \hat{S}_t = \Delta S_t - \Delta \bar{S}_t \qquad t = 0, ..., T-1$$

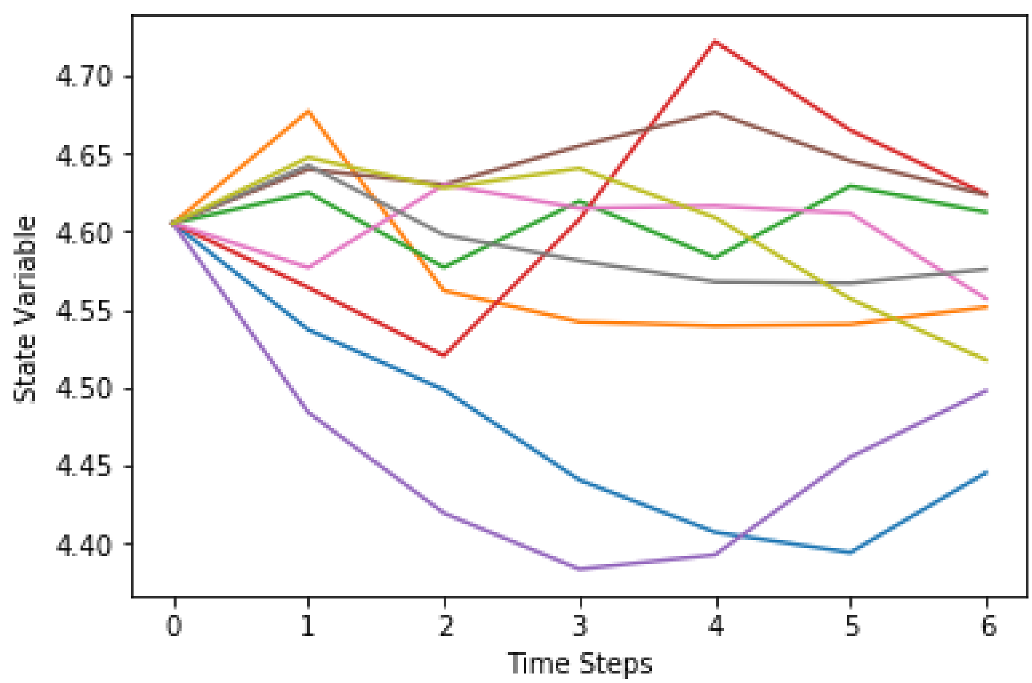
where $\Delta \bar{S}_t$ is the sample mean of all values of ΔS_t.
Plots of 5 stock price S_t and state variable X_t paths are shown below.

```
In [5]: # make a dataset

        starttime = time.time()
        np.random.seed(42) # Fix random seed
        # 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+

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.sq

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

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

# 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()
```

```
Time Cost: 0.05500006675720215 seconds
```





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

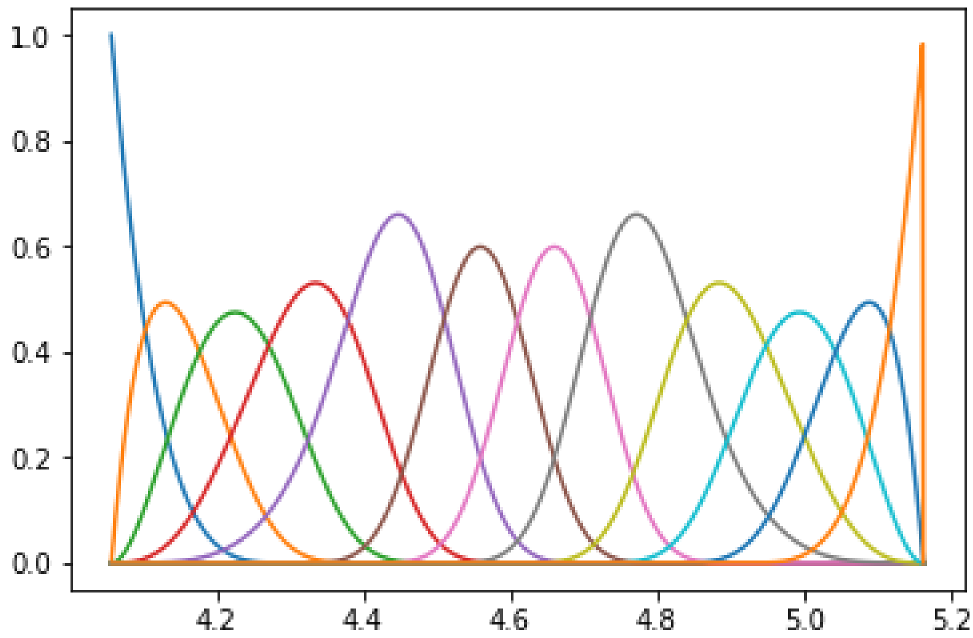
$$H_T\left(S_T\right) = \max\left(K - S_T, 0\right)$$

```
In [6]: def terminal_payoff(ST, K):
            # ST   final stock price
            # K    strike
            payoff = max(K-ST, 0)
            return payoff
```

0.4 Define spline basis functions

```
In [7]: 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 t

        # 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, 7)
X_min, X_max =  4.057527970756566 5.162066529170717
Number of points k =  17
```




```
<Figure size 432x288 with 0 Axes>
```

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

```
Out[8]: bspline.bspline.Bspline
```

```
In [9]: X.values.shape
```

```
Out[9]: (10000, 7)
```

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 [10]: 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)

         # fill it, expand function in finite dimensional space
         # in neural network the basis is the neural network itself
```

```
t_0 = time.time()
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 =  (7, 10000, 12)
Computational time: 14.045999765396118 seconds
```

In [11]:
```
# save these data matrices for future re-use
np.save('data_mat_m=r_A_%d' % N_MC, data_mat_t)
```

In [12]:
```
print(data_mat_t.shape)  # shape num_steps x N_MC x num_basis
print(len(k))
```

```
(7, 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

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

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 [13]: risk_lambda = 0.001 # 0.001 # 0.0001            # risk aversion
         K = 100 #

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

0.7 Part 1: Implement functions to compute optimal hedges

Instructions: Copy-paste implementations from the previous assignment, i.e. QLBS as these are the same functions

```
In [14]: # 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)
             # A_mat = your code goes here ...
             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-S

             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
    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:
    B_vec - np.array() of dimension num_basis x 1
    """
    # coef = 1.0/(2 * gamma * risk_lambda)
    # override it by zero to have pure risk hedge
    coef = 0. # keep it

    ### START CODE HERE ### ( 3-4 lines of code)
    # B_vec = your code goes here ...
    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
```

0.8 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.

```
In [15]: 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
```

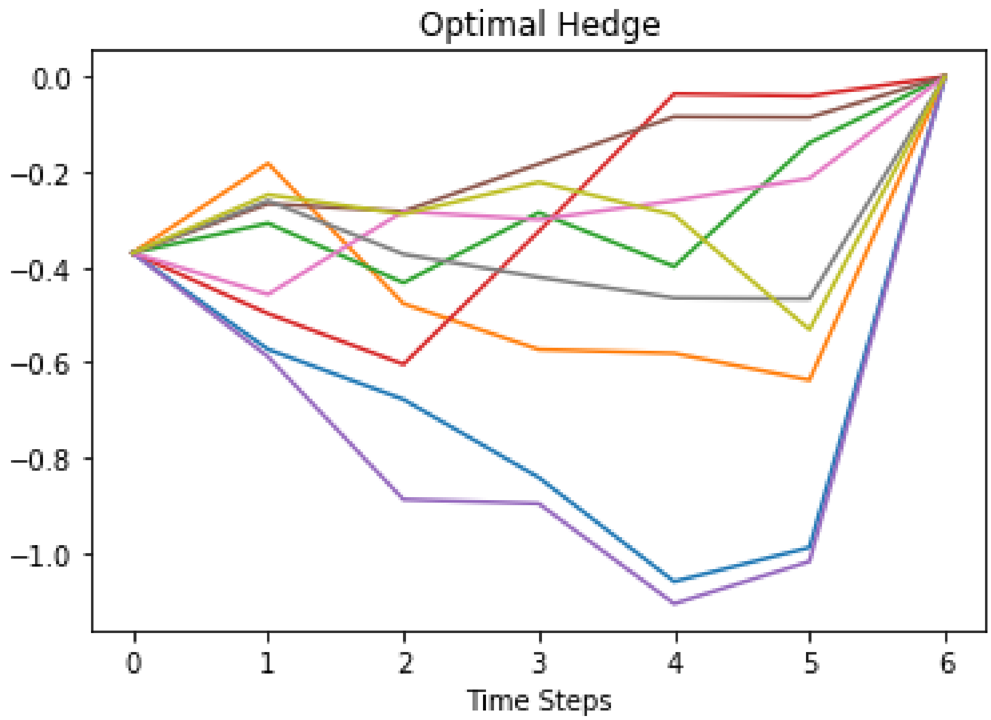
```
    reg_param = 1e-3
    for t in range(T-1, -1, -1):
        A_mat = function_A_vec(t, delta_S_hat, data_mat_t, reg_param)
        B_vec = function_B_vec(t, Pi_hat, delta_S_hat, S, data_mat_t)

        # print ('t =  A_mat.shape = B_vec.shape = ', t, A_mat.shape, B_vec.shape)
        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')
```

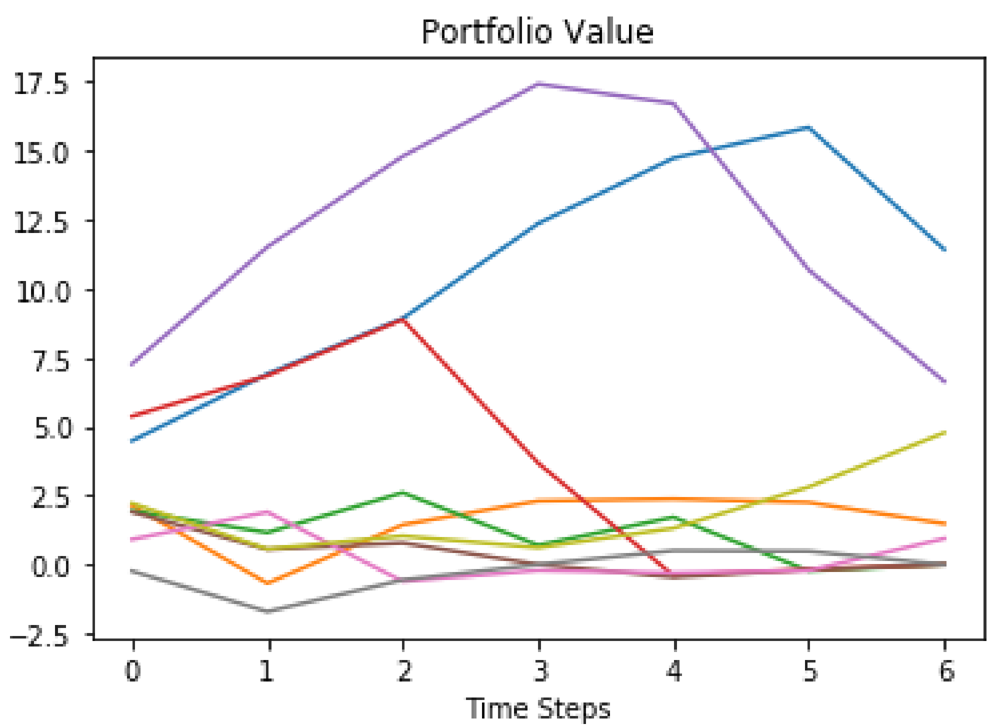
```
Computational time: 0.08799982070922852 seconds
```

```
D:\application\Anaconda3\envs\pyalgo\lib\site-packages\ipykernel_launcher.py:21: FutureWarning
```

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

```
In [16]: # 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()
```


Optimal Hedge
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
1
2
3
4
5
6
Time Steps


Portfolio Value
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
0
1
2
3
4
5
6
Time Steps

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.

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: Copy-paste implementations from the previous assignment,i.e. QLBS as these are the same functions

```
In [17]: 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
             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)
             # C_mat = your code goes 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)
             ### 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 
    gamma - one time-step discount factor $exp(-r \delta t)$

    Return:
    D_vec - np.array of dimension num_basis x 1
    """
    ### START CODE HERE ### ( 2-3 lines of code)
    # D_vec = your code goes here ...
    X_mat = data_mat[t, :, :]
    D_vec = np.dot(X_mat.T, R.loc[:,t] + gamma * Q.loc[:, t+1])
    ### END CODE HERE ###

    return D_vec
```

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 $Q_T^\star(X_T, a_T = 0) = -\Pi_T(X_T) - \lambda Var[\Pi_T(X_T)]$.

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 [18]: # 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
```

0.10 Hedging and Pricing with Reinforcement Learning

Implement a batch-mode off-policy model-free Q-Learning by Fitted Q-Iteration. The only data available is given by a set of N_{MC} paths for the underlying state variable X_t, hedge position a_t, instantaneous reward R_t and the next-time value X_{t+1}.

$$\mathcal{F}_t^k = \left\{ \left(X_t^k, a_t^k, R_t^k, X_{t+1}^k \right) \right\}_{t=0}^{T-1} \quad k = 1, ..., N_{MC}$$

Detailed steps of solving the Bellman optimalty equation by Reinforcement Learning are illustrated below.