In [1]:
```
# Alexander Hebert
# ECE 6390
# Computer Project #2
# Method 2/3
```

In [2]:
```
# Tested using Python v3.4 and IPython v2
```

In [3]:
```
# Import libraries and functions
```

In [4]:
```
import numpy as np
```

In [5]:
```
from PlaneRotationFn import planeRotation1
from PlaneRotationFn import planeRotation2
```

In [6]:
```
import scipy
```

In [7]:
```
import sympy
```

In [8]:
```
from IPython.display import display
```

In [9]:
```
from sympy.interactive import printing
```

In [10]:
```
np.set_printoptions(precision=6)
```

In [11]:
```
#np.set_printoptions(suppress=True)
```

In [12]:
```
# Original system:
```

In [13]:
```
A = np.loadtxt('A_ex1.txt')
```

In [14]:
```
A
```

Out[14]:
```
array([[ 1.38  , -0.2077,  6.715 , -5.676 ],
       [-0.5814, -4.29  ,  0.    ,  0.675 ],
       [ 1.067 ,  4.273 , -6.654 ,  5.893 ],
       [ 0.048 ,  4.273 ,  1.343 , -2.104 ]])
```

In [15]:
```
n,nc = A.shape
```

In [16]:
```
B = np.loadtxt('B_ex1.txt')
```

In [17]:
```
B
```

Out[17]:
```
array([[ 0.   ,  0.   ],
       [ 5.679,  0.   ],
       [ 1.136, -3.146],
       [ 1.136,  0.   ]])
```

In [18]:
```
nr,m = B.shape
```

In [19]:
```
# Compute eigenvalues/poles of A to determine system stability:
```

In [20]:
```
A_eigvals, M = np.linalg.eig(A)
```

In [21]:
```
A_eigvals
```
Out[21]:
```
array([ 1.99096 ,  0.063508, -5.056574, -8.665894])
```

In [22]:
```
# Two poles lie in the RHP and are unstable.
```

In [23]:
```
A_eigvals_desired = np.array([-0.2,-0.5,A_eigvals[2],A_eigvals[3]])
```

In [24]:
```
A_eigvals_desired
```
Out[24]:
```
array([-0.2     , -0.5     , -5.056574, -8.665894])
```

In [25]:
```
Lambda = np.diag(A_eigvals_desired)
```

In [26]:
```
Lambda
```
Out[26]:
```
array([[-0.2     ,  0.      ,  0.      ,  0.      ],
       [ 0.      , -0.5     ,  0.      ,  0.      ],
       [ 0.      ,  0.      , -5.056574,  0.      ],
       [ 0.      ,  0.      ,  0.      , -8.665894]])
```

In [27]:
```
# Pole Assignment Algorithm from journal paper
```

In [28]:
```
# Step A: Decomposition of B using SVD
# B = U*S*V.H
```

In [29]:
```
U, s, VH = np.linalg.svd(B)
```

In [30]:
```
U
```
Out[30]:
```
array([[  6.016779e-18,  -8.304906e-17,  -9.868038e-01,  -1.619203e-01],
       [ -9.460842e-01,  -2.577794e-01,   3.176055e-02,  -1.935609e-01],
       [ -2.628862e-01,   9.648268e-01,  -2.220446e-16,   0.000000e+00],
       [ -1.892501e-01,  -5.156495e-02,  -1.587748e-01,   9.676342e-01]])
```

In [31]:
```
s
```
Out[31]:
```
array([ 5.944254,  3.065158])
```

In [32]:
```
S = np.zeros((4, 2))
S[:2, :2] = np.diag(s)
```

In [33]:
```
S
```
Out[33]:
```
array([[ 5.944254,  0.      ],
       [ 0.      ,  3.065158],
       [ 0.      ,  0.      ],
       [ 0.      ,  0.      ]])
```

In [34]:
```
VH
```
Out[34]:
```
array([[-0.990274,  0.139133],
       [-0.139133, -0.990274]])
```

In [35]:
```
# Extract U_0 and U_1 from matrix U = [U_0,U_1]
```

In [36]:
```
U_0 = U[:n,:m]
```

In [37]:
```
U_0
```

Out[37]:
```
array([[  6.016779e-18,  -8.304906e-17],
       [ -9.460842e-01,  -2.577794e-01],
       [ -2.628862e-01,   9.648268e-01],
       [ -1.892501e-01,  -5.156495e-02]])
```

In [38]:
```
U_1 = U[:n,m:]
```

In [39]:
```
U_1
```

Out[39]:
```
array([[ -9.868038e-01,  -1.619203e-01],
       [  3.176055e-02,  -1.935609e-01],
       [ -2.220446e-16,   0.000000e+00],
       [ -1.587748e-01,   9.676342e-01]])
```

In [40]:
```
# B = [U_0,U_1][Z,0].T
# Compute Z from SVD of B
```

In [41]:
```
Z = np.diag(s).dot(VH)
```

In [42]:
```
Z
```

Out[42]:
```
array([[-5.886439,  0.82704 ],
       [-0.426464, -3.035345]])
```

In [43]:
```
# U_1.T *(A - lambda_j*I)
# Compute S_hat_j and S_j

for j in range(len(A_eigvals_desired)):

    lambda_j = A_eigvals_desired[j]

    # M_j is a temp matrix
    exec("M_%d = np.dot(U_1.T,(A - lambda_j*np.identity(n)))" %(j+1))

    # U_1.T *(A - lambda_j*I) = T_j *[Gamma_j,0]*[S_j_hat,S_j].T
    exec("T_%d, gamma_%d, SH_%d = np.linalg.svd(M_%d)" %(j+1,j+1,j+1,j+1))

    exec("S_hat_%d = SH_%d[:m,:].T" %(j+1,j+1))
    exec("S_%d = SH_%d[m:,:].T" %(j+1,j+1))
```

In [44]:
```
# Initial eigenvectors in X_tilde
X_tilde = np.eye(n)
X_tilde
```

Out[44]:
```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

In [45]:
```
# Initial eigenvectors in X
X = np.zeros((n,n))
X
```

Out[45]:
```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

In [46]:
```
# Step X with Method 2/3

maxiter = 1
v4prev = 0
v4current = 0

for r in range(maxiter):

    for j in range(n):

        xt_j = X_tilde[:,j].reshape((n,1))

        for k in range(j+1,n,1):

            xt_k = X_tilde[:,k].reshape((n,1))

            exec("phi_j = np.linalg.norm(np.dot(S_hat_%d.T,xt_j))" %(j+1))
            exec("phi_k = np.linalg.norm(np.dot(S_hat_%d.T,xt_k))" %(k+1))

            if (phi_j < phi_k):

                sin_theta = phi_j
                cos_theta = np.sqrt(1 - phi_j**2)
                protation = planeRotation2(n,cos_theta,sin_theta,j,k)
                v4current = v4current + phi_j**2

            else:

                sin_theta = phi_k
                cos_theta = np.sqrt(1 - phi_k**2)
                protation = planeRotation2(n,cos_theta,sin_theta,j,k)
                v4current = v4current + phi_k**2

            X_tilde = np.dot(protation,X_tilde)

    v4current = np.sqrt(v4current)
    print(v4current - v4prev)
    v4prev = v4current
```

```
1.18443024921
```

In [47]:
```
# Compute eigenvectors x_j in X from X_tilde

for j in range(n):

    xt_j = X_tilde[:,j].reshape((n,1))
    exec("x_j = np.dot(np.dot(S_%d,S_%d.T),xt_j) / np.linalg.norm(np.dot(S
_%d.T,xt_j))" %(j+1,j+1,j+1))
    X[:,j] = x_j[:,0]
```

In [48]:
```
X
```

Out[48]:
```
array([[ 0.948809, -0.093524, -0.671977,  0.142779],
       [ 0.081417,  0.084568, -0.246883, -0.879926],
       [ 0.035477,  0.657038,  0.694792,  0.126892],
       [ 0.303108,  0.743238,  0.068987,  0.435021]])
```

In [49]:
```
np.linalg.matrix_rank(X)
```

Out[49]:
```
4
```

In [50]:
```
X_inv = np.linalg.inv(X)
```

In [51]:
```
X_inv
```

Out[51]:
```
array([[ 1.338859, -0.09351 ,  1.363573, -1.026314],
       [-0.554042,  0.641841, -0.468308,  1.61671 ],
       [ 0.46658 , -0.426158,  1.894783, -1.567827],
       [-0.060276, -0.963857, -0.450465,  0.500306]])
```

In [52]:
```
# M defined as A + BF
M = X.dot(Lambda).dot(X_inv)
```

In [53]:
```
M
```

Out[53]:
```
array([[ 1.38    , -0.2077  ,  6.715   , -5.676   ],
       [ 0.12447 , -7.907368, -1.071934,  1.806108],
       [-1.400423,  2.346898, -6.017361,  4.434204],
       [ 0.189199,  3.549399,  1.128575, -1.877739]])
```

In [54]:
```
# Compute feedback matrix F
F = np.dot(np.linalg.inv(Z),np.dot(U_0.T,(M - A)))
```

In [55]:
```
F
```

Out[55]:
```
array([[ 0.124295, -0.636973, -0.188754,  0.199174],
       [ 0.829187,  0.382232, -0.270522,  0.535619]])
```

In [56]:
```
np.linalg.norm(F)
```

Out[56]:
```
1.300078330255144
```

In [57]:
```
np.linalg.cond(X)
```

Out[57]:
```
4.93555471970971
```