1.Solution: Since $-\log x_i$ is a convex function with respect to variable x_i, the linear combination of these convex functions with nonnegative coefficients leads to a new convex function, $-b^T \log(x)$. It is easily seen that $\frac{1}{2} x^T \Sigma x$, so the whole objective function as well, is a convex function.

The implemented code by me is as follows:

```
import cvxpy as cp
import numpy as np

def definite_problem():
    sigma = [[1.0, 0.0015,-0.02], [0.0015, 1.0, -0.1], [-0.02, -0.1, 1.0]]
    b = [0.1594, 0.0126, 0.8280]
    n =3
    one = [1, 1, 1]

    x = cp.Variable(n)
    objective = cp.Minimize(0.5 * cp.quad_form(x, sigma) - (b)*cp.log(x))
    constraints = [x >= 0]
    prob = cp.Problem(objective, constraints)

    result = prob.solve()
    print("The optimal value is", prob.value)
    print(x.value)

    w =x.value * (1.0/ np.inner(x.value, one))
    return w

if __name__=="__main__":
    w = definite_problem()
    print(w)
```

And the printed solution from the above code is
[0.2726693, 0.11152186, 0.61580883]

2.Solution: The objective function of this problem is so similar to the above one that it is easy to see that it is a convex function. Along with the affine equality and convex inequality constraints, it follows that this problem is a convex one.

The code to solve this problem is as follows:

```python
# -*- coding: utf-8 -*-
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt

def definition(lambda1):
    sigma = [[1.0, 0.0015,-0.02], [0.0015, 1.0, -0.1], [-0.02, -0.1, 1.0]]
    u = np.array([0.001, 0.05, 0.005])
    one = [1, 1, 1]

    x = cp.Variable(3)
    objective = cp.Minimize(cp.quad_form(x, sigma) - lambda1 * u.T @ x)
    constraints = [x >=0, one * x == 1]
    prob = cp.Problem(objective, constraints)
    result = prob.solve()

    print("The optimal value is", prob.value)
    print(x.value)

    expect_return = np.inner(x.value, u)
    volatility = np.sqrt(np.dot(np.dot(x.value, sigma), x.value))
    return expect_return, volatility

if __name__=="__main__":
    lambdas1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    lambdas = [x * (10e-4) for x in lambdas1 ]
    returns = []
    volatilities = []
    for i in range(len(lambdas)):
        lambdai = lambdas[i]
        expect_return, volatility = definition(lambdai)
        returns.append(expect_return)
        volatilities.append(volatility)

    fig = plt.figure(figsize = (12, 8))
    plt.plot(returns, volatilities)
```

And the wanted figure is shown in figure 1:

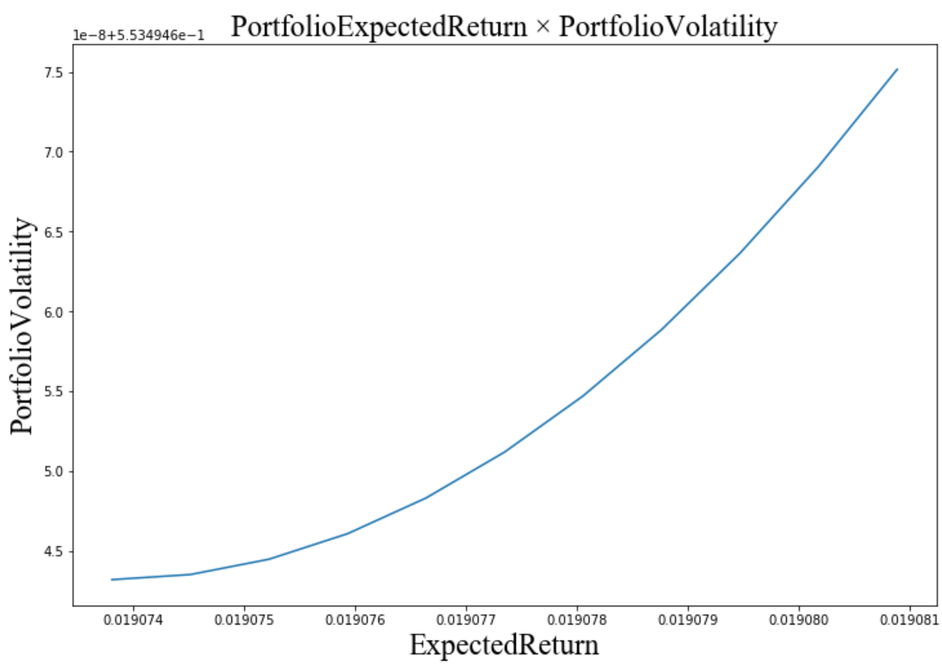

Figure 1. the curve of volatility with respect to expected return

3.Solution:

The piece of code to solve the problem is as follows:

```python
# -*- coding: utf-8 -*-

import cvxpy as cp
import numpy as np

def definition(m,n,X, Y, A):
    w = cp.Variable(n)
    objective = cp.Minimize(cp.sum_squares(Y - X @w))
    constraints = [A @w <= 0]


    prob = cp.Problem(objective, constraints)
    result = prob.solve()

    print("The optimal value is", prob.value)
    print(w.value)
    return w.value

if __name__=="__main__":
    m = 2
    n = 3

    A = np.zeros((n-1, n))
    for i in range(n-1):
        A[i][i] = 1
        A[i][i+1] = -1
    X = np.random.randn(m, n)
    Y = np.random.randn(m)
    x = definition(2, 3, X, Y, A)
```

By taking m = 2 and n = 3, we obtain that the optimal value β^* derived from above code is [-0.31162825, -0.24432935, 0.10136403].

4.Solution: The code to solve the problem is as follows:

```python
# -*- coding: utf-8 -*-
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt
import random

def definition(p, alpha, S):
    theta = cp.Variable((p,p), symmetric=True)
    objective = cp.Minimize(cp.trace(S@theta) - cp.log_det(theta * S) + alpha * cp.mixed_norm(theta, 1, 1))
    constraints = []

    prob = cp.Problem(objective, constraints)
    result = prob.solve()

    print("The optimal value is", prob.value)
    return theta.value

if __name__=="__main__":
    alphas = [0, 0.1, 0.2, 0.3, 0.4]
    thetas = []
    theta_norms = []

    p = 2
    n = 3
    X = np.zeros((n, p))
    for i in range(n):
        for j in range(p):
            X[i][j] = random.gauss(0, 1)
    S = (1/n)* np.dot(X.T, X)

    for iterationi in range(5):
        alpha = alphas[iterationi]
        theta = definition(p, alpha, S)
        thetas.append(thetas)

        sum_abs = 0
        for i in range(p):
            for j in range(p):
                sum_abs = sum_abs + abs(theta[i][j])
        theta_norms.append(sum_abs)

    fig = plt.figure(figsize = (12, 8))
    plt.plot(alphas, theta_norms)
    font1 = {'family' : 'Times New Roman',
             'weight' : 'normal',
             'size'   : 23,
             }
    plt.xlabel(r'$\alpha$', font1)
    plt.ylabel(r'||$\Theta^*(\alpha)||_1$', font1)
```

By taking p = 2, n = 3 and $\alpha=[0,0.1,0.2,0.3,0.4]$, we get the values of $\Theta^*(\alpha)$ as follows:

```
The optimal value is 1.401838125806458
[[ 1.20383951 -0.92584581]
 [-0.92584581  2.22287016]]
The optimal value is 1.8257564585132324
[[ 0.90698909 -0.48694111]
 [-0.48694111  1.57383016]]
The optimal value is 2.1224353806048475
[[ 0.76257541 -0.27341503]
 [-0.27341503  1.25822325]]
The optimal value is 2.3509719540464187
[[ 0.67716362 -0.14716747]
 [-0.14716747  1.07160457]]
The optimal value is 2.536887119750053
[[ 0.62074239 -0.06374426]
 [-0.06374426  0.94826257]]
```

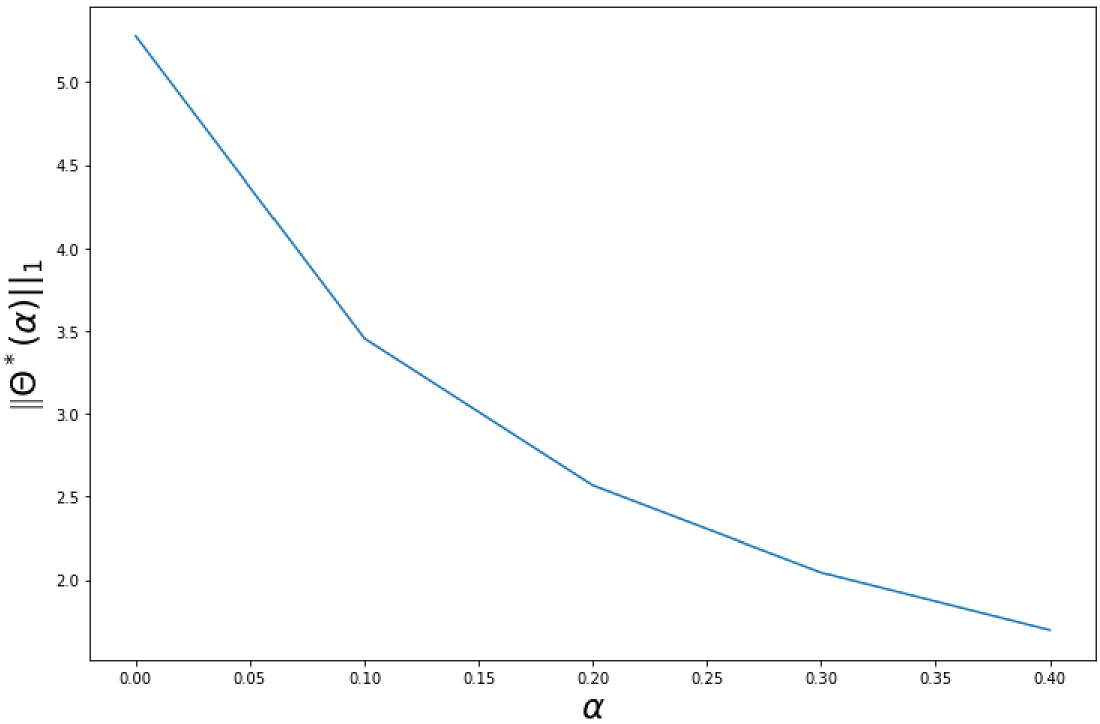

Figure 2. the value of $\|\Theta^*(\alpha)\|$ with respect to α