CS111 F21 Homework 7

Michael Glushchenko, 9403890
Partners with: Shiv Kapoor

November 14, 2021

QR Factorization

1 Problem 1

1.1

```
# 1.1
A = np.array([[4,-1,-1], [-1,4,-1], [-1,-1,4]])
b = (np.array([15,-3,12])).T

Q, R = spla.qr(A) # QR factorization

print("Matrix Q:\n", Q)
print("Matrix R:\n", R)
print("Is R upper-triangular?", (R == np.triu(R)).all())
print("Comparing Q to the Identity matrix:", npla.norm(Q.T@Q - np.eye(3))/npla.norm(np.eye(3)))
print("Compare Q@R to A:", npla.norm(Q@R - A)/npla.norm(A))
```

```
Matrix Q:
 [[-0.9428 -0.1421  0.3015]
 [ 0.2357 -0.9239  0.3015]
 [ 0.2357  0.3553  0.9045]]
Matrix R:
 [[-4.2426  1.6499  1.6499]
 [ 0.     -3.9087  2.4873]
 [ 0.      0.      3.0151]]
Is R upper-triangular? True
Comparing Q to the Identity matrix: 2.846963623252602e-16
Compare Q@R to A: 2.3599796770763007e-16
```

1.2

```
# 1.2
A = np.array([[4,-1,-1], [-1,4,-1], [-1,-1,4]])
b = (np.array([15,-3,12])).T

Q, R = spla.qr(A) # QR factorization

x = cs111.Usolve(R, Q.T @ b)
print("Relative residual norm:", npla.norm(A@x-b, 2)/npla.norm(b, 2))
```

```
Relative residual norm: 1.8835570444992786e-16
```

The relative residual norm being almost 0 confirms our solution.

2 Problem 2

2.1

Here's the code used for this question (didn't comment out m or n, just redundant):

```
# 2.1
m = 9
n = 5
A = np.random.rand(9, 5)
Q1, R1 = spla.qr(A)
A_approx = Q1@R1

print("shape of Q1:", Q1.shape)
print("shape of Q2:", R1.shape)
print("is Q1 orthogonal?", npla.norm(Q1.T@Q1 - np.eye(m)) / npla.norm(np.eye(m)))
print("is R1 upper-triangular?", (R1 == np.triu(R1)).all())
print("matrix A:\n", A)
print("matrix Q1@R1:\n", A_approx)
print("relative residual norm:", npla.norm(A_approx - A) / npla.norm(A))
```

It produces the following output, answering all the questions:

```
shape of Q1: (9, 9)
shape of Q2: (9, 5)
is Q1 orthogonal? 3.414381609999984e-16
is R1 upper-triangular? True
matrix A:
 [[0.2082 0.013  0.2922 0.1378 0.4909]
 [0.9169 0.3487 0.0511 0.3952 0.41  ]
 [0.2264 0.9199 0.305  0.8026 0.0832]
 [0.7749 0.7331 0.4988 0.5702 0.1916]
 [0.068  0.5028 0.6253 0.1664 0.6542]
 [0.6203 0.9374 0.5969 0.7626 0.9977]
 [0.0958 0.9496 0.4953 0.5035 0.7989]
 [0.2367 0.9432 0.1876 0.4713 0.3002]
 [0.3388 0.6957 0.6239 0.053  0.8116]]
matrix Q1@R1:
 [[0.2082 0.013  0.2922 0.1378 0.4909]
 [0.9169 0.3487 0.0511 0.3952 0.41  ]
 [0.2264 0.9199 0.305  0.8026 0.0832]
 [0.7749 0.7331 0.4988 0.5702 0.1916]
 [0.068  0.5028 0.6253 0.1664 0.6542]
 [0.6203 0.9374 0.5969 0.7626 0.9977]
 [0.0958 0.9496 0.4953 0.5035 0.7989]
 [0.2367 0.9432 0.1876 0.4713 0.3002]
 [0.3388 0.6957 0.6239 0.053  0.8116]]
relative residual norm: 3.561140417582373e-16
```

2.2

Here's the code for this question:

```
# 2.2
m = 9
n = 5
A = np.random.rand(9, 5)
Q1, R1 = spla.qr(A)
Q2, R2 = spla.qr(A, mode = 'economic')
A_approx = Q2@R2

print("shape of Q1:", Q2.shape)
print("shape of Q2:", R2.shape)
print("what is Q2^T@Q2\n", Q2.T@Q2)
print("is Q2 orthogonal?", npla.norm(Q2.T@Q2 - np.eye(n)) / npla.norm(np.eye(n)))
print("is R2 upper-triangular?", (R2 == np.triu(R2)).all())
print("matrix A:\n", A)
print("matrix Q2@R2:\n", A_approx)
print("relative residual norm:", npla.norm(A_approx - A) / npla.norm(A))
print("forbenius norm of the difference of Q1 and Q2", (npla.norm(Q1) - npla.norm(Q2))/npla.norm(Q2))
print("forbenius norm of the difference of R1 and R2", (npla.norm(R1) - npla.norm(R2))/npla.norm(R2))
```

The following output is produced as a result:

```
shape of Q1: (9, 5)
shape of Q2: (5, 5)
what is Q2^T@Q2
 [[ 1.0000e+00  8.3893e-18 -4.7813e-17  1.1181e-17 -1.5985e-17]
 [ 8.3893e-18  1.0000e+00  2.5388e-17 -2.8663e-18  5.7455e-18]
 [-4.7813e-17  2.5388e-17  1.0000e+00 -6.0493e-17 -5.6627e-17]
 [ 1.1181e-17 -2.8663e-18 -6.0493e-17  1.0000e+00 -7.5284e-17]
 [-1.5985e-17  5.7455e-18 -5.6627e-17 -7.5284e-17  1.0000e+00]]
is Q2 orthogonal? 1.6902462501157263e-16
is R2 upper-triangular? True
matrix A:
 [[0.7503 0.452  0.0392 0.6544 0.2789]
 [0.3044 0.8543 0.6376 0.0746 0.8317]
 [0.5189 0.6017 0.5424 0.6161 0.8733]
 [0.3785 0.0525 0.3977 0.0259 0.9383]
 [0.7543 0.3721 0.6858 0.4428 0.833 ]
 [0.6558 0.0855 0.4643 0.8344 0.523 ]
 [0.1241 0.3078 0.451  0.7635 0.0225]
 [0.6033 0.777  0.4679 0.4866 0.33  ]
 [0.311  0.682  0.5371 0.5864 0.7176]]
matrix Q2@R2:
 [[0.7503 0.452  0.0392 0.6544 0.2789]
 [0.3044 0.8543 0.6376 0.0746 0.8317]
 [0.5189 0.6017 0.5424 0.6161 0.8733]
 [0.3785 0.0525 0.3977 0.0259 0.9383]
 [0.7543 0.3721 0.6858 0.4428 0.833 ]
 [0.6558 0.0855 0.4643 0.8344 0.523 ]
 [0.1241 0.3078 0.451  0.7635 0.0225]
 [0.6033 0.777  0.4679 0.4866 0.33  ]
 [0.311  0.682  0.5371 0.5864 0.7176]]
relative residual norm: 1.3126278495071444e-16
forbenius norm of the difference of Q1 and Q2 0.34164078649987395
forbenius norm of the difference of R1 and R2 0.0
```

We generate the economy-sized QR factorization of A, confirm that Q_2 is orthogonal, confirm by inspection that R_2 is upper-triangular, and then verify that $Q_2R_2 = A$. We can clearly see from this problem that Q_2 is exactly matrix Q_1, truncated to be an m-by-n matrix (since Q_1 is m-by-m). This truncation results in the forbenius norm of 0.3416, since we truncate nonzero values from matrix Q_1 in the process.

As for R_1 and R_2, R_2 is the same matrix as R_1, truncated to be n-by-n. The forbenius norm here is 0 since we truncate 0s from R_1 to obtain R_2

3 Problem 3

3.1

```
# generate the random 9x1 vector
b = np.random.rand(m)

# 3.1
x = npla.lstsq(A, b, rcond = None)[0] # computing x here
r = b - A@x                            # calculate the residual
print("x:", x)
print("relative residual norm:", npla.norm(r, 2)/npla.norm(b, 2))

# verify the residual is orthogonal
print("Verifying orthogonality:", npla.norm(A.T@r, 2)) # should be close to 0
```

```
x: [ 0.3062  0.2361 -0.0867  0.0186  0.28  ]
relative residual norm: 0.5402798672869803
Verifying orthogonality: 4.433173899242564e-16
```

3.2

```
# 3.2
y = Q1.T@b
x = cs111.Usolve(R1[:min(A.shape)], y[:min(A.shape)]) # compute x
r = b - A@x                            # calculate the residual
print("x:", x)
print("relative residual norm:", npla.norm(r, 2)/npla.norm(b, 2))

# verify the residual is orthogonal
print("Verifying orthogonality:", npla.norm(A.T@r, 2)) # should be close to 0
```

```
x: [ 0.3062  0.2361 -0.0867  0.0186  0.28  ]
relative residual norm: 0.5402798672869803
Verifying orthogonality: 1.0335396186701975e-15
```

3.3

```
# 3.3
y = Q2.T@b
x = cs111.Usolve(R2, y) # compute x
r = b - A@x                            # calculate the residual
print("x:", x)
print("relative residual norm:", npla.norm(r, 2)/npla.norm(b, 2))

# verify the residual is orthogonal
print("Verifying orthogonality:", npla.norm(A.T@r, 2)) # should be close to 0
```

```
x: [ 0.3062  0.2361 -0.0867  0.0186  0.28  ]
relative residual norm: 0.5402798672869803
Verifying orthogonality: 1.0335396186701975e-15
```

We can see that **3.3** gives us the same result as **3.2** and **3.1** do. This is because in **3.2**, we use QR factorization, then extract a submatrix from R_1, and slice y to craft the input for Usolve. In **3.3** we basically do the reverse, where the QR factorization takes care of the dimension issue by providing a square R_2 for us (ends up being exactly the same matrix as the one we extracted from R_1); the truncation in **3.2** makes the y input also be exactly the same as the y input in **3.3**. Section **2.2** explains it well.

3.4

```
# 3.4
b = A@np.ones(5)# change b

y = Q2.T@b
x = cs111.Usolve(R2, y) # compute x
r = b - A@x                              # calculate the residual

print("x:", x)
print("relative residual norm:", npla.norm(r, 2)/npla.norm(b, 2))
print("Verifying orthogonality:", npla.norm(A.T@r, 2)) # should be close to 0
```

```
x: [1. 1. 1. 1. 1.]
relative residual norm: 2.402811363312991e-16
Verifying orthogonality: 5.985770209381298e-15
```

The new relative residual norm is much smaller in **3.4** than the relative residual norms in the other sections of problem 3. Since our b is a linear combination of A's columns, no round-off error happens during QR factorization; we can see this by examining Q_2 and R_2, and seeing that unlike other sections, there's only 4 digits of precision necessary.