2025 Spring 3D CV HW0

Yelin Zhang

March 23, 2025

Contents

1 SVD

1.1 Briefly describe SVD: its input, outputs, basic properties, and applications.

SVD can be used to decompose a matrix into three matrices.

Input: Any Matrix A with size $m \times n$.

Output:

$$A = U\Sigma V^T$$

- U: orthogonal $m \times m$ matrix.
- Σ: diagonal $m \times n$ matrix. With singular values in the diagonal.
- V: orthogonal $n \times n$ matrix.

Applications:

- Pseudoinverse (Moore-Penrose inverse) of a matrix. Which is applied to singular or nearly singular matrices.
- Principal Component Analysis

1.2 Describe the relation between SVD and Eigenvalue Decomposition

SVD and Eigenvalue Decomposition are linked by the values in the SVD ouput.
U and V contain the eigenvectors and Σ^2 the eigenvalues due to the formulas:

$$A^T A = V\Sigma^2 V^T$$

$$AA^T = U\Sigma^2 U^T$$

Singular values in Σ are the square roots of the eigenvalues.

$$\sigma_i = \sqrt{\lambda_i}$$

1.3 Describe Principal Component Analysis (PCA) and its relation with SVD.

PCA is used to reduce the dimensionality of data.
The PCA covariance matrix can be created with the SVD of the data.
X is a $m \times n$ matrix and C the covariance matrix.

$$X = U\Sigma V^T$$

$$C = \frac{1}{m-1}(U\Sigma V^T)^T U\Sigma V^T = V\frac{\Sigma^2}{m-1}V^T$$

The eigenvectors in V are the main axis of the data.

1.4 Apply SVD on the following example matrix and show the results

Apply SVD on the following example matrix and show the results, $A = U\Sigma V^T$. Note that you are allowed to use any existing libraries (e.g., svd() in Matlab). Include both your code and output in your pdf submission (Note that the code should be very simple, at most a couple of lines)

$$A = \begin{bmatrix} 2 & 3 & 3 \\ 5 & 3 & 1 \\ 4 & 5 & 1 \end{bmatrix}$$

Code:

```
import numpy as np

A = np.array([[2, 3, 3], [5, 3, 1], [4, 5, 1]])
svd = np.linalg.svd(A, full_matrices=True, compute_uv=True)

print("U:")
print(svd[0])
print("S:", svd[1])
print("V^T:")
print(svd[2])
print("Reconstructed A=U*S*V^T:")
print(svd[0] @ np.diag(svd[1]) @ svd[2])
print("Wrongly reconstructed A=U*S*V:")
print(svd[0] @ np.diag(svd[1]) @ svd[2].T)
```

Output:

```
U:
[[-0.44055623  0.85416795 -0.2762378 ]
 [-0.59895345 -0.50888162 -0.61829949]
 [-0.66870394 -0.10694211  0.73579781]]
S: [9.55899032 2.37444865 1.40985736]
V^T:
[[-0.68529157 -0.67601792 -0.27087862]
 [-0.53226699  0.21105886  0.81984511]
 [-0.49705866  0.70601269 -0.50445889]]
Reconstructed A=U*S*V^T:
[[2. 3. 3.]
 [5. 3. 1.]
 [4. 5. 1.]]
Wrongly reconstructed A=U*S*V:
[[1.62036039 2.35029292 3.72163344]
 [4.97653183 2.0777405  2.4325142 ]
 [4.27113484 4.19921104 2.47467852]]
```

1.5 In above outputs, verify that both U and V are orthogonal matrices.

There are multiple signs that U and V are orthogonal matrices.

- Their determinantes are 1.
- Their inverse is their transpose.
- $UU^T = I$ and $VV^T = I$.

Checked with continuation of above code:

```
print("Determinate of U:", np.linalg.det(svd[0]))
print("Determinate of V:", np.linalg.det(svd[2]))
print("Inverse of U:")
print(np.linalg.inv(svd[0]))
print("Transpose of U:")
print(svd[0].T)
```

```
print("Inverse of V:")
print(np.linalg.inv(svd[2].T))
print("Transpose of V:")
print(svd[2])
print("U*U^T:")
print(svd[0] @ svd[0].T)
print("V*V^T:")
print(svd[2].T @ svd[2])
```

Output:

```
Determinate of U: 1.0
Determinate of V: 1.0000000000000002
Inverse of U:
[[-0.44055623 -0.59895345 -0.66870394]
 [ 0.85416795 -0.50888162 -0.10694211]
 [-0.2762378  -0.61829949  0.73579781]]
Transpose of U:
[[-0.44055623 -0.59895345 -0.66870394]
 [ 0.85416795 -0.50888162 -0.10694211]
 [-0.2762378  -0.61829949  0.73579781]]
Inverse of V:
[[-0.68529157 -0.67601792 -0.27087862]
 [-0.53226699  0.21105886  0.81984511]
 [-0.49705866  0.70601269 -0.50445889]]
Transpose of V:
[[-0.68529157 -0.67601792 -0.27087862]
 [-0.53226699  0.21105886  0.81984511]
 [-0.49705866  0.70601269 -0.50445889]]
U*U^T:
[[ 1.00000000e+00 -8.17791566e-17 -4.00487889e-17]
 [-8.17791566e-17  1.00000000e+00  2.97408685e-16]
 [-4.00487889e-17  2.97408685e-16  1.00000000e+00]]
V*V^T:
[[ 1.00000000e+00 -1.35722246e-16 -8.36105969e-17]
 [-1.35722246e-16  1.00000000e+00 -1.65853766e-16]
 [-8.36105969e-17 -1.65853766e-16  1.00000000e+00]]
```

In the output we see the mentioned properties of orthogonal matrices.

1.6 Generate any 5x3 matrix by yourself and apply SVD

Generate any 5x3 matrix by yourself and apply SVD again (show the output). Also verify whether U and V are orthogonal matrices.

$$A = \begin{bmatrix} 5 & 2 & 8 \\ 8 & 0 & 1 \\ 1 & 1 & 5 \\ 3 & 4 & 2 \\ 8 & 6 & 5 \end{bmatrix}$$

Matrix is generated and SVD applied by following code. Very similar to the previous code.

Code:

```python
import numpy as np

rng = np.random.default_rng(seed=12)
size = (5, 3)
A = rng.integers(low=0, high=9, size=size)
print(A)

svd = np.linalg.svd(A, full_matrices=True, compute_uv=True)

print("U:")
print(svd[0])
print("S:", svd[1])
print("V^T:")
print(svd[2])
print("Reconstructed A=U*S*V^T:")
sigma = np.zeros(size)
sigma[:3, :3] = np.diag(svd[1])
print(svd[0] @ sigma @ svd[2])
print("Wrongly reconstructed A=U*S*V:")
print(svd[0] @ sigma @ svd[2].T)

# Orthogonal check
print("Determinate of U:", np.linalg.det(svd[0]))
print("Determinate of V:", np.linalg.det(svd[2]))
print("Inverse of U:")
print(np.linalg.inv(svd[0]))
print("Transpose of U:")
print(svd[0].T)
print("Inverse of V:")
print(np.linalg.inv(svd[2].T))
print("Transpose of V:")
print(svd[2])
print("U*U^T:")
print(svd[0] @ svd[0].T)
print("V*V^T:")
print(svd[2].T @ svd[2])
```

Output:

```
[[5 2 8]
 [8 0 1]
 [1 1 5]
 [3 4 2]
 [8 6 5]]
U:
[[-0.53752297 -0.47168307 -0.40732658 -0.2168383  -0.52502413]
 [-0.37826413  0.70105587 -0.53395389  0.2776286   0.05703084]
 [-0.23898187 -0.51049696 -0.18391354  0.47639527  0.6492333 ]
 [-0.28930532  0.02602911  0.54081897  0.66697473 -0.42223745]
 [-0.65358657  0.15732476  0.4718781  -0.4517696   0.34829413]]
S: [16.74903071  6.29984624  4.33380982]
V^T:
[[-0.71940391 -0.38167991 -0.58032616]
 [ 0.64703486 -0.06441396 -0.75973465]
```

```
 [-0.25259434  0.92204734 -0.29329986]]
Reconstructed A=U*S*V^T:
[[ 5.00000000e+00  2.00000000e+00  8.00000000e+00]
 [ 8.00000000e+00 -4.09447413e-15  1.00000000e+00]
 [ 1.00000000e+00  1.00000000e+00  5.00000000e+00]
 [ 3.00000000e+00  4.00000000e+00  2.00000000e+00]
 [ 8.00000000e+00  6.00000000e+00  5.00000000e+00]]
Wrongly reconstructed A=U*S*V:
[[ 8.63539474 -4.29269811  0.05196712]
 [ 4.2150251  -2.62574618  6.35130069]
 [ 4.56961798 -1.7771935  -1.72051585]
 [ 2.06317196 -4.92649519  0.68772567]
 [ 6.31019696 -8.70057497  3.07918975]]
Determinate of U: 1.0000000000000009
Determinate of V: -1.0000000000000007
Inverse of U:
[[-0.53752297 -0.37826413 -0.23898187 -0.28930532 -0.65358657]
 [-0.47168307  0.70105587 -0.51049696  0.02602911  0.15732476]
 [-0.40732658 -0.53395389 -0.18391354  0.54081897  0.4718781 ]
 [-0.2168383   0.2776286   0.47639527  0.66697473 -0.4517696 ]
 [-0.52502413  0.05703084  0.6492333  -0.42223745  0.34829413]]
Transpose of U:
[[-0.53752297 -0.37826413 -0.23898187 -0.28930532 -0.65358657]
 [-0.47168307  0.70105587 -0.51049696  0.02602911  0.15732476]
 [-0.40732658 -0.53395389 -0.18391354  0.54081897  0.4718781 ]
 [-0.2168383   0.2776286   0.47639527  0.66697473 -0.4517696 ]
 [-0.52502413  0.05703084  0.6492333  -0.42223745  0.34829413]]
Inverse of V:
[[-0.71940391 -0.38167991 -0.58032616]
 [ 0.64703486 -0.06441396 -0.75973465]
 [-0.25259434  0.92204734 -0.29329986]]
Transpose of V:
[[-0.71940391 -0.38167991 -0.58032616]
 [ 0.64703486 -0.06441396 -0.75973465]
 [-0.25259434  0.92204734 -0.29329986]]
U*U^T:
[[ 1.00000000e+00 -1.37148310e-17  1.82430599e-16  2.75101264e-17
   1.33219413e-16]
 [-1.37148310e-17  1.00000000e+00 -7.05932683e-17 -1.18503828e-16
  -1.82722049e-16]
 [ 1.82430599e-16 -7.05932683e-17  1.00000000e+00 -5.62264413e-17
   7.25903976e-17]
 [ 2.75101264e-17 -1.18503828e-16 -5.62264413e-17  1.00000000e+00
   1.76709107e-16]
 [ 1.33219413e-16 -1.82722049e-16  7.25903976e-17  1.76709107e-16
   1.00000000e+00]]
V*V^T:
[[ 1.00000000e+00 -6.23374040e-17 -7.86918687e-17]
 [-6.23374040e-17  1.00000000e+00  3.08669914e-16]
 [-7.86918687e-17  3.08669914e-16  1.00000000e+00]]
```

1.7 Generate any 3x4 matrix by yourself and apply SVD

Generate any 3x4 matrix by yourself and apply SVD again (show the output). Also verify whether U and V are orthogonal matrices.

$$A = \begin{bmatrix} 5 & 2 & 8 & 8 \\ 0 & 1 & 1 & 3 \\ 5 & 5 & 4 & 2 \end{bmatrix}$$

Same code as above with generated matrix size changed to (3, 4).

Output:

```
[[5 2 8 8]
 [0 1 1 1]
 [5 3 4 2]]
U:
[[ 0.87493648  0.46944757  0.11876505]
 [ 0.10064354  0.06361422 -0.99288676]
 [ 0.47366342 -0.88066578 -0.00841154]]
S: [14.20699394  3.35233698  0.96081213]
V^T:
[[ 0.47462535  0.23027438  0.63312402  0.5664437 ]
 [-0.61333065 -0.48905822  0.08845522  0.6138593 ]
 [ 0.57427204 -0.81242863 -0.07952912 -0.06201989]
 [ 0.26223636  0.2185303  -0.76485605  0.54632575]]
Reconstructed A=U*S*V^T:
[[ 5.00000000e+00  2.00000000e+00  8.00000000e+00  8.00000000e+00]
 [-4.30017407e-17  1.00000000e+00  1.00000000e+00  1.00000000e+00]
 [ 5.00000000e+00  3.00000000e+00  4.00000000e+00  2.00000000e+00]]
Wrongly reconstructed A=U*S*V:
[[ 6.33433608 -8.38339316  5.85069446  3.5162878 ]
 [ 0.12376062 -1.06564503  0.72373182  1.15121511]
 [ 2.50895897 -2.68418036  6.26363442  1.1256929 ]]
Determinate of U: -0.9999999999999997
Determinate of V: 1.0000000000000004
Inverse of U:
[[ 0.87493648  0.10064354  0.47366342]
 [ 0.46944757  0.06361422 -0.88066578]
 [ 0.11876505 -0.99288676 -0.00841154]]
Transpose of U:
[[ 0.87493648  0.10064354  0.47366342]
 [ 0.46944757  0.06361422 -0.88066578]
 [ 0.11876505 -0.99288676 -0.00841154]]
Inverse of V:
[[ 0.47462535  0.23027438  0.63312402  0.5664437 ]
 [-0.61333065 -0.48905822  0.08845522  0.6138593 ]
 [ 0.57427204 -0.81242863 -0.07952912 -0.06201989]
 [ 0.26223636  0.2185303  -0.76485605  0.54632575]]
Transpose of V:
[[ 0.47462535  0.23027438  0.63312402  0.5664437 ]
 [-0.61333065 -0.48905822  0.08845522  0.6138593 ]
 [ 0.57427204 -0.81242863 -0.07952912 -0.06201989]
 [ 0.26223636  0.2185303  -0.76485605  0.54632575]]
U*U^T:
[[ 1.00000000e+00 -6.35690649e-17 -2.48606937e-16]
 [-6.35690649e-17  1.00000000e+00  1.35309394e-16]
 [-2.48606937e-16  1.35309394e-16  1.00000000e+00]]
```

```
V*V^T:
[[ 1.00000000e+00 -1.20443304e-16  1.53724894e-16  2.24815993e-16]
 [-1.20443304e-16  1.00000000e+00 -1.97172578e-18  1.93686988e-16]
 [ 1.53724894e-16 -1.97172578e-18  1.00000000e+00  1.73007898e-16]
 [ 2.24815993e-16  1.93686988e-16  1.73007898e-16  1.00000000e+00]]
```

2 QR

2.1 Briefly describe QR decomposition: its input, outputs, basic properties, and applications.

QR decomposition decomposes a matrix into an orthogonal matrix Q and upper triangular matrix R. [4]

Input: A linearly independent matrix A

Output: The input matrix is decomposied into a orthogonal Q and upper triangular R matrix.

$$A = QR$$

Applications: It can be used to estimate linear regression. The OLS formular can be simplified with QR decomposition.

2.2 Briefly describe RQ decomposition.

Briefly describe RQ decomposition. In many cases, RQ decomposition is not available in libraries. Then, address how we can modify QR decomposition to RQ decomposition.

RQ decomposition is the same as QR decomposition but the order of the output matrices is reversed. [2] Due to matrix multiplication not being commutative R and Q have to be modified that the order of multiplication can be changed. By reversing the input matrix and flipping the output of QR decomposition, RQ decomposition can be made from QR decomposition.

2.3 Find the QR decomposition output of the following matrix

Find the QR decomposition output of the following matrix, using any libraries you prefer (e.g., Matlab or python). Verify Q is an orthogonal matrix in your output.:

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Code:

```python
import numpy as np

A = np.array([[1, 1, 0], [1, 0, 1], [0, 1, 1]])

q, r = np.linalg.qr(A)
print("Q:", q, "R:", r, "Q*R:", q @ r, sep="\n")

# Orthogonal check
print("Determinate of Q = 1:", np.allclose(np.linalg.det(q), 1))
print("Q Inverse = Q Transposed:", np.allclose(np.linalg.inv(q), q.T))
print("Q*Q^T=I:", np.allclose(q @ q.T, np.eye(3)))
```

Output:

```
Q:
[[-0.70710678  0.40824829 -0.57735027]
  [-0.70710678 -0.40824829  0.57735027]
  [-0.          0.81649658  0.57735027]]
R:
[[-1.41421356 -0.70710678 -0.70710678]
  [ 0.          1.22474487  0.40824829]
  [ 0.          0.          1.15470054]]
Q*R:
[[ 1.00000000e+00  1.00000000e+00 -3.09999437e-16]
  [ 1.00000000e+00 -1.10848874e-16  1.00000000e+00]
  [ 0.00000000e+00  1.00000000e+00  1.00000000e+00]]
Determinate of Q = 1: True
Inverse = Transpose Q: True
Q*Q^T=I: True
```

Q is orthogonal due to the multiple factors being true. Determinate is 1, inverse is the transpose and $Q^TQ = I$.

3 Cholesky

3.1 Briefly describe Cholesky decomposition, its input, outputs, and basic properties.

Cholesky decomposition is used to decompose a symmetric positive definite matrix into a lower triangular matrix and its transpose. [1]

Input: Symmetric positive definite matrix.

Output: One unique lower triangular matrix.

$$A = LL^T$$

By multiplying the lower triangular matrix with its transpose. The original matrix can be reconstructed.

Properties:

Applications:

- Can be used to solve systems of linear equations.
- Metric Rectification from Already Affine Rectified Images. [3]

3.2 Find the Cholesky decomposition output of the following matrix

Find the Cholesky decomposition output of the following matrix, using any libraries you prefer (e.g., Matlab or python):

$$A = \begin{bmatrix} 1 & -1 & 2 \\ -1 & 5 & -4 \\ 2 & -4 & 6 \end{bmatrix}$$

The Cholesky decomposition was done in Python with the scipy library.

Code:

```
import numpy as np
from scipy.linalg import cholesky

A = np.array([[1, -1, 2], [-1, 5, -4], [2, -4, 6]])
l, l_t  = cholesky(A)
print("L:")
print(l)
print("L*:")
print(l_t)
```

Output:

```
L:
[[ 1. -1.  2.]
 [ 0.  2. -1.]
 [ 0.  0.  1.]]
```

4 LU

4.1 Briefly describe LU decomposition, its input, outputs, and basic properties.

LU decomposition is used to split the matrix into a upper and lower triangular matrix.

Input: A matrix which has a non zero determinant.

Output: An lower triangular matrix L and upper triangular matrix U

$$A = LU$$

Applications: It can be used to solve systems of linear equations.

4.2 Find the LU decomposition output of the following matrix

Find the LU decomposition output of the following matrix, using any libraries you prefer (e.g., Matlab or python):

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 4 & -1 & 3 \\ -2 & 5 & 5 \end{bmatrix}$$

The LU decomposition was done in Python with the scipy library.

Code:

```
import numpy as np
from scipy.linalg import lu

A = np.array([[2, 1, 3], [4, -1, 3], [-2, 5, 5]])
p, l, u  = lu(A)
print("P:")
print(p)
```

```
print("L:")
print(l)
print("U:")
print(u)
```

Output:

```
P:
[[0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]
L:
[[ 1.          0.          0.        ]
 [-0.5         1.          0.        ]
 [ 0.5         0.33333333  1.        ]]
U:
[[ 4.         -1.          3.        ]
 [ 0.          4.5         6.5       ]
 [ 0.          0.         -0.66666667]]
```

5 Ax=0

5.1 Describe what Null space means given a Matrix A.

The null space of a matrix A contains all the solutions to the homogenous linear system of $Ax = 0$

What happen when a matrix A is full-rank? A full-rank matrix is invertible and has only the trivial solution of 0.

When does a matrix A have the non-trivial null space? The null space is non-trivial when the matrix is not invertible.

5.2 Describe an algorithm to solve the homogeneous linear system Ax = 0 via SVD, given A.

x can be any vector of the null space of A. This means that for any singular value $\sigma_i = 0$ the corresponding column in V is in the null space.

5.3 Find a non-trivial solution of Ax = 0

Find a non-trivial solution ($x \neq 0$) of Ax = 0 (submit both code and outputs) via SVD

$$A = \begin{bmatrix} 3.0975 & -1.9844 & 3.9311 \\ 14.0057 & 7.9991 & -17.9960 \\ 3.9054 & 2.0152 & -5.0669 \\ 1.9940 & 78.0010 & -1.0043 \end{bmatrix}$$

Code: [5]

```
import numpy as np

A = np.array(
    [
```

```
        [3.0975, -1.9844, 3.9311],
        [14.0057, 7.9991, -17.9960],
        [3.9054, 2.0152, -5.0669],
        [1.9940, 78.0010, -1.0043],
    ]
)

u, s, v = np.linalg.svd(A, full_matrices=True, compute_uv=True)

print("U:", u, "S:", s, "V:", v, sep="\n")
```

Output:

```
U:
[[ 0.02550768 -0.04924988  0.99844633 -0.00536027]
 [-0.11985998  0.9539549   0.05156734  0.2700823 ]
 [-0.03071587  0.26825396  0.00884774 -0.96281777]
 [-0.99198764 -0.12483716  0.01916901 -0.0029587 ]]
S:
[78.60837552 23.32634764  4.87024249]
V:
[[-0.04703944 -0.99795111  0.04336893]
 [ 0.60047842 -0.06294709 -0.79715954]
 [ 0.7982562  -0.01145583  0.6022091 ]]
```

There is no non-trivial solution to the system $Ax = 0$. Due to no singular value being 0.

6 Ax=b

6.1 Describe how to compute the pseudo inverse of a matrix A using SVD

THe pseudo inverse can be computed with SVD by using the formula:

$$A^+ = V\Sigma^+ U^T$$

Where Σ^+ is the pseudo inverse of Σ.

6.2 Find the solution of the following by computing the pseudo inverse of A

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 6 & 0 & 3 \\ 2 & 1 & 2 \\ 1 & 8 & 0 \end{bmatrix}, b = \begin{bmatrix} 8.2593 \\ 17.9659 \\ 15.9216 \\ 3.1024 \end{bmatrix}$$

Code:

```
import numpy as np

A = np.array([[1, 1, 1], [6, 0, 3], [2, 1, 2], [1, 8, 0]])
b = np.array([8.2593, 17.9659, 15.9216, 3.1024])

svd = np.linalg.svd(A, full_matrices=True, compute_uv=True)

print("U:", svd[0], "S:", svd[1], "V:", svd[2], sep="\n")
```

```
sigma = np.zeros((3, 4))
sigma[:3, :3] = np.diag([1/s if s != 0 else 0 for s in svd[1]])
a_pinv = svd[2].T @ sigma @ svd[0].T

print("A_pinv:", a_pinv, sep="\n")

x = a_pinv @ b + (np.eye(3) - a_pinv @ A) @ np.array([0, 0, 0])
print("x:", x)

print("Ax=b:", np.allclose(A @ x, b))
```

Output:

```
U:
[[-0.18173108  0.09355165 -0.42365008 -0.88246389]
 [-0.41210115  0.82127826  0.39417669 -0.01730321]
 [-0.26375701  0.26244507 -0.80205445  0.46718676]
 [-0.85298224 -0.49786815  0.14783098  0.05190964]]
S:
[8.53864378 6.93829543 0.9755096 ]
V:
[[-0.47253807 -0.85134667 -0.22785218]
 [ 0.72759127 -0.52274345  0.44424118]
 [ 0.49731149 -0.04413762 -0.86644859]]
A_pinv:
[[-0.19610778  0.30988024 -0.36676647  0.07035928]
 [ 0.03023952 -0.03862275  0.04281437  0.11586826]
 [ 0.38712575 -0.28652695  0.73622754 -0.14041916]]
x: [-1.67366198  0.59700772  9.33595731]
Ax=b: True
```

7 Levenberg-Marquardt

Briefly describe Levenberg-Marquardt (LM) algorithm. How the LM method differs from the gradient descent and Gauss-Newton algorithm.

Levenberg-Marquardt is an interative optimization algorithm used to minimize a function. [6] It is used in non-linear least squares problems.

The Levenberg-Marquardt algorithm differs from the gradient descent and Gauss-Newton algorithm by combining the two. It uses a damping factor to control the step size.

References

[1] *Cholesky decomposition*. Wikipedia. URL: https://en.wikipedia.org/wiki/Cholesky_decomposition (visited on 03/23/2025).

[2] johnnycrab. *RQ decomposition*. URL: https://math.stackexchange.com/questions/1640695/rq-decomposition (visited on 03/23/2025).

[3] Hanbyul Joo. *Lecture 5: Homography Estimation*. (Visited on 03/23/2025).

[4] *QR decomposition*. StatLect. URL: https://www.statlect.com/matrix-algebra/QR-decomposition (visited on 03/23/2025).

[5] *Singular Value Decomposition*. Rensselaer Polytechnic Institute. URL: https://sites.ecse.rpi.edu/~qji/CV/svd_review.pdf (visited on 03/23/2025).

[6] *Wikipedia*. Levenberg-Marquardt algorithm. URL: https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm (visited on 03/23/2025).