Math 6108 Homework 7

Jacob Hauck

October 21, 2024

Problem 1.

Let
$$B = \left\{ \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 2 \\ 1 & -1 \end{bmatrix} \right\} \subseteq \mathcal{M}_2.$$
Recall that $\langle C, D\rangle = \operatorname{tr}(C^*D)$ in $\mathcal{M}_2$. We can use the Gram-Schmidt process to orthonormalize the elements B; the resulting orthonormal set will be an orthonormal basis for $\operatorname{span}(B)$. We begin by setting

$$\mathbf{v}_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 0 & 2 \\ 1 & -1 \end{bmatrix}.$$

Now we start the Gram-Schmidt process: let $\mathbf{u}_1 = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}$. Since $\|\mathbf{v}_1\| = \sqrt{1^2+1^2+1^2+1^2} = 2$, we have

$$\mathbf{u}_1 = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Next, we compute
$$\begin{aligned}
\widetilde{\mathbf{u}}_2 &= \mathbf{v}_2 - \langle \mathbf{u}_1, \mathbf{v}_2\rangle \mathbf{u}_1 = \mathbf{v}_2 - \operatorname{tr}(\mathbf{u}_1^*\mathbf{v}_2)\mathbf{u}_1 \\
&= \mathbf{v}_2 - \left(\frac{1}{2}\cdot 1 + 0 + 0 + \frac{1}{2}\cdot 1\right)\mathbf{u}_1 \\
&= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\
&= \frac{1}{2}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.
\end{aligned}$$
Then we set $\mathbf{u}_2 = \frac{\widetilde{\mathbf{u}}_2}{\|\widetilde{\mathbf{u}}_2\|}$. Since $\|\widetilde{\mathbf{u}}_2\| = \sqrt{\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}} = 1$, we have $\mathbf{u}_2 = \widetilde{\mathbf{u}}_2$. Lastly, we compute

$$\begin{aligned}
\widetilde{\mathbf{u}}_3 &= \mathbf{v}_3 - \langle \mathbf{u}_1, \mathbf{v}_3\rangle \mathbf{u}_1 - \langle \mathbf{u}_2, \mathbf{v}_3\rangle \mathbf{u}_2 \\
&= \mathbf{v}_3 - \left(\frac{1}{2}\cdot 2 + \frac{1}{2}\cdot 1 - \frac{1}{2}\cdot 1\right)\mathbf{u}_1 - \left(-\frac{1}{2}\cdot 2 - \frac{1}{2}\cdot 1 - \frac{1}{2}\cdot 1\right)\mathbf{u}_2 \\
&= \begin{bmatrix} 0 & 2 \\ 1 & -1 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \\
&= \frac{1}{2}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}.
\end{aligned}$$

Finally, we set $\mathbf{u}_3 = \frac{\widetilde{\mathbf{u}}_3}{\|\widetilde{\mathbf{u}}_3\|}$. Since $\|\widetilde{\mathbf{u}}_3\| = \sqrt{\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}} = 1$, we have $\mathbf{u}_3 = \widetilde{\mathbf{u}}_3$. Since none of the vectors from the Gram-Schmidt process were zero, it follows that B is linearly independent, and $U = \{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ is orthonormal and a subset of $\operatorname{span}(B)$. Then $\dim(\operatorname{span}(B)) = 3$, so U is an orthonormal basis for $\operatorname{span}(B)$.

Problem 2.

Let $U, V \in \mathcal{M}_n$ be unitary. Then UV is also unitary.

Proof. First, we observe that

$$(UV)^*(UV) = V^*U^*UV = V^*V = I$$

by the unitarity of U and V. Second, we observe that

$$(UV)(UV)^* = UVV^*U^* = UU^* = I$$

by the unitarity of U and V. This implies that UV is unitary. □

Problem 3.

Let $U \in \mathcal{M}_n$ be a unitary matrix, and let $\lambda \in \mathbb{C}$ be an eigenvalue of U. Then $|\lambda| = 1$.

Proof. If λ is an eigenvalue of U, then, by definition, there is a nonzero vector $\mathbf{v} \in \mathbb{C}^n$ such that $U\mathbf{v} = \lambda\mathbf{v}$. This implies that

$$|\lambda|^2\|\mathbf{v}\|^2 = \|\lambda\mathbf{v}\|^2 = \|U\mathbf{v}\|^2 = (U\mathbf{v})^*(U\mathbf{v}) = \mathbf{v}^*U^*U\mathbf{v} = \mathbf{v}^*\mathbf{v} = \|\mathbf{v}\|^2.$$

Dividing both sides by $\|\mathbf{v}\|^2 \neq 0$ gives $|\lambda|^2 = 1$, which implies that $|\lambda| = 1$. □

Problem 4.

Consider the overconstrained system $A\mathbf{x} = \mathbf{z}$, where

$$A = \begin{bmatrix} -2 & 1 \\ -1 & 1 \\ 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}, \qquad \mathbf{z} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 2 \\ 2 \end{bmatrix}.$$

We find the least-squares solution $\mathbf{x}$ using two methods.

1. By definition, $\mathbf{x}$ is the minimizer of $\|A\mathbf{x} - \mathbf{z}\|^2$, which is given explicitly by

$$\begin{aligned}
\|A\mathbf{x} - \mathbf{z}\|^2 &= (A\mathbf{x} - \mathbf{z})^T(A\mathbf{x} - \mathbf{z}) \\
&= \mathbf{x}^T A^T A\mathbf{x} - 2\mathbf{z}^T A\mathbf{x} + \mathbf{z}^T\mathbf{z} \\
&= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 10 & 0 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - 2 \cdot \begin{bmatrix} 5 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + 9 \\
&= 10x_1^2 + 5x_2^2 - 10x_1 - 10x_2 + 9 \\
&= 10\left(x_1 - \frac{1}{2}\right)^2 + 5\left(x_2 - 1\right)^2 + 4,
\end{aligned}$$

which is evidently minimal when $x_1 = \frac{1}{2}$, and $x_2 = 1$. Thus, $\mathbf{x} = \frac{1}{2}\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ is the least-squares solution $A\mathbf{x} = \mathbf{z}$.

2. We can also use the fact that the least squares solution $\mathbf{x}$ of $A\mathbf{x} = \mathbf{z}$ is the solution of $A\mathbf{x} = \mathbf{y}$, where $\mathbf{y}$ is the projection of $\mathbf{z}$ onto $\operatorname{col}(A)$. In order to find $\mathbf{y}$, we need an orthonormal basis for $\operatorname{col}(A)$. We observe that the two columns of A are already orthogonal because

$$\begin{bmatrix} -2 \\ -1 \\ 0 \\ 1 \\ 2 \end{bmatrix}^* \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = -2 + -1 + 0 + 1 + 2 = 0.$$

Let $A = \begin{bmatrix}\mathbf{a}_1, \mathbf{a}_2\end{bmatrix}$. Then $\{\mathbf{u}_1, \mathbf{u}_2\}$, with $\mathbf{u}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|}$, and $\mathbf{u}_2 = \frac{\mathbf{a}_2}{\|\mathbf{a}_2\|}$ is an orthonormal basis for $\mathrm{col}(A)$. Since $\|\mathbf{a}_1\| = \sqrt{4+1+0+1+4} = \sqrt{10}$, and $\|\mathbf{a}_2\| = \sqrt{1+1+1+1+1} = \sqrt{5}$, we have $\mathbf{u}_1 = \frac{1}{\sqrt{10}}\begin{bmatrix}-2 & -1 & 0 & 1 & 2\end{bmatrix}^T$, and $\mathbf{u}_2 = \frac{1}{\sqrt{5}}\begin{bmatrix}1 & 1 & 1 & 1 & 1\end{bmatrix}^T$.

Then we get

$$\mathbf{y} = \langle \mathbf{u}_1, \mathbf{z}\rangle \mathbf{u}_1 + \langle \mathbf{u}_2, \mathbf{z}\rangle \mathbf{u}_2 = \frac{1}{10}\begin{bmatrix}-2 & -1 & 01 & 2\end{bmatrix}\begin{bmatrix}0\\1\\0\\2\\2\end{bmatrix}\begin{bmatrix}-2\\-1\\0\\1\\2\end{bmatrix} + \frac{1}{5}\begin{bmatrix}1 & 1 & 1 & 1 & 1\end{bmatrix}\begin{bmatrix}0\\1\\0\\2\\2\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix}$$

$$= \frac{1}{2}\begin{bmatrix}0\\1\\2\\3\\4\end{bmatrix}.$$

Finally, $\mathbf{x}$ is the least-squares solution of $A\mathbf{x} = \mathbf{z}$ if and only if $\mathbf{x}$ is the solution of $A\mathbf{x} = \mathbf{y}$. Since $\mathbf{y} \in \mathrm{col}(A)$, we can use any two rows of A to solve for $\mathbf{x}$ and the result will work for the other rows. Using the third and fourth rows, we have $x_2 = 1$, and $x_1 + x_2 = \frac{3}{2}$, which implies that $x_1 = \frac{1}{2}$. Thus, $\mathbf{x} = \frac{1}{2}\begin{bmatrix}1\\2\end{bmatrix}$ is the least-squares solution of $A\mathbf{x} = \mathbf{z}$.

Problem 5.

To implement QR factorization, we just need to do Gram-Schmidt orthogonalization (normalizing the vectors as well) to obtain the orthogonal factor U. If $A = \begin{bmatrix}\mathbf{a}_1 & \dots & \mathbf{a}_n\end{bmatrix}$, then the upper-triangular factor T is given by

$$T = \begin{bmatrix}\langle \mathbf{u}_1, \mathbf{a}_1\rangle & \langle \mathbf{u}_1, \mathbf{a}_2\rangle & \dots & \langle \mathbf{u}_1, \mathbf{a}_n\rangle \\ 0 & \langle \mathbf{u}_2, \mathbf{a}_2\rangle & \dots & \langle \mathbf{u}_n, \mathbf{a}_2\rangle \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & \langle \mathbf{u}_n, \mathbf{a}_n\rangle\end{bmatrix}.$$

In obtaining $\mathbf{u}_j$, we already compute $\langle \mathbf{u}_i, \mathbf{a}_j\rangle$, for $i = 1, \dots, n-1$; in other words, the jth column of T except for the diagonal element $\langle \mathbf{u}_j, \mathbf{a}_j\rangle$. Thus, we need only save these values as we compute them to obtain $\mathbf{u}_j$ to construct T, and we also need to compute the diagonal element $\langle \mathbf{u}_j, \mathbf{a}_j\rangle$. The diagonal element, however, is also secretly computed as a part of the Gram-Schmidt process because to obtain $\mathbf{u}_j$, we first compute $\widetilde{\mathbf{u}}_j$ via

$$\widetilde{\mathbf{u}}_j = \mathbf{a}_j - \sum_{i=1}^{j-1} \langle \mathbf{u}_i, \mathbf{a}_j\rangle \mathbf{u}_i$$

and then normalize to obtain $\mathbf{u}_j = \frac{\widetilde{\mathbf{u}}_j}{\|\widetilde{\mathbf{u}}_j\|}$. Hence,

$$\langle \mathbf{u}_j, \mathbf{a}_j\rangle = \left\langle \frac{\widetilde{\mathbf{u}}_j}{\|\widetilde{\mathbf{u}}_j\|}, \widetilde{\mathbf{u}}_j + \sum_{i=1}^{j-1} \langle \mathbf{u}_i, \mathbf{a}_j\rangle \mathbf{u}_i \right\rangle = \frac{\langle \widetilde{\mathbf{u}}_j, \widetilde{\mathbf{u}}_j\rangle}{\|\widetilde{\mathbf{u}}_j\|} + \sum_{i=1}^{j-1} \langle \mathbf{u}_i, \mathbf{a}_j\rangle\langle \mathbf{u}_i, \mathbf{u}_j\rangle = \|\widetilde{\mathbf{u}}_j\|$$

because $\mathbf{u}_j$ is orthogonal to $\mathbf{u}_i$ for $i = 1, \dots, j-1$. These considerations lead to Algorithm 1.

A Python implementation of this algorithm is provided in Listing 1. The command `python -m qr` can be used to run the tests, which verify that the function works across a range of input types that cover every code path. The output from running these tests is given in Listing 2.

Algorithm 1: QR Decomposition

Input: Nonsingular matrix $A \in \mathbb{R}^{n\times n}$ with columns $\mathbf{a}_1, \dots, \mathbf{a}_n \in \mathbb{R}^n$

Output: Matrices $U, T \in \mathbb{R}^{n\times n}$ with columns $\mathbf{u}_1, \dots, \mathbf{u}_n$ and $\mathbf{t}_1, \dots, \mathbf{t}_n$ such that U is orthogonal, T is upper-triangular, and $A = UT$.

$c \leftarrow 1$;
repeat
  $\mathbf{t}_c \leftarrow \mathbf{0}$;
  $p \leftarrow 1$;
  repeat
    $(\mathbf{t}_c)_p \leftarrow \langle \mathbf{u}_p, \mathbf{a}_c \rangle$; `// Loop is no-op by convention if` $c = 1$
  until $p = c - 1$;
  $\mathbf{u}_c \leftarrow \mathbf{a}_c - \sum_{p=1}^{c-1} (\mathbf{t}_c)_p \mathbf{u}_p$; `// Sum is 0 by convention if` $c = 1$
  $(\mathbf{t}_t)_c \leftarrow \|\mathbf{u}_c\|$;
  $\mathbf{u}_c \leftarrow \frac{\mathbf{u}_c}{t_c}$;
until $c = n$;

Listing 1: Python implementation of the QR decomposition

```python
import numpy as np


class SingularMatrixError(BaseException):
    """Exception class that is raised to indicate a singular input matrix
    """
    pass


def qr(a, eps_d=1e-10):
    """
    Perform QR decomposition on the given matrix a, returning the matrices
    u and t such that a = ut, where u is orthogonal, and t is upper-triangular.

    :param a: Nonsingular n x n matrix. Raises SingularMatrixError if a is
        singular or almost singular
    :param eps_d: Tolerance for approximate singularity independence (minimum
        norm of the computed orthogonal columns). Default = 10^{-10}
    :return: n x n matrix u whose columns are orthogonal and n x n
        upper-trianguler matrix t such that a = ut.
    """

    # ==== Input Validation ====

    # Ensure input has the correct data type
    a = np.array(a, dtype=float)
    assert len(a.shape) == 2
    assert a.shape[0] == a.shape[1]  # matrix must be square

    # ==== Run Gram-Schmidt process while recording inner products ====

```

```python
    # Initialization

    # The first step for each column is copying the corresponding column from a,
    # so we initialize the output equal to a. Since we already copied the input
    # with np.array(), we can use that memory for our output matrix
    u = a

    # Initialize the upper-triangular matrix with zeros so we don't have to worry
    # about setting the lower part to 0 manually.
    t = np.zeros_like(a)

    # Normalize the first column of u, saving the norm as the upper-left
    # entry of t
    t[0, 0] = np.linalg.norm(u[:, 0])

    # Check for approximate linear dependence before possible divide-by-zero
    if t[0, 0] < eps_d:
        raise SingularMatrixError('Matrix is singular or almost singular '
                                  'because first column is almost 0')
    u[:, 0] /= t[0, 0]

    # Iteration
    for col in range(1, a.shape[1]):
        # Recall that u[:, col] == a[:, col] because of initialization

        # Save inner products in t[:, col]
        t[:col, col] = u[:, :col].T @ u[:, col]

        # Subtract out previous orthonormal columns
        u[:, col] -= u[:, :col] @ t[:col, col]

        # Normalize new column, and save the norm as the current
        # diagonal element of t
        t[col, col] = np.linalg.norm(u[:, col])

        # Check for approximate linear dependence before possible divide-by-zero
        if t[col, col] < eps_d:
            raise SingularMatrixError('Aborting QR factorization. Matrix has '
                                      'linearly dependent or almost linearly '
                                      'dependent columns.')
        u[:, col] /= t[col, col]

    # Return orthogonal columns u and corresponding inner-product matrix t
    return u, t


# Test example
if __name__ == '__main__':
    # Set RNG seed for reproducible results
    np.random.seed(2024)

```

```python
    print('Test 1: random square matrix')
    a = np.random.normal(size=(5, 5))
    print('Input matrix (a)')
    print(a)
    try:
        print()
        print('Orthogonal factor (u)')
        u, t = qr(a)
        print(u)
        print()
        print('Upper-triangular factor (t)')
        print(t)
        print()
        print('u is orthogonal?', np.allclose(u.T @ u, np.eye(5)))
        print('a = ut?', np.allclose(u @ t, a))
    except SingularMatrixError:
        print('Bad luck! You randomly chose a singular matrix')
    print()

    print('Test 2: matrix with too many rows')
    a = np.random.random((5, 3))
    print('Input matrix (a)')
    print(a)
    print()
    try:
        qr(a)
    except AssertionError:  # Should fail validation assertion
        print('Matrix was the wrong size')
    print()

    print('Test 3: matrix with first column 0')
    a = np.array([
        [0, 1, 2],
        [0, 3, 4],
        [0, 5, 6]
    ])
    print('Input matrix (a)')
    print(a)
    print()
    try:
        qr(a)  # should raise an error
    except SingularMatrixError as e:
        print(e)
    print()

    print('Test 4: singular matrix')
    a = np.array([
        [1, 2, -1],
        [2, 5, -3],
        [3, 3, 0]
    ])
```

```python
    print('Input matrix (a)')
    print(a)
    print()
    try:
        qr(a)  # should raise an error
    except SingularMatrixError as e:
        print(e)
    print()
```

Listing 2: Output for test cases

```
>python -m qr
Test 1: random square matrix
Input matrix (a)
[[ 1.66804732  0.73734773 -0.20153776 -0.15091195  0.91605181]
 [ 1.16032964 -2.619962   -1.32529457  0.45998862  0.10205165]
 [ 1.05355278  1.62404261 -1.50063502 -0.27783169  1.19399502]
 [ 0.86181533 -0.41704604 -0.24953642  0.94367735 -0.76631064]
 [ 0.20822873  1.40872293 -1.48910401 -1.47580853  0.99084632]]

Orthogonal factor (u)
[[ 0.67957418  0.22419956  0.51727769 -0.46886764  0.02237023]
 [ 0.47272643 -0.74100785 -0.43180719 -0.13850957  0.1476304 ]
 [ 0.42922479  0.4732441  -0.33826165  0.4980688   0.47886598]
 [ 0.35110961 -0.11263953  0.14832524  0.58494213 -0.7070196 ]
 [ 0.08483385  0.40496207 -0.63995705 -0.41321716 -0.49851329]]

Upper-triangular factor (t)
[[ 2.45454783 -0.06728495 -1.62151243  0.20177634  0.9982582 ]
 [ 0.          3.49274926 -0.34822064 -1.21011363  1.18238049]
 [ 0.          0.          1.89157808  0.90171527 -0.72185849]
 [ 0.          0.          0.          1.03049165 -0.70663258]
 [ 0.          0.          0.          0.          0.6551684 ]]

u is orthogonal? True
a = ut? True

Test 2: matrix with too many rows
Input matrix (a)
[[0.23898683 0.4377217  0.8835387 ]
 [0.28928114 0.78450686 0.75895366]
 [0.41778538 0.22576877 0.42009814]
 [0.06436369 0.59643269 0.83732372]
 [0.89248639 0.20052744 0.50239523]]

Matrix was the wrong size

Test 3: matrix with first column 0
Input matrix (a)
[[0 1 2]
 [0 3 4]
 [0 5 6]]
```

```

Matrix is singular or almost singular because first column is almost 0

Test 4: singular matrix
Input matrix (a)
[[ 1  2 -1]
 [ 2  5 -3]
 [ 3  3  0]]

Aborting QR factorization. Matrix has linearly dependent or almost linearly dependent
    ↪ columns.
```