# The QLBS model for a European option

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.
- 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.
- When encountering `# dummy code - remove` please replace this code with your own
- In case you get an importerror on bspline, invoke pip install bspline

**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!

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("..")
```

# Parameters for MC simulation of stock prices

In [2]:

```
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
```

## 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

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

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

Plots of 5 stock price $S_t$ and state variable $X_t$ paths are shown below.

In [3]:

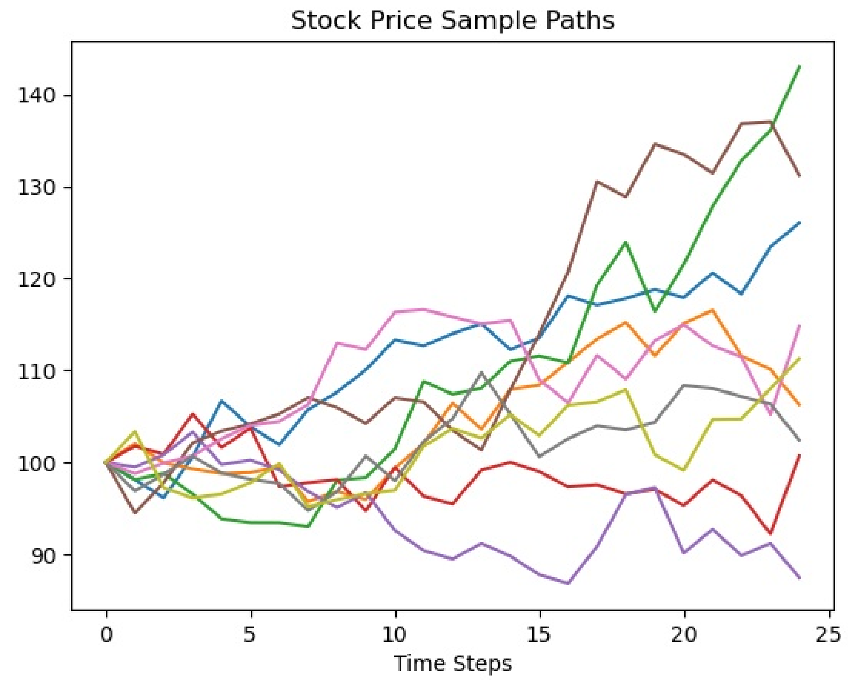
```python
# make a dataset
starttime = time.time()
np.random.seed(42)

# 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_t) * RN.loc[:,t])

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 due to their conventions

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

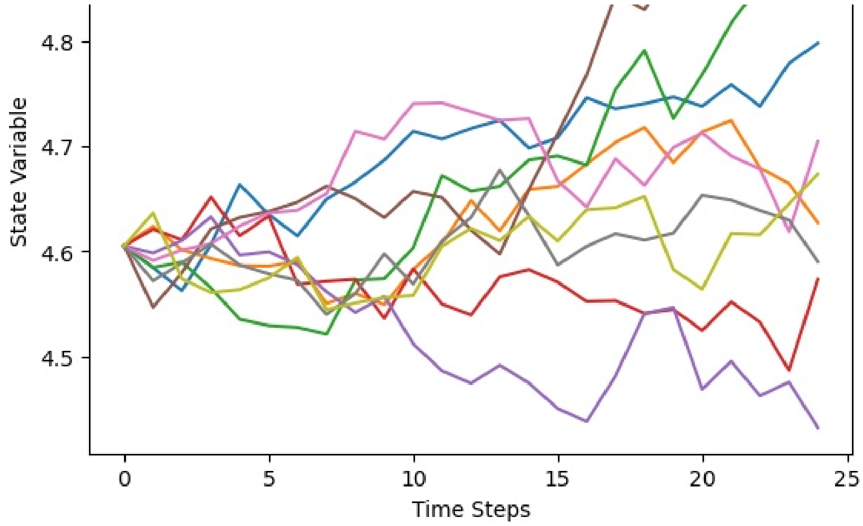
```
Time Cost: 0.05694699287414551 seconds
```

```
<ipython-input-3-c401a8694546>:7: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attempt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  S.loc[:,0] = S0
<ipython-input-3-c401a8694546>:13: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attempt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  S.loc[:,t] = S.loc[:,t-1] * np.exp((mu - 1/2 * sigma**2) * delta_t + sigma * np.sqrt(delta_t) * RN.loc[:,t])
```

In [4]:

```python
# 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.

$H_T(S_T) = \max$

In [5]:
```
def terminal_payoff(ST, K):
    # ST   final stock price
    # K    strike
    payoff = max(K - ST, 0)
    return payoff
```

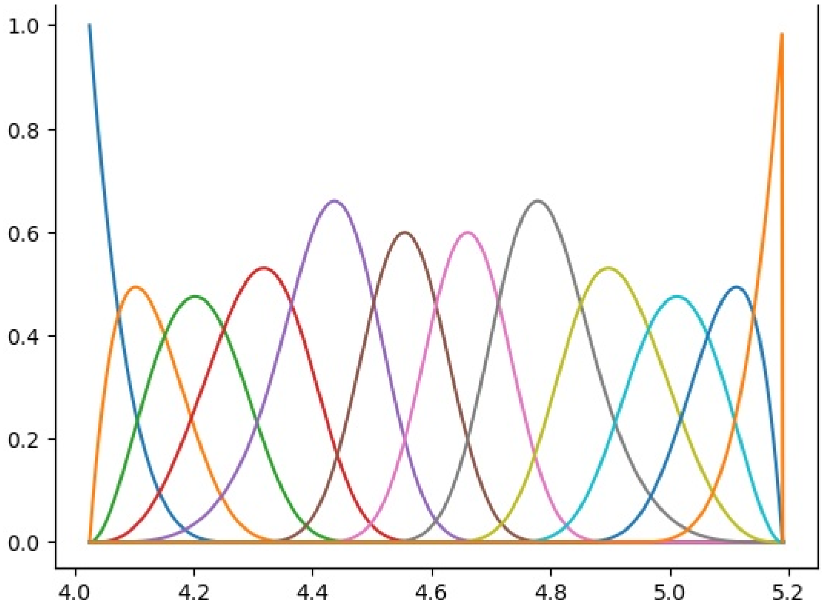
In [6]:
```
type(delta_S)
```

Out[6]:
```
pandas.core.frame.DataFrame
```

## 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 to interpolate

# 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 =  4.024923524903037 5.190802775129617
Number of points k =  17
/Users/hejifan/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py:84: FutureWarning: In a future ver
sion, DataFrame.min(axis=None) will return a scalar min over the entire DataFrame. To retain the old behavior, us
e 'frame.min(axis=0)' or just 'frame.min()'
  return reduction(axis=axis, out=out, **passkwargs)
/Users/hejifan/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py:84: FutureWarning: In a future ver
sion, DataFrame.max(axis=None) will return a scalar max over the entire DataFrame. To retain the old behavior, us
e 'frame.max(axis=0)' or just 'frame.max()'
  return reduction(axis=axis, out=out, **passkwargs)
```




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

Out[8]: bspline.bspline.Bspline

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

Out[9]: (10000, 25)

## 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)

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: 22.40867590904236 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))
```

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

## Dynamic Programming solution for QLBS

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

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

Coefficients $\phi_{nt}$ and $\omega_{nt}$ are computed recursively backward in time for $t=T−1,...,0$.

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

$A_{nm}^{\left(t\right)}=\sum_{k=1}^{N_{MC}}{\Phi_n\left(X_t^k\right)\Phi_m\left(X_t^k\right)\left(\Delta\hat{S}_t^k\right)^2}\quad\quad$
$B_n^{\left(t\right)}=\sum_{k=1}^{N_{MC}}{\Phi_n\left(X_t^k\right)\left[\hat\Pi_{t+1}^k\Delta\hat{S}_t^k+\frac{1}{2\gamma\lambda}\Delta$
$S_t^k\right]}\Delta S_t=S_{t+1} - e^{-r\Delta t} S_t\space \quad t=T-1,...,0$
where $\Delta\hat{S}_t$ is the sample mean of all values of $\Delta S_t$.

Define function *function_A* and *function_B* to compute the value of matrix $\mathbf A_t$ and vector $\mathbf B_t$.

## Define the option strike and risk aversion parameter

In [13]:
```
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 risk-based hedging
```

### Part 1 Calculate coefficients $\phi_{nt}$ of the optimal action $a_t^\star\left(X_t\right)$

**Instructions:**

- implement function_A*vec() which computes $A*{nm}^{\left(t\right)}$ matrix
- implement function_B_vec() which computes $B_n^{\left(t\right)}$ column vector

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)
    # store result in A_mat for grading
    Phi_t = data_mat[t]
    delta_S_hat_t = np.array(delta_S_hat[t]).reshape(-1, 1)
    delta_S_hat_2 = np.diag(np.dot(delta_S_hat_t, delta_S_hat_t.T))
    delta_S_hat_2_diag_mat = np.diag(delta_S_hat_2)
    A_mat = np.dot(Phi_t.T, delta_S_hat_2_diag_mat)
    A_mat = np.dot(A_mat, Phi_t) + reg_param * np.eye(data_mat.shape[2])
    ### 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-Scholes-Merton article

    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
    coef = 0. # keep it

    ### START CODE HERE ### (≈ 5-6 lines of code)
    # store result in B_vec for grading
    delta_S = np.array(S.loc[:,t+1].values - np.exp(r * delta_t) * S.loc[:,t]).reshape(-1, 1)
    sum_term_inside_bracket = np.diag(np.dot(np.array(Pi_hat[t+1]).reshape(-1, 1),
                                     np.array(delta_S_hat[t]).reshape(-1, 1).T)).reshape(-1, 1) + coef*delta_S
    B_vec = np.dot(data_mat[t].T, sum_term_inside_bracket)
    ### END CODE HERE ###
    return B_vec
```

## Compute optimal hedge and portfolio value

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

Once the optimal hedge a_t^\star\left(X_t\right) is computed, the portfolio value \Pi_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\left(S_T\right)=\max\left(K-S_T,0\right) 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 \Pi_t.

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

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 # 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')
```

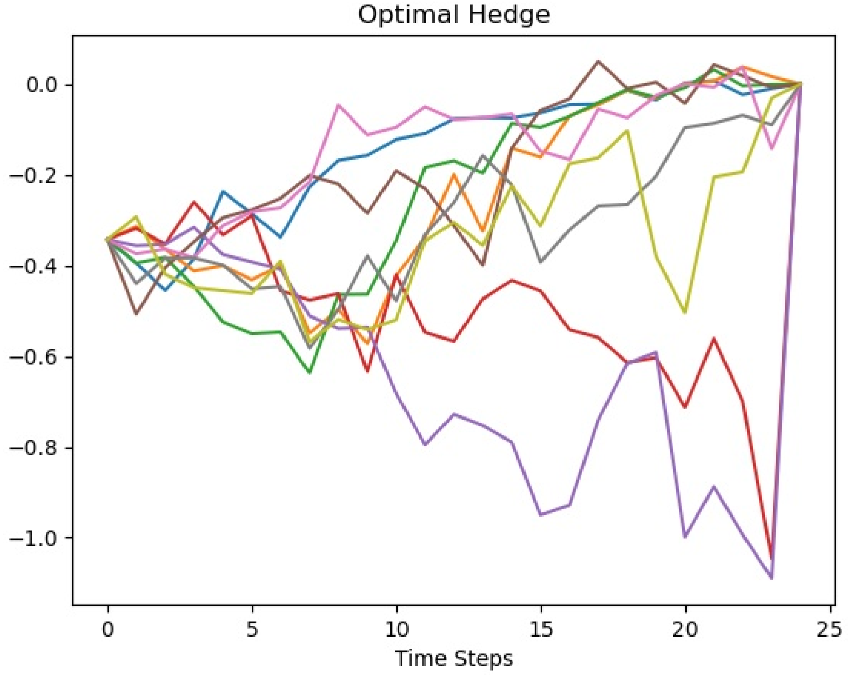
```
<ipython-input-15-04ad9718dece>:5: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attemp
t to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.
```

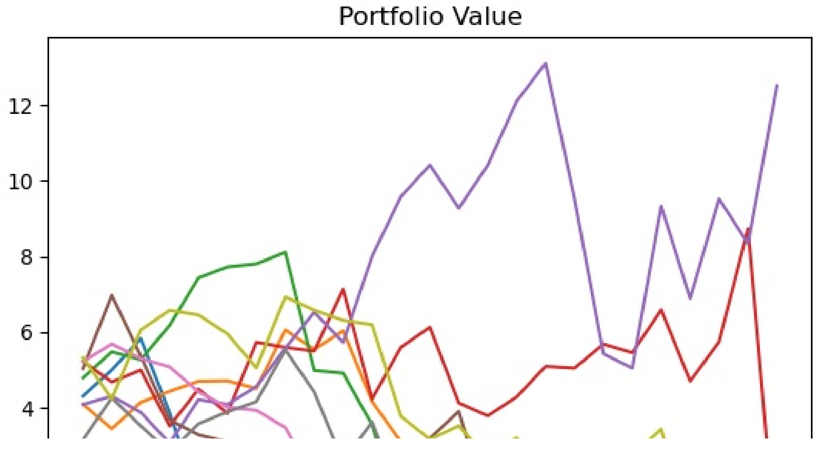
```
columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  Pi.iloc[:,-1] = S.iloc[:,-1].apply(lambda x: terminal_payoff(x, K))
<ipython-input-15-04ad9718dece>:8: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attemp
t to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.
columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  Pi_hat.iloc[:,-1] = Pi.iloc[:,-1] - np.mean(Pi.iloc[:,-1])
<ipython-input-15-04ad9718dece>:12: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attem
pt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df
.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  a.iloc[:,-1] = 0
<ipython-input-15-04ad9718dece>:23: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attem
pt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df
.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  a.loc[:,t] = np.dot(data_mat_t[t,:,:],phi)
<ipython-input-15-04ad9718dece>:24: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attem
pt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df
.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  Pi.loc[:,t] = gamma * (Pi.loc[:,t+1] - a.loc[:,t] * delta_S.loc[:,t])
<ipython-input-15-04ad9718dece>:25: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attem
pt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df
.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  Pi_hat.loc[:,t] = Pi.loc[:,t] - np.mean(Pi.loc[:,t])
```

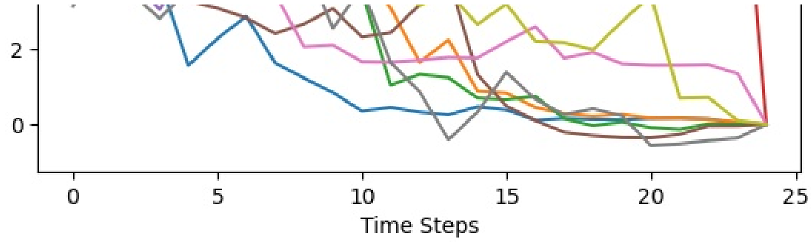
```
Computational time: 20.29300022125244 seconds
```

In [16]:

```python
# 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()
```






## Compute rewards for all paths

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

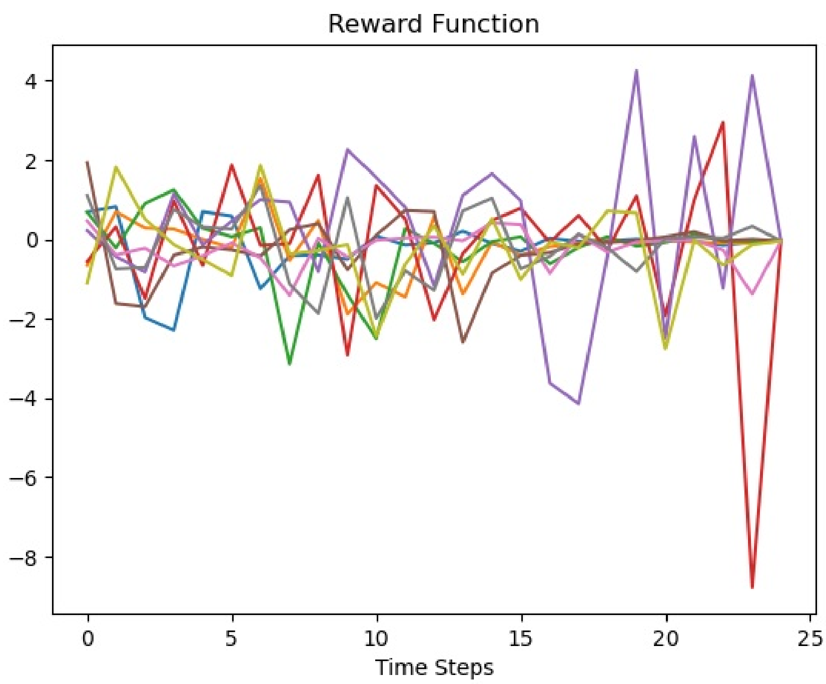
R_t\left(X_t,a_t,X_{t+1}\right)=\gamma a_t\Delta S_t-\lambda Var\left[\Pi_t\space|\space\mathcal F_t\right]\quad t=0,...,T-1
with terminal condition R_T=-\lambda Var\left[\Pi_T\right].

Plot of 5 reward function R_t paths is shown below.

In [17]:

```
# 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[1:,t])

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: 0.03611898422241211 seconds
<ipython-input-17-6633e0b1b21a>:5: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attemp
t to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.
columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  R.iloc[:,-1] = - risk_lambda * np.var(Pi.iloc[:,-1])
<ipython-input-17-6633e0b1b21a>:8: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attemp
t to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.
columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  R.loc[1:,t] = gamma * a.loc[1:,t] * delta_S.loc[1:,t] - risk_lambda * np.var(Pi.loc[1:,t])
```



## 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

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

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 $C*{nm}^{\left(t\right)}$ matrix
- implement function_D_vec() which computes $D_n^{\left(t\right)}$ column vector

In [18]:

```
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 num_basis
    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 ....
    # C_mat = your code here ...
    Phi_t = data_mat[t]
    C_mat = np.dot(Phi_t.T, Phi_t)
    C_mat += reg_param * np.eye(data_mat.shape[2])
    ### 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 num_basis
    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 ....
    # D_vec = your code here ...
    Phi_t = data_mat[t]
    R_t = R[t]
    D_vec = np.dot(Phi_t.T, R_t + gamma * Q[t+1]).reshape(-1, 1)
    ### END CODE HERE ###
    return D_vec
```

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

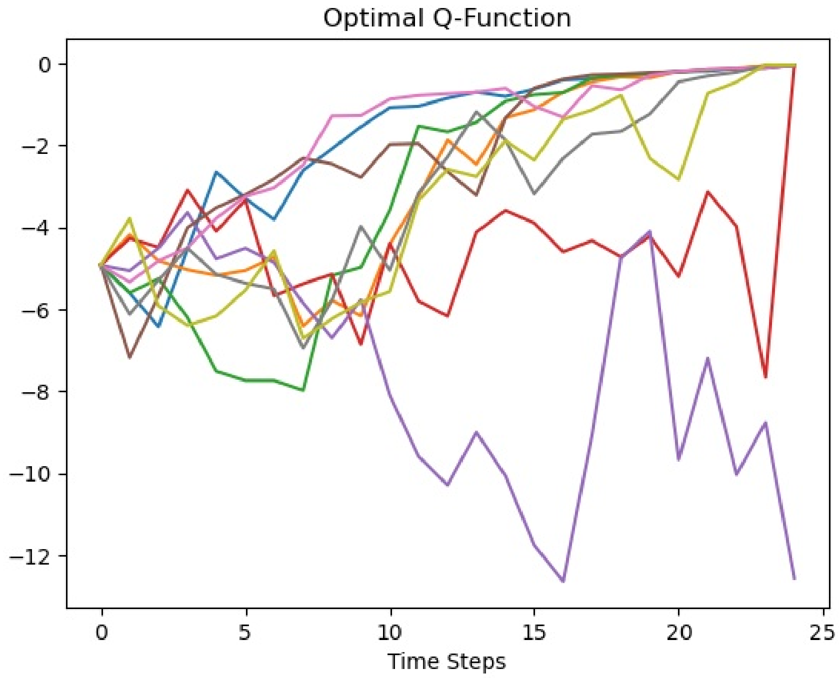
In [19]:

```
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()
```

```
<ipython-input-19-f144d51850a9>:5: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attemp
t to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df.
columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  Q.iloc[:,-1] = - Pi.iloc[:,-1] - risk_lambda * np.var(Pi.iloc[:,-1])
<ipython-input-19-f144d51850a9>:14: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attem
pt to set the values inplace instead of always setting a new array. To retain the old behavior, use either `df[df
.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  Q.loc[:,t] = np.dot(data_mat_t[t,:,:], omega)
```

```
Time Cost: 0.13919520378112793 seconds
```




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

## 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^{\left(BS\right)}=Ke^{-r\left(T-t\right)}\mathcal N\left(-d_2\right)-S_t\mathcal N\left(-d_1\right)

In [20]:

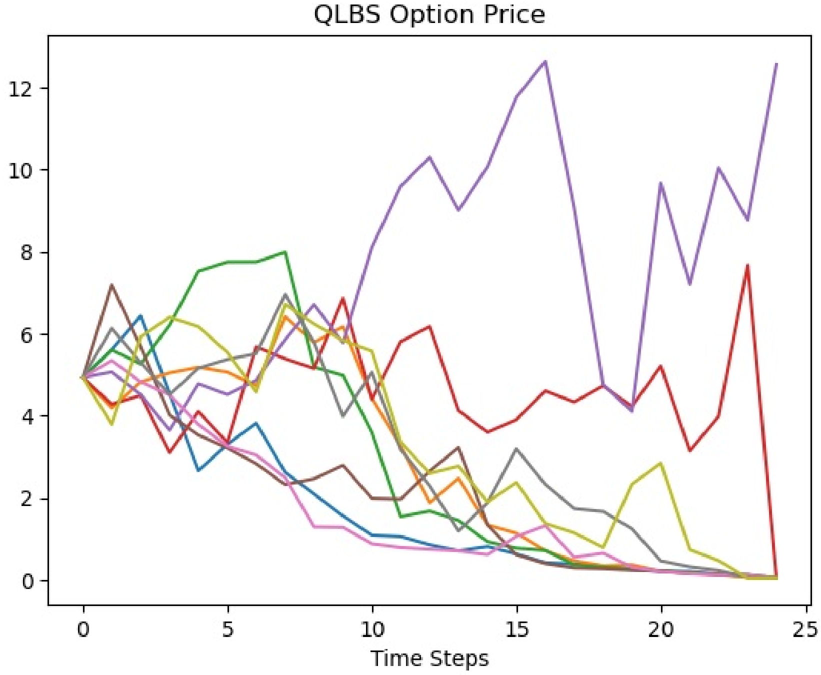
```
# The Black-Scholes prices
def bs_put(t, S0=S0, K=K, r=r, sigma=sigma, T=M):
    d1 = (np.log(S0/K) + (r + 1/2 * sigma**2) * (T-t)) / sigma / np.sqrt(T-t)
    d2 = (np.log(S0/K) + (r - 1/2 * sigma**2) * (T-t)) / sigma / np.sqrt(T-t)
    price = K * np.exp(-r * (T-t)) * norm.cdf(-d2) - S0 * norm.cdf(-d1)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

## The DP solution for QLBS

In [21]:

```
# 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
```

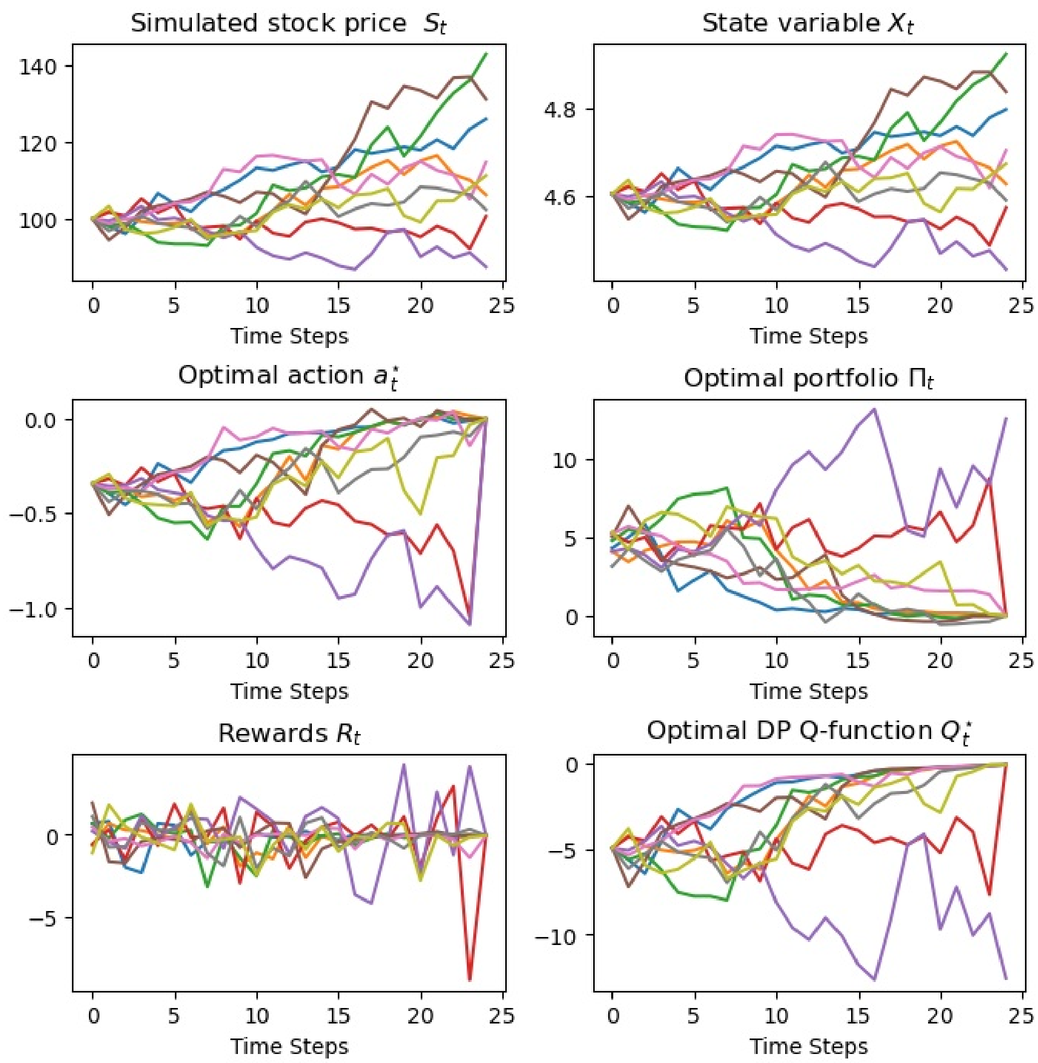



make a summary picture

In [22]:

```
# 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])
```

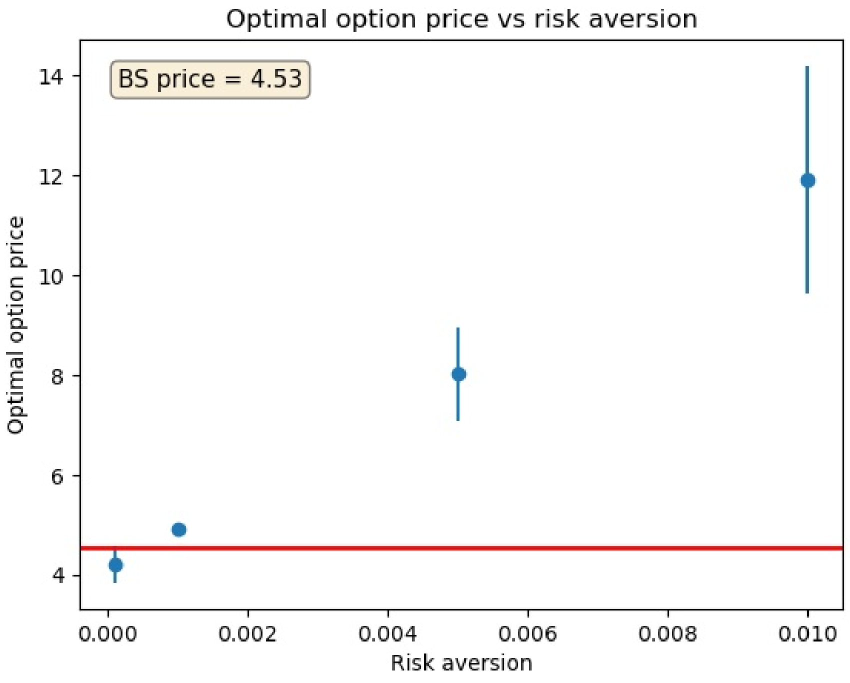
```python
axarr[0, 0].set_xlabel('Time Steps')
axarr[0, 0].set_title(r'Simulated stock price  $S_t$')

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 [23]:

```python
# 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', bbox=props)
plt.savefig('Opt_price_vs_lambda_Markowitz.png')
plt.show()
```




In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js