ADMM ALGORITHM WITH LASSO OBJECTIVE

Subtitle

Author
Chijie An
Where?
When?

Contents

1 A Small Test Case

```python
import numpy as np
#define the funtion for soft threshold
def soft_thresholding(x, lambda_val):
    return np.sign(x) * np.maximum(np.abs(x) - lambda_val, 0)

#define the function of ADMM for lasso regression
def admm_lasso(A, b, lmbda, rho, max_iter=10000):
    # Pre-compute some values
    Atb = A.T @ b
    AtA = A.T @ A
    m, n = A.shape
    I = np.eye(n)
    #initialize the variables
    x = np.zeros(n)
    z = np.zeros(n)
    u = np.zeros(n)
    value_list = []
    p_residual_list = []
    d_residual_list = []

    L = np.linalg.cholesky(AtA + rho * I)
    for k in range(max_iter):
      # x-update (using Cholesky factorization), not using calculating inverse
      q = Atb + rho * (z - u)
      x = np.linalg.solve(L.T, np.linalg.solve(L, q))
      #then do the z update
      z_old = z
      z = soft_thresholding(x+u,lmbda/rho)
```

Figure 1: Code for ADMM for Smaller Case

```python
      #then do the u update
      u+=(x-z)
      #then calculate the objective value
      value = lasso_objective(x,A,b,lmbda)
      value_list.append(value)
      #calculate primal and dual residual
      p_residual = x-z
      p_residual_list.append(p_residual)
      d_residual = rho*(-I)@(z-z_old)
      d_residual_list.append(d_residual)
      #check p_residual and d_residual as stopping criterions
      if np.linalg.norm(p_residual, 2) < 1e-15 and np.linalg.norm(d_residual, 2) < 1e-15:
        #print(k)
        print('The algorithm reaches stopping criterion after iteration',k)
        break
    return x,value,value_list,p_residual_list,d_residual_list
```

Figure 2: Code for ADMM for Smaller Case

```python
#check by a small example
np.random.seed(0)  # For reproducibility
m, n = 30, 10  # m samples, n features
A = np.random.randn(m, n)
b = np.random.randn(m)
lmbda = 1.0
rho = 1.0
```

Figure 3: The Code to Generate the Smaller Test Case

I set the stopping condition for the algorithm as the l_2 norm of both primal residual and dual residual is less than $1 \times 10^1 5$. And the algorithm reaches this stopping condition after 955 iterations. The optimal point is $x = [8.36884427e-16-3.11785748e-021.40144087e-015.46824690e-02-4.83525304e-021.76757264e-012.00082196e-01-4.64651026e-16-1.95695958e-011.72860563e-01]$ and the optimal value is 9.43045057550707.This is very close to the result from CVX package, which show the optimal point is $x = [2.35312583e-08-3.11785884e-021.40144011e-015.46824694e-02-4.83524261e-021.76757235e-012.00082135e-01-4.05366359e-07-1.95695924e-011.72860487e-01]$ and the optimal value is 9.430450674162639. Then I ran experiment on ρ by fixing $\lambda = 1$. I set $\rho =$

$0.1, 0.5, .1.0, 5.0, 10.0$ respectively, and got the following result.

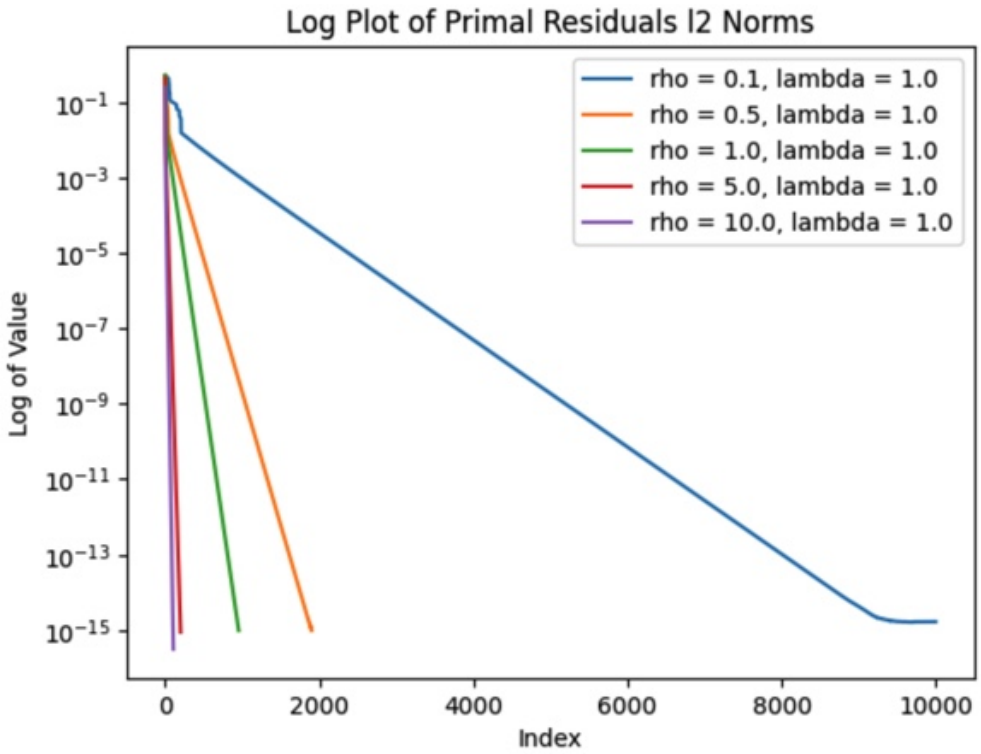

Figure 4: ρ experiment with first case

When $\rho = 0.1$, the algorithm didn't reach stopping condition after 10000 iterations. When $\rho = 0.5$, the algorithm reached stopping condition after 1902 iterations. When $\rho = 1.0$, the algorithm reached stopping condition after 955 iterations. When $\rho = 5.0$, the algorithm reached stopping condition after 201 iterations. When $\rho = 10.0$, the algorithm reached stopping condition after 107 iterations. So I observe that for the values pf ρ we take in range [0.10], the larger ρ is, the quicker ADMM algorithm converges.

Then we take the value of $\rho = 100, 500, 1000, 5000, 10000$ and run the same experiments.

When $\rho = 100$, the algorithm didn't reach stopping condition after 452 iterations. When $\rho = 500$, the algorithm reached stopping condition after 2130 iterations. When $\rho = 1000$, the algorithm reached stopping condition after 4330 iterations. When $\rho = 5000$, the algorithm reached stopping condition after more than 10000 iterations. When $\rho = 10000$, the algorithm reached stopping condition after more than 10000 iterations.So I observe that for the values pf ρ we take in range $[100, 10000]$, the larger ρ is, the slower ADMM algorithm converges.

Then I launched experiments on λ. I fixed $\rho = 1.0$ and I set the value of $\lambda = 0.1, 1, 10, 100, 1000$. I consider the numerical number less than $e - 15$ as 0. When $\lambda = 0.1$, among the 10 elements in the final result, 0 of them was 0. When $\lambda = 1$, 1 element was 0. When $\lambda = 10$, and larger, all the elements in the result were 0. Hence we see a increase of sparsity when λ gets larger.

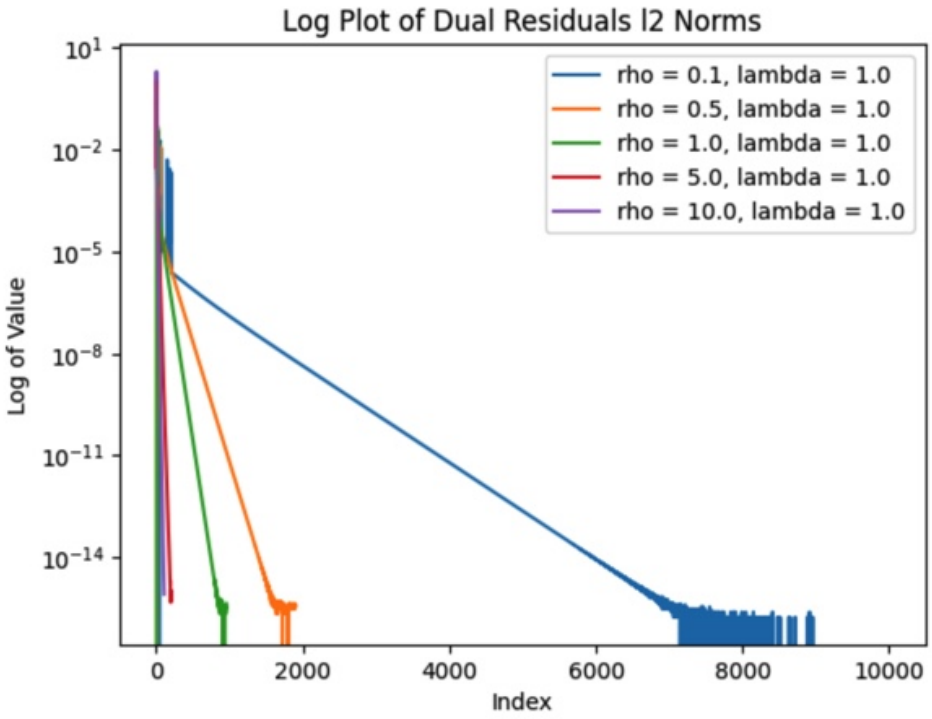

Figure 5: ρ experiment with first case

2 The Larger Test Case

Then fix $\lambda = 1$, and let $\rho = 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5$. I found that in this range $[0, 0.5]$, the larger ρ get, the quicker algorithm converges. But when I tried to take values of ρ larger than this interval, the algorithm fails to converge. It seems that there's an optimal value of ρ, and when I take ρ less than this optimal value, when ρ gets larger, the algorithm converges faster. When I take ρ larger than this value, when ρ gets larger, the algorithm converges slower. When ρ is taken too large, the algorithm fails to converge.

Then I fixed $\rho = 0.1$ and ran experiments on λ. I took $\lambda = 0.1, 0.5, 1.0, 100$. When $\lambda = 0.1$, there're 6890/10000 elements which are non zero. When $\lambda = 0.5$, there're 3744/10000 elements which are non zero. When $\lambda = 1.0$, there're 1091/10000 elements which are non zero. When $\lambda = 100$, none of the elements is non zero. This implies an increase in sparsity as λ increases.

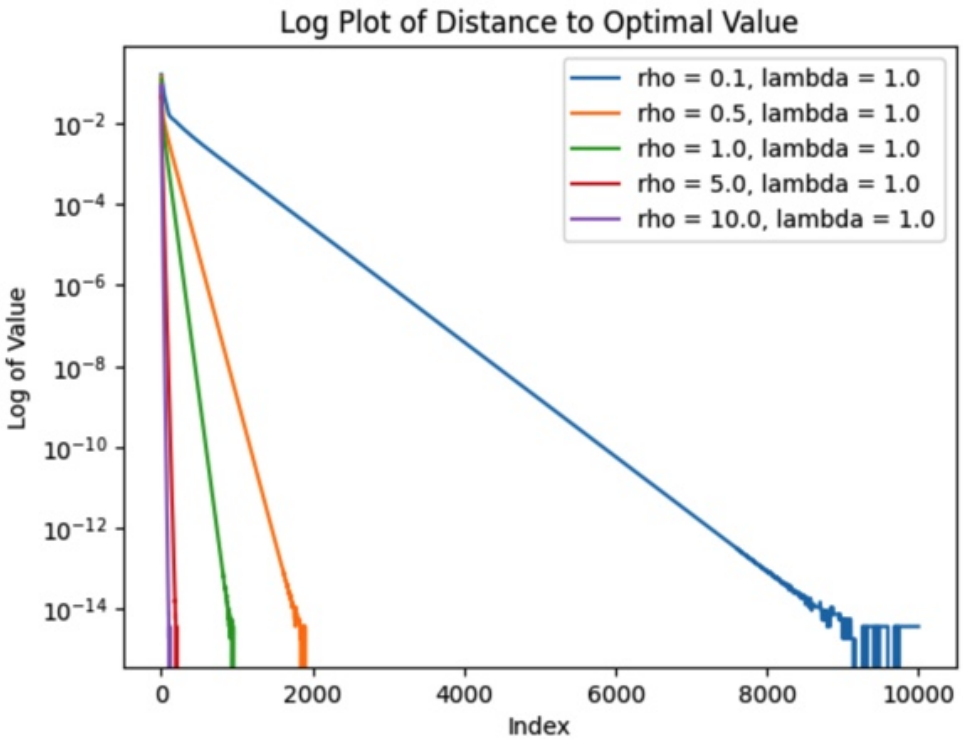

Figure 6: ρ experiment with first case

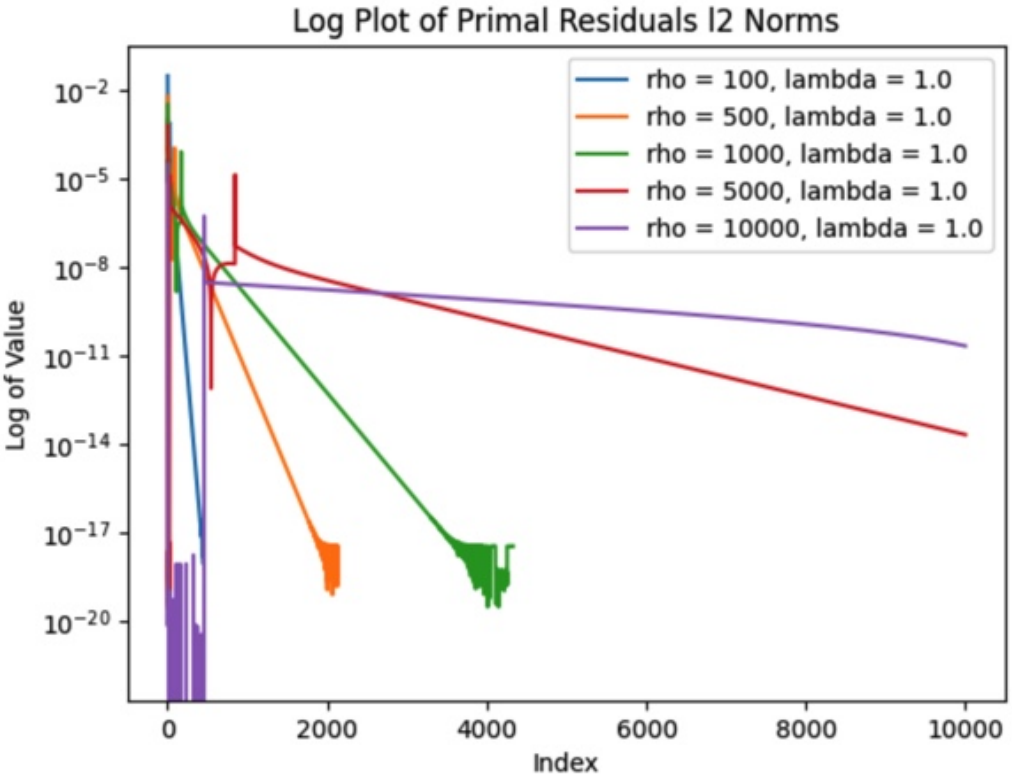

Figure 7: ρ experiment with first case

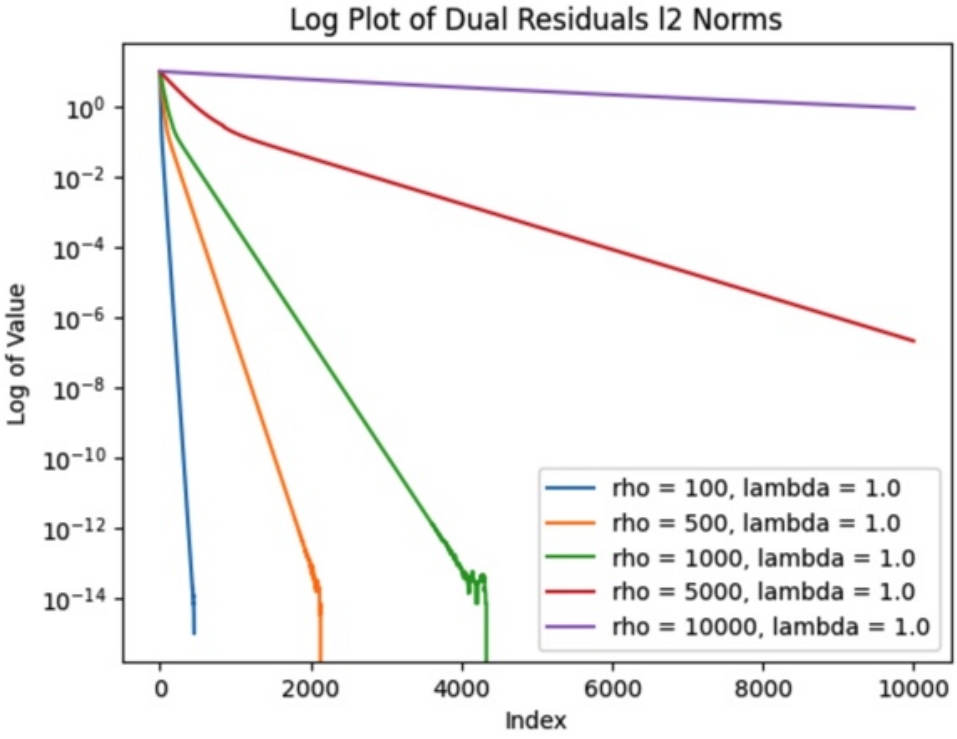

Figure 8: ρ experiment with first case

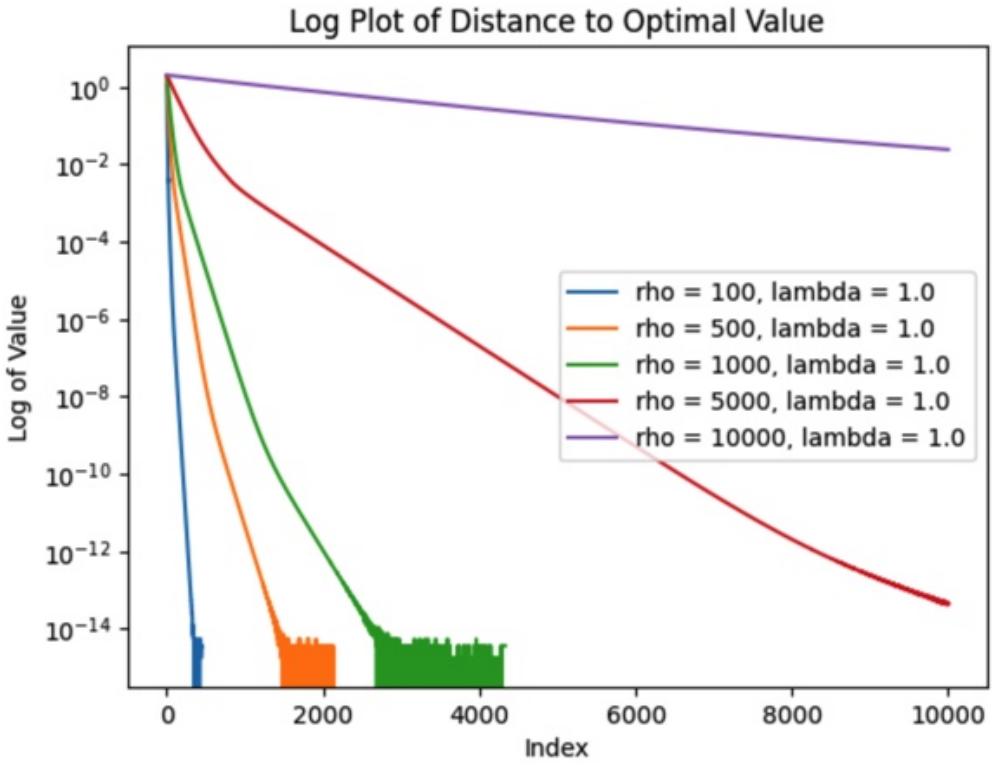

Figure 9: ρ experiment with first case

```
lambda = 0.1 [-0.00799249 -0.04070849  0.18019842  0.0820492  -0.08375469  0.19986012
  0.24312589 -0.01238344 -0.23601075  0.2168028 ]
lambda = 1 [ 8.36884427e-16 -3.11785748e-02  1.40144087e-01  5.46824690e-02
 -4.83525304e-02  1.76757264e-01  2.00082196e-01 -4.64651026e-16
 -1.95695958e-01  1.72860563e-01]
lambda = 10 [ 1.07672351e-16 -2.16173602e-16 -4.33611312e-16  2.18212330e-16
 -1.02061262e-16  4.32635283e-16  3.98812863e-16 -2.06369368e-16
  4.30232402e-16  3.69314447e-16]
lambda = 100 [ 1.07672351e-16 -2.16173602e-16 -4.33611312e-16  2.18212330e-16
 -1.02061262e-16  4.32635283e-16  3.98812863e-16 -2.06369368e-16
  4.30232402e-16  3.69314447e-16]
lambda = 1000 [ 1.07672351e-16 -2.16173602e-16 -4.33611312e-16  2.18212330e-16
 -1.02061262e-16  4.32635283e-16  3.98812863e-16 -2.06369368e-16
  4.30232402e-16  3.69314447e-16]
```

Figure 10: λ experiment with first case

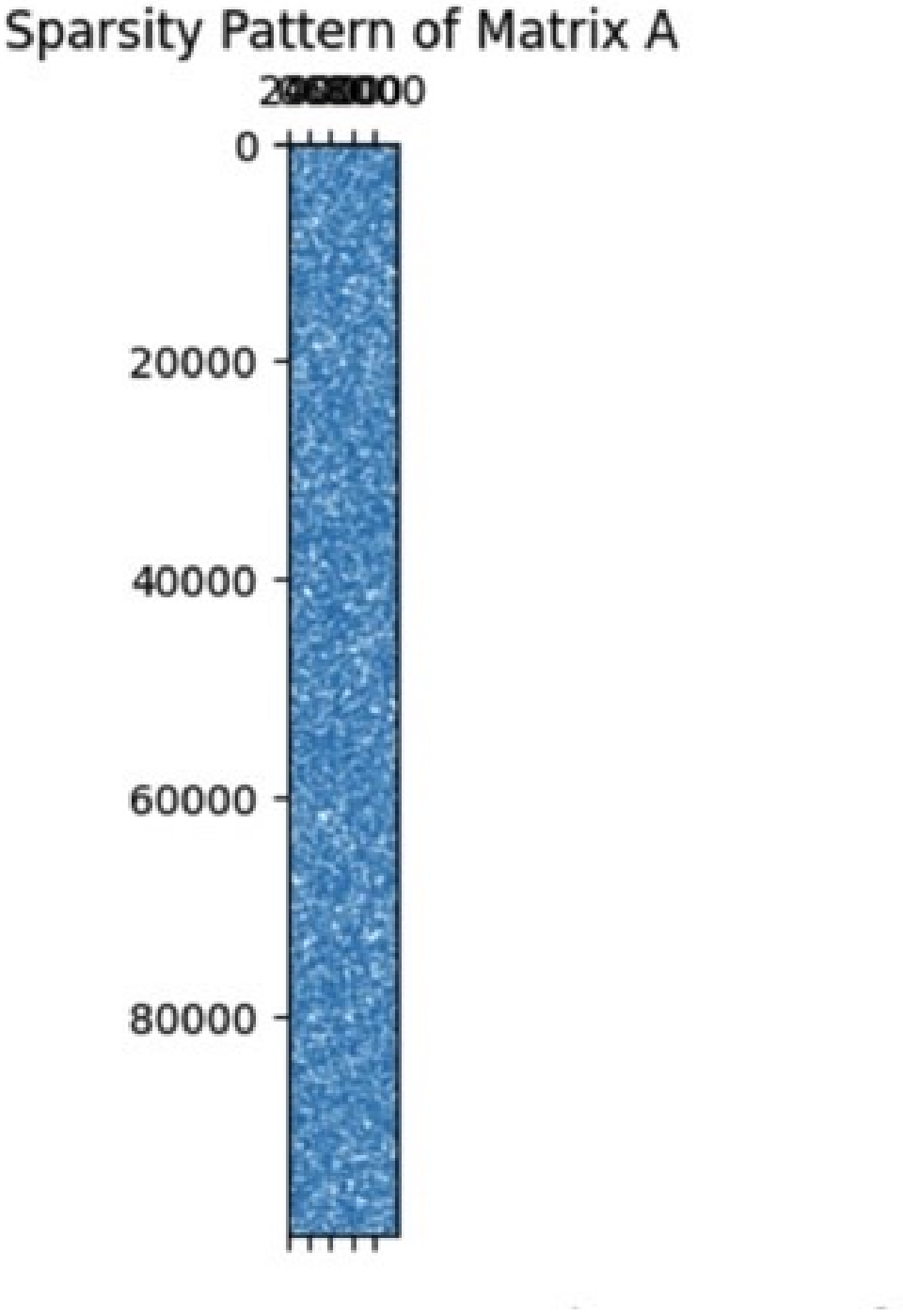

Figure 11: Sparsity Pattern of A

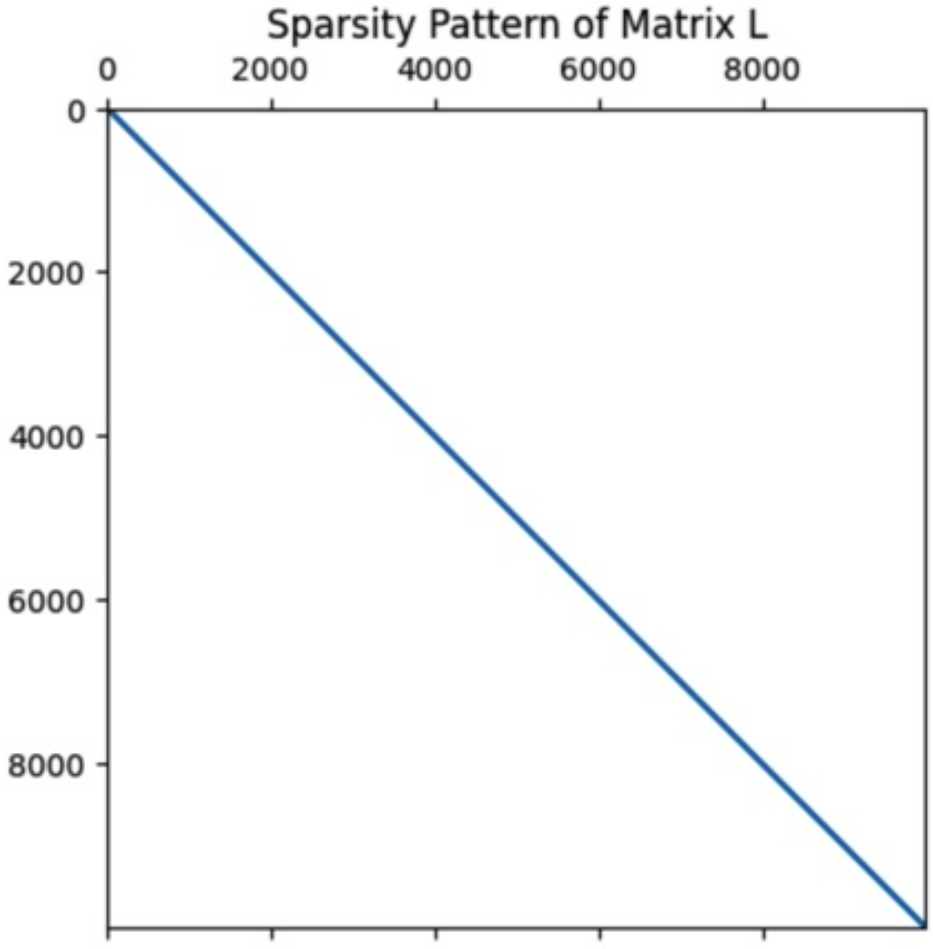

Figure 12: Sparsity Pattern of L

```
#define the sparse version of admm, with the chelosky decomposition L computed outside of the algorithm
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
from scipy.sparse.linalg import splu
import matplotlib.pyplot as plt
from scipy.sparse import eye
import numpy as np
from scipy.sparse import csr_matrix
def soft_thresholding_sparse(x, lambda_val):
    dense_x = x.toarray().ravel()
    return np.sign(dense_x) * np.maximum(np.abs(dense_x) - lambda_val, 0)
def admm_lasso_sparse(A, b, lmbda, rho, L, max_iter=1000):
    # Pre-compute some values
    Atb = A.T @ b
    dense_array = Atb.toarray().ravel()
    Atb = csr_matrix(Atb)
    #print(Atb.shape)
    #Atb = Atb.flatten()
    m, n = A.shape
    I = eye(n, format='csr')
    #I = np.eye(n)
    #initialize the variables
    x = np.zeros(n)
    x = csr_matrix(x).T
    z = np.zeros(n)
    z = csr_matrix(z).T
    u = np.zeros(n)
    u = csr_matrix(u).T
    #print('the shape of x is',x.shape)
    #print('the shape of z is',z.shape)
    #print('the shape of u is',u.shape)
    value_list = []
    p_residual_list = []
    d_residual_list = []
    value = 0
```

Figure 13: Code for Sparse Version of ADMM

```
    for k in range(max_iter):
      print(k)
      # x-update (using Cholesky factorization), not using calculating inverse
      #print('the shape of Atb is',Atb.shape)
      #print('the shape of z-u',(z-u).shape)
      q = Atb + rho * (z - u)
      #print(q.shape)
      x=csr_matrix(spsolve(L.T,spsolve(L,q))).T
      #print(x.shape)
      #print(u.shape)
      #then do the z update
      z_old = z
      z = csr_matrix(soft_thresholding_sparse(x+u,lmbda/rho).T).T
      #print(x.shape)
      #print(z.shape)
      #then do the u update
      u+=(x-z)
      p_residual = x-z
      #print(p_residual.shape)
      p_residual_list.append(p_residual)
      d_residual = rho*(-I)@(z-z_old)
      #print(d_residual.shape)
      d_residual_list.append(d_residual)
      value = lasso_objective_sparse(x,A,b,lmbda)
      #print(value)
      value_list.append(value)
      '''if np.linalg.norm(p_residual, 2) < 1e-15 and np.linalg.norm(d_residual, 2) < 1e-15:
        #print(k)
        print('The algorithm reaches stopping criterion after iteration',k)
        break'''

    return x,value,value_list,p_residual_list,d_residual_list
```

Figure 14: Code for Sparse Version of ADMM

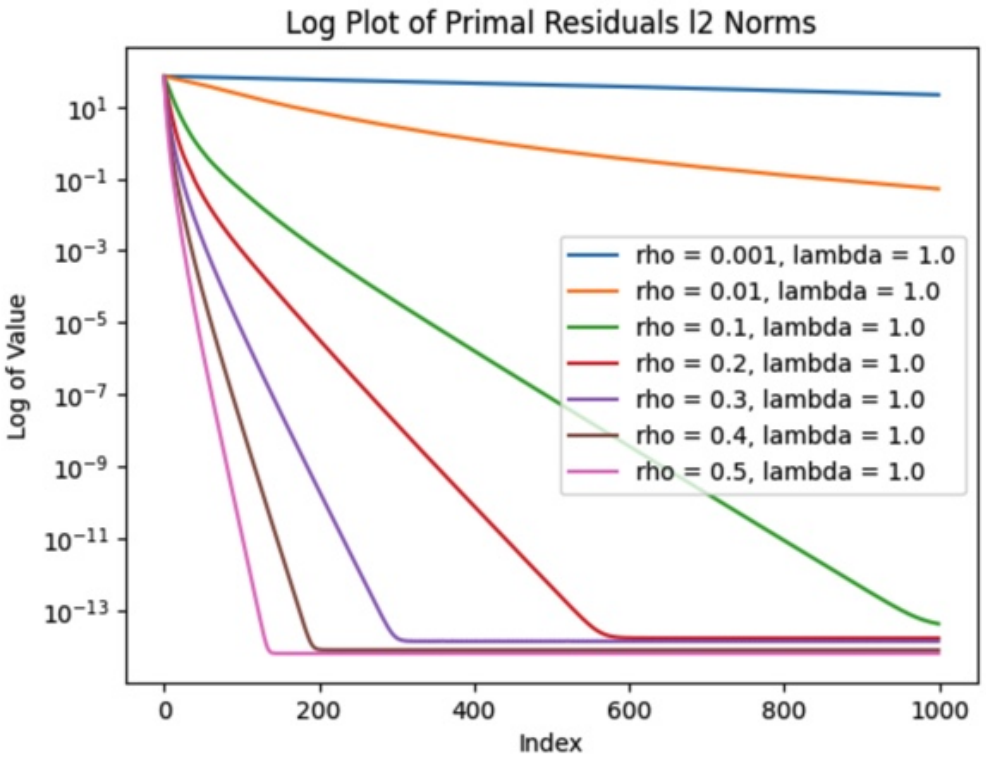

Figure 15: The Experiment of ρ for the larger case

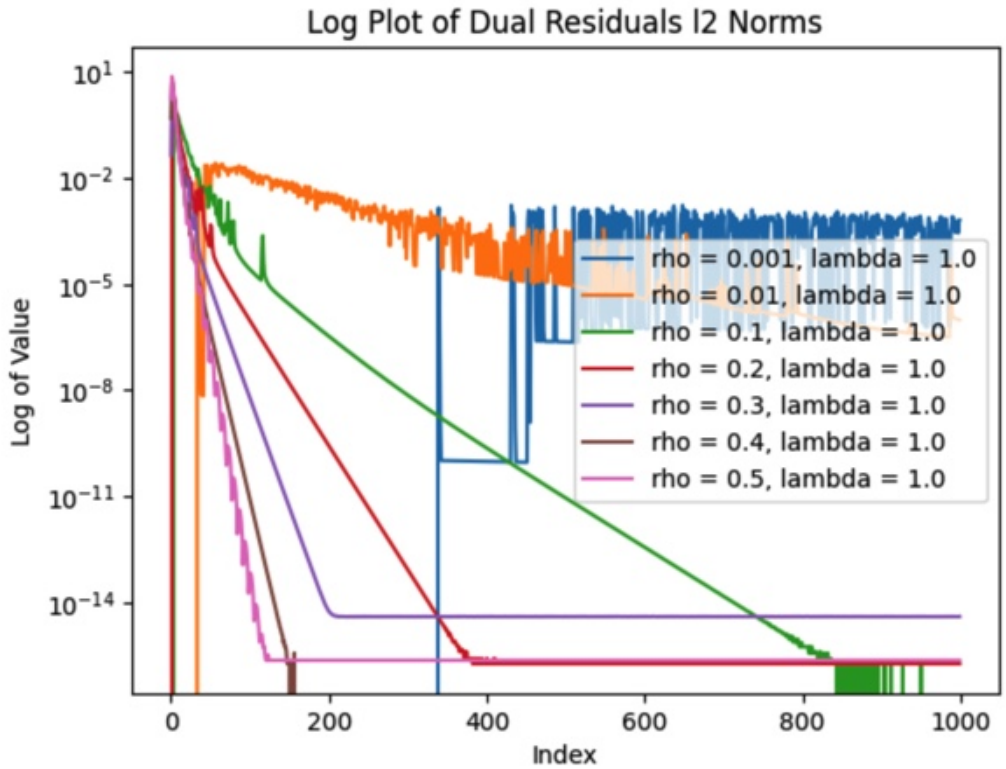

Figure 16: The Experiment of ρ for the larger case

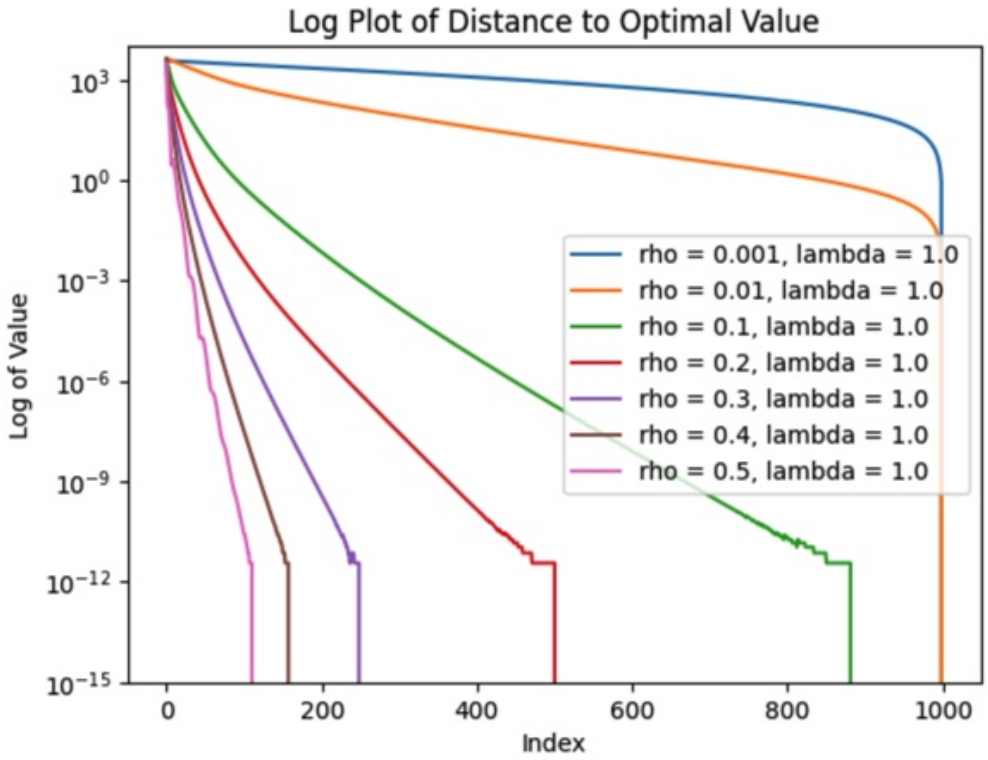

Figure 17: The Experiment of ρ for the larger case

```
#the length of x_l_1.data, which is the non strictly zero terms in x_l_1, is 8657
#however, due to the result is derived numerically, we can consider the terms which are
#very small, such as the terms less than 1e-15 as zero
nonzero_elements1 = x_l_1.data
count1 = np.sum(nonzero_elements1 < 1e-15)
print('lambda = 0.1, the number of non zero elemtents is',8657-count1)
nonzero_elements2 = x_l_2.data
count2 = np.sum(nonzero_elements2 < 1e-15)
print('lambda = 0.5, the number of non zero elements is',8657-count2)
nonzero_elements3 = x_l_3.data
count3 = np.sum(nonzero_elements3 < 1e-15)
print('lambda = 1.0, the number of non zero elements is',8657-count3)
nonzero_elements4 = x_l_4.data
count4 = np.sum(nonzero_elements4 < 1e-15)
print('lambda = 100.0, the number of non zero elements is',8657-count4)
```

```
lambda = 0.1, the number of non zero elemtents is 6890
lambda = 0.5, the number of non zero elements is 3744
lambda = 1.0, the number of non zero elements is 1091
lambda = 100.0, the number of non zero elements is 0
```

Figure 18: The Experiment of λ for the larger case