Bonus Homework

1.

Result:

1. Gaussian Elimination

[ 49. -46.66666667 14. -1.33333333]

The result of Gaussian Elimination is good, but its [arithmetic](https://en.wikipedia.org/wiki/Arithmetic#Arithmetic_operations) complexity is O(n^3).

From (2)-(5), the first result is given with initial condition x01, the second is given with initial condition x02.

1. Jacobi

x: [-3.54135383e+299 -1.95006098e+299 -5.75377760e+298 -1.10641082e+298]

iter: 1000

computed b: [-1.06281155e+300 -1.75572458e+300 -2.76286712e+300 -4.15062380e+300]

real b: [ 1. -1. 1. -1.]

x: [1.80324112e+302 9.92962100e+301 2.92979715e+301 5.63379314e+300]

iter: 1000

computed b: [5.41178763e+302 8.94006900e+302 1.40683926e+303 2.11347859e+303]

real b: [ 1. -1. 1. -1.]

The result is wrong. The Jacobi method does not converge. Because the eigenvalues of the Jacobi matrix are not smaller than 1. I set the max iteration as 1000, otherwise the result will be INF.

1. Gauss-Seidel

x: [ 48.99999999 -46.66666666 14. -1.33333333]

iter: 1679

computed b: [ 1. -1. 1. -1.]

real b: [ 1. -1. 1. -1.]

x: [ 48.99999999 -46.66666666 14. -1.33333333]

iter: 1689

computed b: [ 1. -1. 1. -1.]

real b: [ 1. -1. 1. -1.]

The result of Gauss-Seidel method is the same as that of Gaussian Elimination. The Gauss-Seidel method converge. Its arithmetic complexity is O(n^2).

1. Conjugate gradient

x: [ 49. -46.66666667 14. -1.33333333]

iter: 5

computed b: [ 1. -1. 1. -1.]

real b: [ 1. -1. 1. -1.]

x: [ 49. -46.66666667 14. -1.33333333]

iter: 5

computed b: [ 1. -1. 1. -1.]

real b: [ 1. -1. 1. -1.]

The result of Conjugate gradient method is the same as that of Gaussian Elimination. The Conjugate gradient method guarantee the convergence. Its arithmetic complexity is O(n). and it converge much faster than the Jacobi and Gauss-Seidel method.

1. 

When I set the epsilon as 1 or 0.01, the results blow up. In these cases, this method does not converge.

In order to make sure of the convergence of this method:

一些文字和图案

描述已自动生成

I set the max iteration as 100000000 and epsilon as -8e-5, the result is below which is close to x I get from (1). I can derive that the computed b is close to the real b, and this method can converge if I do not limit the number of iterations. The speed of convergence is slow.

x: [ 48.77960776 -46.45527472 13.93682297 -1.32738024]

iter: 100000000

computed b: [ 0.99730827 -0.99407619 0.99534093 -0.9987218 ]

real b: [ 1. -1. 1. -1.]

x: [ 48.78028483 -46.45592414 13.93701706 -1.32739853]

iter: 100000000

computed b: [ 0.99731654 -0.99409439 0.99535525 -0.99872572]

real b: [ 1. -1. 1. -1.]

(1)

**import** numpy **as** np  
  
A = np.array([[1, 2, 4, 8], [1, 3, 9, 27], [1, 4, 16, 64], [1, 5, 25, 125]], dtype=**'float'**)  
b = np.array([1, -1, 1, -1], dtype=**'float'**)  
  
Ab = np.hstack([A, b.reshape(-1, 1)])  
n = len(b)  
  
**for** i **in** range(n):  
 a = Ab[i]  
 **for** j **in** range(i + 1, n):  
 b = Ab[j]  
 m = a[i] / b[i]  
 Ab[j] = a - m \* b  
  
**for** i **in** range(n - 1, -1, -1):  
 Ab[i] = Ab[i] / Ab[i, i]  
 a = Ab[i]  
 **for** j **in** range(i - 1, -1, -1):  
 b = Ab[j]  
 m = b[i] / a[i]  
 Ab[j] = b - m \* a *# a - m \* b*x = Ab[:, n]  
print(x)

(2)

**import** numpy **as** np  
  
  
**def** jacobi(A, b, x\_init, epsilon=1e-10, max\_iter=1000):  
 D = np.diag(np.diag(A))  
 LU = A - D  
 x = x\_init  
 **for** i **in** range(0, max\_iter):  
 x\_new = np.dot(np.linalg.inv(D), b - np.dot(LU, x))  
 **if** np.linalg.norm(x\_new - x) < epsilon:  
 **return** x\_new, i+1  
 x = x\_new  
 **return** x, i+1  
  
A = np.array([[1, 2, 4, 8], [1, 3, 9, 27], [1, 4, 16, 64], [1, 5, 25, 125]], dtype=**'float'**)  
b = np.array([1, -1, 1, -1], dtype=**'float'**)  
x\_init = np.zeros(len(b))  
x, iter = jacobi(A, b, x\_init)  
  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)  
  
x\_init = np.ones(len(b))  
x, iter = jacobi(A, b, x\_init)  
  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)

(3)

**import** numpy **as** np  
  
**def** gauss\_seidel(A, b, x\_init, epsilon=1e-10, max\_iter=2000):  
 DL = np.tril(A)  
 U = A - DL  
 x = x\_init  
 **for** i **in** range(0, max\_iter):  
 x\_new = np.dot(np.linalg.inv(DL), b - np.dot(U, x))  
 **if** np.linalg.norm(x\_new - x) < epsilon:  
 **return** x\_new, i+1  
 x = x\_new  
 **return** x, i+1  
  
  
A = np.array([[1, 2, 4, 8], [1, 3, 9, 27], [1, 4, 16, 64], [1, 5, 25, 125]], dtype=**'float'**)  
b = np.array([1, -1, 1, -1], dtype=**'float'**)  
  
x\_init = np.zeros(len(b))  
x, iter = gauss\_seidel(A, b, x\_init)  
  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)  
  
x\_init = np.ones(len(b))  
x, iter = gauss\_seidel(A, b, x\_init)  
  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)

(4)

**import** numpy **as** np  
**from** scipy.sparse.linalg **import** cg  
  
  
**def** conjugate\_grad(A, b, x\_init):  
 r = b - np.dot(A, x\_init)  
 p = r  
 r\_norm = np.dot(r, r)  
 x = x\_init  
 **for** i **in** range(5000):  
 Ap = np.dot(A, p)  
 alpha = r\_norm / np.dot(p, Ap)  
 x = x + alpha \* p  
 r = r - alpha \* Ap  
 r\_new\_norm = np.dot(r, r)  
 beta = r\_new\_norm / r\_norm  
 r\_norm = r\_new\_norm  
 **if** r\_new\_norm < 1e-5:  
 **return** x, i + 1  
 p = r + beta \* p  
 **return** x, i + 1  
  
  
n = 1000  
  
A = np.array([[1, 2, 4, 8], [1, 3, 9, 27], [1, 4, 16, 64], [1, 5, 25, 125]], dtype=**'float'**)  
b = np.array([1, -1, 1, -1], dtype=**'float'**)  
x\_init = np.zeros(len(b))  
x, iter = conjugate\_grad(np.dot(A.T, A), np.dot(A.T, b), x\_init)  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)  
  
x\_init = np.ones(len(b))  
x, iter = conjugate\_grad(np.dot(A.T, A), np.dot(A.T, b), x\_init)  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)  
  
*# x, iter = cg(np.dot(A.T,A),np.dot(A.T,b), x\_init, maxiter=5000)  
# print('x:', x)  
# print('iter:', iter)  
# print('computed b:', np.dot(A, x))  
# print('real b:', b)*

(5)

**import** numpy **as** np  
  
  
**def** epsilon(A, b, ep, x\_init, max\_iter=100000000):  
 x = x\_init  
 **for** i **in** range(0, max\_iter):  
 x\_new = x + ep \* (np.dot(np.dot(A.T, A), x) - np.dot(A.T, b))  
 **if** np.linalg.norm(x\_new - x) < 1e-8:  
 **return** x\_new, i + 1  
 x = x\_new  
 **return** x, i + 1  
  
  
A = np.array([[1, 2, 4, 8], [1, 3, 9, 27], [1, 4, 16, 64], [1, 5, 25, 125]], dtype=**'float'**)  
b = np.array([1, -1, 1, -1], dtype=**'float'**)  
  
x\_init = np.zeros(len(b))  
x, iter = epsilon(A, b, -8e-5, x\_init)  
  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)  
  
x\_init = np.ones(len(b))  
x, iter = epsilon(A, b, -8e-5, x\_init)  
  
print(**'x:'**, x)  
print(**'iter:'**, iter)  
print(**'computed b:'**, np.dot(A, x))  
print(**'real b:'**, b)

2.

(1)

eigenvalue of B: [1. 0.28011786 0.03678039 0.00171947]

eigenvalue of B^T: [1. 0.28011786 0.03678039 0.00171947]

(2) determinant: 1.771541950113376e-05

(3)

exp(At): [[0.0509050771373172\*1.00172094779154\*\*t + 0.30548673934192\*1.03746516189031\*\*t + 0.562497129281797\*1.32328576898435\*\*t + 0.0811110542389654\*2.71828182845904\*\*t

-0.129937013626906\*1.00172094779154\*\*t - 0.19845890597641\*1.03746516189031\*\*t + 0.247284865364351\*1.32328576898435\*\*t + 0.0811110542389653\*2.71828182845904\*\*t

0.112037306248523\*1.00172094779154\*\*t - 0.192632401852302\*1.03746516189031\*\*t - 0.000515958635186719\*1.32328576898435\*\*t + 0.0811110542389653\*2.71828182845904\*\*t

-0.0326552510203222\*1.00172094779154\*\*t + 0.130548258608008\*1.03746516189031\*\*t - 0.179004061826651\*1.32328576898435\*\*t + 0.0811110542389653\*2.71828182845904\*\*t]

[-0.174437451195041\*1.00172094779154\*\*t - 0.357006523490751\*1.03746516189031\*\*t + 0.372301492899096\*1.32328576898435\*\*t + 0.159142481786696\*2.71828182845904\*\*t

0.445257776779932\*1.00172094779154\*\*t + 0.231928640277622\*1.03746516189031\*\*t + 0.16367110115575\*1.32328576898435\*\*t + 0.159142481786696\*2.71828182845904\*\*t

-0.38392048966023\*1.00172094779154\*\*t + 0.225119506807759\*1.03746516189031\*\*t - 0.000341498934224715\*1.32328576898435\*\*t + 0.159142481786696\*2.71828182845904\*\*t

0.111900405155136\*1.00172094779154\*\*t - 0.15256498548453\*1.03746516189031\*\*t - 0.118477901457302\*1.32328576898435\*\*t + 0.159142481786695\*2.71828182845904\*\*t]

[0.205137361798294\*1.00172094779154\*\*t - 0.416447101301053\*1.03746516189031\*\*t - 0.0745325085394176\*1.32328576898435\*\*t + 0.285842248042177\*2.71828182845904\*\*t

-0.523620386694836\*1.00172094779154\*\*t + 0.270544103810508\*1.03746516189031\*\*t - 0.0327659651578486\*1.32328576898435\*\*t + 0.285842248042177\*2.71828182845904\*\*t

0.451488117085304\*1.00172094779154\*\*t + 0.262601268850043\*1.03746516189031\*\*t + 6.83660224757788e-5\*1.32328576898435\*\*t + 0.285842248042177\*2.71828182845904\*\*t

-0.13159418313226\*1.00172094779154\*\*t - 0.177966624653893\*1.03746516189031\*\*t + 0.023718559743976\*1.32328576898435\*\*t + 0.285842248042176\*2.71828182845904\*\*t]

[-0.0816049877405702\*1.00172094779154\*\*t + 0.467966885449884\*1.03746516189031\*\*t - 0.860266113641477\*1.32328576898435\*\*t + 0.473904215932163\*2.71828182845904\*\*t

0.20829962354181\*1.00172094779154\*\*t - 0.304013838111721\*1.03746516189031\*\*t - 0.378190001362253\*1.32328576898435\*\*t + 0.473904215932163\*2.71828182845904\*\*t

-0.179604933673598\*1.00172094779154\*\*t - 0.295088373805501\*1.03746516189031\*\*t + 0.000789091546935657\*1.32328576898435\*\*t + 0.473904215932163\*2.71828182845904\*\*t

0.0523490289974458\*1.00172094779154\*\*t + 0.199983351530414\*1.03746516189031\*\*t + 0.273763403539977\*1.32328576898435\*\*t + 0.473904215932163\*2.71828182845904\*\*t]]

(4)

Q: [[-0.5 0.67082039 0.5 0.2236068 ]

[-0.5 0.2236068 -0.5 -0.67082039]

[-0.5 -0.2236068 -0.5 0.67082039]

[-0.5 -0.67082039 0.5 -0.2236068 ]]

R: [[-0.5 -0.5 -0.5 -0.5 ]

[ 0. -0.15971914 -0.28986066 -0.38731892]

[ 0. 0. 0.03703704 0.09375 ]

[ 0. 0. 0. -0.00598947]]

(5)

U: [[-0.21177778 -0.690499 0.63518011 -0.27369241]

[-0.33164992 -0.50768972 -0.35615142 0.71092591]

[-0.51113196 -0.12987703 -0.57938079 -0.62144507]

[-0.76413521 0.49859226 0.36608783 0.18298318]]

Singular value: [1.0925983 0.2733689 0.03517928 0.00168599]

V: [[-0.41613987 -0.75856608 0.467148 -0.1821479 ]

[-0.47617221 -0.24316017 -0.57903005 0.61551391]

[-0.52734389 0.20699005 -0.41193189 -0.71370558]

[-0.56744697 0.56798395 0.52612647 0.2803375 ]]

(6)

Cholesky factorization: A=CC^T

Cholesky factorization of BTB:

C=[[0.5 0. 0. 0. ]

[0.5 0.15971914 0. 0. ]

[0.5 0.28986066 0.03703704 0. ]

[0.5 0.38731892 0.09375 0.00598947]]

Cholesky factorization of BBT:

C=[[0.29945057 0. 0. 0. ]

[0.36654721 0.12793805 0. 0. ]

[0.45238947 0.32612569 0.05042429 0. ]

[0.56017195 0.61064612 0.17036487 0.00917037]]

Code for problem 2:

**import** numpy **as** np  
**from** scipy.sparse.linalg **import** expm  
**from** sympy **import** \*  
  
A = np.array([[1, 2, 4, 8], [1, 3, 9, 27], [1, 4, 16, 64], [1, 5, 25, 125]], dtype=**'float'**)  
d = np.array([[4, 0, 0, 0], [0, 14, 0, 0], [0, 0, 54, 0], [0, 0, 0, 224]], dtype=**'float'**)  
  
B = np.dot(A, np.linalg.inv(d))  
print(B)  
value, vector = np.linalg.eig(B)  
print(**'eigenvalue of B:'**, value, **'eigenvector of B:'**, vector)  
value2, vector2 = np.linalg.eig(B.T)  
print(**'eigenvalue of B^T:'**, value2, **'eigenvector of B^T:'**, vector2)  
det = np.linalg.det(B)  
print(**'determinant:'**, det)  
  
t = symbols(**'t'**)  
dia=np.diag(value)  
exp=np.diag(np.exp(value)\*\*t)  
final=np.dot(np.dot(vector,exp),np.linalg.inv(vector))  
print(**'exp（At):'**,final)  
  
q, r = np.linalg.qr(B)  
print(**'Q:'**, q, )  
print(**'R:'**, r)  
u, sigma, vT = np.linalg.svd(B)  
print(**'U:'**, u)  
print(**'singular value:'**, sigma)  
print(**'V:'**, vT.T)  
l1 = np.linalg.cholesky(np.dot(B.T, B))  
l2 = np.linalg.cholesky(np.dot(B, B.T))  
print(**'Cholesky factorization of BTB:'**, l1)  
print(**'Cholesky factorization of BBT:'**, l2)