HW03

October 14, 2015

1 By Brandon Lampe

```
In [1]: import numpy as np
        import scipy.linalg as la
        import ipdb

        np.set_printoptions(precision=3) # precision for numpy operations
        %precision 3

Out[1]: u'%.3f'
```

Subroutines Written for the Assignment

1.1 Problem 2

Compare results of matrix multiplication using inner and outer products

```
In [2]: np.random.seed(seed=100)
        A = np.random.random((4,3))
        B = np.random.random((3,2))

In [3]: print A

[[ 0.543  0.278  0.425]
 [ 0.845  0.005  0.122]
 [ 0.671  0.826  0.137]
 [ 0.575  0.891  0.209]]

In [4]: print B

[[ 0.185  0.108]
 [ 0.22   0.979]
 [ 0.812  0.172]]
```

Method A: Inner product between rows of A and columns of B Must perform a double loop to get the inner product for each component of C

```
In [5]: nrow = A.shape[0]
        ncol = B.shape[1]
        C_a = np.zeros((nrow, ncol))

        for i in xrange(nrow):
            for j in xrange(ncol):
                C_a[i,j] = A[i,:].dot(B[:,j])

        print C_a
```

```
[[ 0.506  0.404]
 [ 0.256  0.117]
 [ 0.417  0.904]
 [ 0.472  0.971]]
```

Method B: Outer product between columns of A and rows of B A single loop is needed and the sum of the outer produts is then the equivalent to Method A

```
In [6]: nrow = A.shape[0]
        ncol = B.shape[1]
        C_b = np.zeros((nrow, ncol))

        nloop_outer = A.shape[1]# could have used B.shape[0] also
        for i in xrange(nloop_outer):
            C_b = C_b + np.outer(A[:,i],B[i,:])

        print C_b
```

```
[[ 0.506  0.404]
 [ 0.256  0.117]
 [ 0.417  0.904]
 [ 0.472  0.971]]
```

1.2 Problem 3

Obtain an orthonormal basis via Gram-Schmidt

```
In [7]: np.random.seed(seed=15)
        v1 = np.random.randint(low=0, high=10, size=4)
        v2 = np.random.randint(low=0, high=10, size=4)
        v3 = np.random.randint(low=0, high=10, size=4)
        v4 = 2*v1# + v2 + v3
```

```
In [8]: print v1
```

```
[8 5 5 7]
```

```
In [9]: print v2
```

```
[0 7 5 6]
```

```
In [10]: print v3
```

```
[1 7 0 4]
```

```
In [11]: print v4
```

```
[16 10 10 14]
```

```
In [12]: A_3 = np.array([v1, v4, v2, v3])
         print A_3
```

```
[[ 8  5  5  7]
 [16 10 10 14]
 [ 0  7  5  6]
 [ 1  7  0  4]]
```

```
In [13]: def GS(A):
             """Performs the Gram-Schmidt procedure of orthonormalising a set of vectors
             A: a nxn square matrix where each row is an independent vector space

             Returns:
             Q = orthonormal vector space of dimension nxn
             vspace = integer value of the number of independent vector spaces
                     if vspace < n, one or more of the vector spaces in A was not independent
             """
             nrow = A.shape[0]
             ncol = A.shape[1]

             Q = np.zeros((nrow, ncol))
             neg_terms = np.zeros(nrow)
             cnt = 0
             vspace = 0
             for i in xrange(ncol):
                 for j in xrange(cnt):
                     neg_terms = neg_terms - Q[:,j].dot(A[:,cnt]) * Q[:,j]
                 Q_star = A[:,cnt] + neg_terms
                 Q[:,i] = Q_star / np.sqrt(Q_star.dot(Q_star))
                 # increment/clear terms
                 cnt = cnt + 1
                 neg_terms = np.zeros(nrow)

                 # check if vector space is independent
                 norm_check = np.sqrt(Q_star.dot(Q_star))
                 machine_accuracy = 7./3 - 4./3 - 1
                 if norm_check > machine_accuracy * 10**3:
                     vspace = vspace + 1
                 else:
                     print "Input Matrix is NOT Linearly Independent"
             return Q, vspace
```

```
In [14]: Q_3, vector_space = GS(A_3)
         print Q_3
         print 'Vector space is of dimension:'
         print vector_space
```

```
Input Matrix is NOT Linearly Independent
[[ 0.447 -0.017  0.018  0.   ]
 [ 0.893 -0.034  0.037  0.   ]
 [ 0.     0.74   0.673  0.124]
 [ 0.056  0.672 -0.739  0.992]]
Vector space is of dimension:
3
```

Check if Q is orthonormal Is the Definition of an orthonormal matrix met: $[Q][Q]^T = [I]$

```
In [15]: Q_3.dot(np.transpose(Q_3))
```

```
Out[15]: array([[  2.000e-01,   4.000e-01,  -1.475e-16,  -3.469e-17],
                [  4.000e-01,   8.000e-01,  -2.949e-16,  -6.939e-17],
                [ -1.475e-16,  -2.949e-16,   1.015e+00,   1.231e-01],
                [ -3.469e-17,  -6.939e-17,   1.231e-01,   1.985e+00]])
```

off diagonal components in the last row and last column are nonzero; therefore, Q is not orthonormal because the second vector is not independent

transform the first vector into the new orthonormal basis

```
In [16]: e1 = v1.dot(Q_3)
         print e1
```

```
[ 8.428  8.099 -1.475  7.566]
```

the second component of the vector e_1 is still a linear combination of the original vector v_1

```
In [17]: e1[0]*2
```

```
Out[17]: 16.856
```

1.3 Problem 4

create 4 independent vectors

```
In [18]: np.random.seed(seed=15)
         v1 = np.random.randint(low=0, high=10, size=4)
         v2 = np.random.randint(low=0, high=10, size=4)
         v3 = np.random.randint(low=0, high=10, size=4)
         v4 = np.random.randint(low=0, high=10, size=4)
```

```
In [19]: A_4 = np.transpose(np.array([v1, v2, v3, v4]))
         print A_4
```

```
[[8 0 1 9]
 [5 7 7 7]
 [5 5 0 5]
 [7 6 4 3]]
```

1.3.1 4 (i) Apply Gram-Schmidt to obtain the matrix $[Q]$

```
In [20]: Q_4, vector_space = GS(A_4)
         print Q_4
         print 'Vector space is of dimension:'
         print vector_space
```

```
[[ 0.627 -0.737  0.147  0.207]
 [ 0.392  0.57   0.582  0.428]
 [ 0.392  0.275 -0.8    0.362]
 [ 0.548  0.238 -0.012 -0.802]]
Vector space is of dimension:
4
```

Is the Definition of an orthonormal matrix met: $[Q][Q]^T = [I]$

```
In [21]: Q_4.dot(np.transpose(Q_4))
```

```
Out[21]: array([[  1.000e+00,  -2.220e-16,   1.665e-16,   4.718e-16],
                [ -2.220e-16,   1.000e+00,  -2.776e-16,   1.110e-16],
                [  1.665e-16,  -2.776e-16,   1.000e+00,  -3.331e-16],
                [  4.718e-16,   1.110e-16,  -3.331e-16,   1.000e+00]])
```

yes, diagonal components are the only nonzero values and they have values of unity

1.3.2 4 (ii) Find the matrix $[R] = [Q]^T[A]$. Is $[R]$ upper diagonal?

No, $[R]$ is not upper diagonal... $[R]$ is in the wrong basis

```
In [22]: def QR(A):
             """
             1 - Calls GS, the Gram-Schmidt procedure of orthonormalising a set of vectors
             2 - Calculates [Q] and [R] for QR decomposition

             input:
             A = a nxn square matrix where each row is an independent vector space

             Returns:
             Q = orthonormal vector space of dimension nxn
             R = upper diagonal
             vspace = integer value of the number of independent vector spaces
                     if vspace < n, one or more of the vector spaces in A was not independent
             """

             nrow = A.shape[0]
             ncol = A.shape[1]

             Q = np.zeros((nrow, ncol))
             R = np.zeros((nrow, ncol))
             neg_terms = np.zeros(nrow)
             diag_vect = np.zeros(nrow)
             diag_vect_sum = np.zeros(nrow)
             diag_norm = 0
             Q, vector_space = GS(A)

         #     ipdb.set_trace()
             for i in xrange(nrow):
                 for j in xrange(i+1, ncol, 1):
                     R[i,j] = Q[:,i].dot(A[:,j])
                 for k in xrange(0, i):
                     diag_vect_sum = diag_vect_sum + Q[:,k].dot(A[:,i])*Q[:,k]
                 diag_vect = A[:,i] - diag_vect_sum
                 diag_norm = np.sqrt(diag_vect.dot(diag_vect))
                 R[i,i] = diag_norm
                 diag_vect_sum = np.zeros(nrow) # zero the summation
             return Q, R
```

```
In [23]: Q_4, R_4= QR(A_4)
```

```
In [24]: A_ap = Q_4.dot(R_4)
         print A_ap
```

```
[[ 8.  0.  1.  9.]
 [ 5.  7.  7.  7.]
 [ 5.  5.  0.  5.]
 [ 7.  6.  4.  3.]]
```

```
In [25]: print Q_4
```

```
[[ 0.627 -0.737  0.147  0.207]
 [ 0.392  0.57   0.582  0.428]
```

```
 [ 0.392  0.275 -0.8    0.362]
 [ 0.548  0.238 -0.012 -0.802]]
```

```
In [26]: print R_4
```

```
[[ 12.767   7.989   5.561  11.984]
 [  0.      6.795   4.205  -0.551]
 [  0.      0.      4.171   1.36 ]
 [  0.      0.      0.      4.27 ]]
```

```
In [27]: Q_4.dot(np.transpose(Q_4))
```

```
Out[27]: array([[  1.000e+00,  -2.220e-16,   1.665e-16,   4.718e-16],
                [ -2.220e-16,   1.000e+00,  -2.776e-16,   1.110e-16],
                [  1.665e-16,  -2.776e-16,   1.000e+00,  -3.331e-16],
                [  4.718e-16,   1.110e-16,  -3.331e-16,   1.000e+00]])
```

1.3.3 4 (iii) Write program to perform back substitution

```
In [28]: b = np.array(np.arange(1,5))
         print b
```

```
[1 2 3 4]
```

```
In [29]: b_hat = np.transpose(Q_4).dot(b)
         print b_hat
```

```
[ 4.778  2.182 -1.138 -1.056]
```

```
In [30]: def BackSub(R, b):
             """
             Performs back substitution on a square-upper-diagonal matrix [R] and vector {b} to solve f
             where: [R]{x} = {b}

             input:
             R = nxn square upper diagonal matrix
             b = n length array

             output:
             x = n length array containing the solution: {x} = [R inv] {b}
             """
             nrow = R.shape[0]
             ncol = R.shape[1]
             cnt = 0
             x = np.zeros(nrow)

             # ipdb.set_trace()

             for i in reversed(xrange(nrow)):
                 num_star = 0
                 for j in np.arange(nrow - cnt, nrow,1):
                     # this loop is skipped on first i loop
                     num_star = num_star - R[i,j]* x[j]
                 cnt = cnt + 1
                 num = b[i] + num_star
                 den = R[i,i]
                 x[i] = num / den
             return x
```

```
In [31]: x = BackSub(R_4, b_hat)
         print x
```

```
[ 0.427  0.42  -0.192 -0.247]
```

1.3.4 4 (iv) Compute a scalar measure of error

```
In [32]: def QR_solve(A, b, opt = 0):
             """
             Wrapper for the QR and BackSub routines.
             Given [A]{x}={b}, this function will solve for {x} using the QR algorithm

             input:
             A = a nxn square matrix where each row is an independent vector space
             b = nxm length array, where
             opt: determines output type:
                 opt = 0: returns only {x}
                 opt = 1: returns {x}, [R], and [Q]

             output:
             x = n length array containing the solution: {x} = [R inv] {b}
             """
             Q_orth = np.zeros((A.shape))
             R_ud = np.zeros((A.shape))

             Q_orth, R_ud = QR(A) # performs Gram-Schmidt procedure and obtains [Q] and [R]
             b_dim = len(b.shape)
             nrow = b.shape[0]

             if b_dim > 1: # if b is a 2D array
                 ncol = b.shape[1]
                 x = np.zeros((nrow, ncol))
                 for i in xrange(ncol):
                     b_hat = np.transpose(Q_orth).dot(b[:,i])
                     x[:,i] = BackSub(R_ud, b_hat)
             else: # if be is a vector
                 x = np.zeros(nrow)
                 b_hat = np.transpose(Q_orth).dot(b)
                 x = BackSub(R_ud, b_hat)

             if opt == 0:
                 return x
             if opt != 0:
                 return x, R_ud, Q_orth
```

the exact solution $\{x\}^{ex}$

```
In [33]: np.random.seed(seed=20)
         x_ex = np.random.randint(low=0, high=10, size=4)
         print x_ex
```

```
[3 9 4 6]
```

```
In [34]: print A_4
```

```
[[8 0 1 9]
 [5 7 7 7]
 [5 5 0 5]
 [7 6 4 3]]
```

```
In [35]: b_4 = A_4.dot(x_ex)
         print b_4
```

```
[ 82 148  90 109]
```

the approximate solution

```
In [36]: x_ap, R, Q_ap, = QR_solve(A_4, b_4, opt = 1)
         print x_ap
```

```
[ 3.  9.  4.  6.]
```

check to see if I got the original matrix back

```
In [37]: A_ap = Q_ap.dot(R)
         print A_ap
```

```
[[ 8.  0.  1.  9.]
 [ 5.  7.  7.  7.]
 [ 5.  5.  0.  5.]
 [ 7.  6.  4.  3.]]
```

Error based on the exact solution

```
In [38]: x_diff = x_ex - x_ap
         error = np.sqrt(x_diff.dot(x_diff))/np.sqrt(x_ex.dot(x_ex))
         print '%.3e' % error
```

```
2.755e-15
```

Error based on a residual

```
In [39]: r = b_4 - A_4.dot(x_ap)
         error_r = np.sqrt(r.dot(r))/np.sqrt(b_4.dot(b_4))
         print '%.3e' % error_r
```

```
3.411e-16
```

1.3.5 4 (v) Iterative Improvement

```
In [40]: delta_1 = QR_solve(A_4, r)
         x_ap1 = x_ap + delta_1
```

Error based on the exact solution - with one iterative improvement

```
In [41]: x_diff = x_ex - x_ap1
         error = np.sqrt(x_diff.dot(x_diff))/np.sqrt(x_ex.dot(x_ex))
         print '%.3e' % error
```

```
5.689e-16
```

Error based on a residual - with one iterative improvement

```
In [42]: r = b_4 - A_4.dot(x_ap1)
         error_r = np.sqrt(r.dot(r))/np.sqrt(b_4.dot(b_4))
         print '%.3e' % error_r
```

```
6.446e-17
```

1.3.6 4 (vi) Calculate the Inverse

```
In [43]: def Inv(A):
             """
             Calculates the inverse of matrix [A] using the QR_solve routine
             """
             nrow = A.shape[0]
             ncol = A.shape[1]
             I = np.identity(nrow)
             A_inv = np.zeros((nrow, ncol))
             x = np.zeros(nrow)

             for i in xrange(ncol):
                 x = QR_solve(A_4, I[:,i])
                 A_inv[:,i] = x
             return A_inv
```

```
In [44]: A_1 = Inv(A_4)
         print A_1
```

```
[[ 0.068 -0.126 -0.068  0.204]
 [-0.117  0.026  0.183 -0.016]
 [ 0.019  0.107 -0.219  0.058]
 [ 0.049  0.1    0.085 -0.188]]
```

Check to see that $[A]^{-1}\dot{[}A] = [I]$

```
In [45]: I_ap = A_1.dot(A_4)
         I_ex = np.identity(A_4.shape[0])
```

```
In [46]: I_error = I_ex - I_ap
         frob_norm = la.norm(I_error)
         Error_scalar = frob_norm / la.norm(I_ex)
         print '%.3e' % Error_scalar
```

```
1.685e-15
```