Computer Vision Practice Exercises

Instructor: PhD Thái Đình Kim
Computer Vision, Spring 2025
VNU-IS March 16, 2025

Computer Vision Practice Exercise
`https://vnuis.edu.vn/`

This document provides programming exercises for students to practice computer vision using Python and OpenCV.

Mục lục

General Instructions

- **Objective:** These exercises cover various core topics in computer vision, enabling students to explore, implement, and evaluate essential techniques using Python and OpenCV. Through hands-on practice, students develop a deeper understanding of image processing, feature extraction, geometric transformations, and visual recognition.
- **Tools:** Use Python 3.x, OpenCV (`cv2`), NumPy, and Matplotlib. Install via:

```
pip install opencv-python numpy matplotlib
```

- **Submission:** Submit Python code and a report with input/output images and brief explanations for each exercise.

1 Lecture 7: 2D Transformations

This lab session aims to help students understand and implement fundamental 2D geometric transformations in image processing, including translation, rotation, scaling, shearing, and affine transformations. Through a series of programming exercises, students will apply mathematical formulations of these transformations using homogeneous coordinates and OpenCV functions. The lab also includes estimating affine transformations from both minimal and redundant point correspondences using the least squares method. These exercises enhance students' practical understanding of how geometric operations are applied to images in computer vision.

Programming Exercises

Exercise 1: Translation

- **Objective:** Translate an image 100 pixels to the right and 50 pixels downward.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The translated image, with cropped areas filled with black.
- **Algorithm Applied:** Translation is represented by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where $t_x = 100$, $t_y = 50$. OpenCV uses a 2×3 matrix with `cv2.warpAffine`.

Exercise 2: Rotation

- **Objective:** Rotate an image by 45 degrees around its center.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The rotated image, with cropped areas filled with black.
- **Algorithm Applied:** Rotation is represented by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  where $\theta = 45^\circ$. OpenCV's `cv2.getRotationMatrix2D` generates a 2×3 matrix for `cv2.warpAffine`.

Exercise 3: Scaling

- **Objective:** Scale an image up by a factor of 1.5 in both x and y directions.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The scaled image, with cropped areas filled with black.
- **Algorithm Applied:** Scaling is represented by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  where $s_x = s_y = 1.5$. OpenCV uses a 2×3 matrix with `cv2.warpAffine`.

Exercise 4: Shearing

- **Objective:** Apply a shearing transformation along the x-axis with a shear factor of 0.5.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The sheared image, with cropped areas filled with black.
- **Algorithm Applied:** Shearing is represented by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  where $s = 0.5$. OpenCV uses a 2×3 matrix with `cv2.warpAffine`.

Exercise 5: General Affine Transformation

- **Objective:** Apply an affine transformation based on 3 point correspondences.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The affine-transformed image, with cropped areas filled with black.
- **Algorithm Applied:** Affine transformation is:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  Use `cv2.getAffineTransform` with 3 point pairs, then apply with `cv2.warpAffine`.

Exercise 6: Affine Transformation with Least Squares

- **Objective:** Determine an affine transformation from multiple point correspondences using Least Squares.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The affine-transformed image, with cropped areas filled with black.
- **Algorithm Applied:** Solve:

$$x' = ax + by + t_x, \quad y' = cx + dy + t_y$$

  Construct $Ax = b$, where A is $2n \times 6$, $x = [a, b, t_x, c, d, t_y]$, and solve using `np.linalg.lstsq`.

2 Lecture 8: Image Homographies

This lab session focuses on understanding and applying image homographies, which are fundamental for tasks such as image warping, perspective correction, and panorama stitching. Students will learn how to apply a homography transformation manually, compute it using the Direct Linear Transform (DLT) algorithm from point correspondences, and refine it using RANSAC to handle outliers. The lab also covers automatic feature detection and matching using SIFT, which supports robust homography estimation. Finally, students will integrate these techniques to stitch images into a single panoramic view, laying the foundation for more advanced computer vision applications.

Programming Exercises

Exercise 1: Applying a Homography Transformation

- **Objective:** Apply a manually defined homography matrix to warp an image.
- **Input Image:** A single image (e.g., `lena.jpg`).
- **Expected Output:** The warped image, with cropped areas filled with black.
- **Algorithm Applied:** Homography is:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

  Apply using `cv2.warpPerspective`.

Exercise 2: Computing Homography with DLT

- **Objective:** Compute a homography matrix H using DLT from 4+ point correspondences.
- **Input Images:** Two overlapping images.
- **Expected Output:** The second image warped onto the first's perspective.
- **Algorithm Applied:** Solve $Ah = 0$, where:

$$A_i = \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix}$$

  Use SVD, reshape h to H, and apply with `cv2.warpPerspective`.

Exercise 3: Feature Detection and Matching

- **Objective:** Detect and match feature points between two images using SIFT.
- **Input Images:** Two overlapping images.
- **Expected Output:** Visualization of matched feature points.
- **Algorithm Applied:** Use `cv2.SIFT_create()`, `cv2.BFMatcher`, and `cv2.drawMatches` with a ratio test.

Exercise 4: Estimating Homography with RANSAC

- **Objective:** Estimate a homography matrix H using RANSAC.
- **Input Images:** Two overlapping images.
- **Expected Output:** The second image warped onto the first's perspective.
- **Algorithm Applied:** Use `cv2.findHomography` with `cv2.RANSAC`, then apply with `cv2.warpPerspective`.

Exercise 5: Creating a Panorama

- **Objective:** Stitch two images into a panorama using homography and RANSAC.
- **Input Images:** Two overlapping images.
- **Expected Output:** A panoramic image.
- **Algorithm Applied:** Detect features (SIFT), estimate H (RANSAC), warp with `cv2.warpPerspective`, and blend images.

Extensions

- Experiment with different transformation parameters or feature detectors (e.g., ORB).
- Extend panorama stitching to multiple images.
- Implement advanced blending techniques (e.g., multi-band blending).

3 Lecture 9: Geometric Camera Models

These exercises cover geometric camera models from Lecture 9, with a focus on pinhole camera models, camera matrices, and coordinate transformations. The goal is to help students implement these concepts using Python and NumPy. The input data can be either synthetic 3D points or real-world image data, such as a checkerboard image, for testing purposes. The expected output includes 2D image coordinates obtained from 3D points, along with visualizations of projections or transformed images. Students are required to submit Python code along with a report that includes input/output results and brief explanations for each exercise.

Programming Exercises

Exercise 1: Simple Pinhole Projection

- **Objective:** Project a 3D point to a 2D image plane using a pinhole camera model.
- **Input Data:** A 3D point (X, Y, Z) and focal length f.
- **Expected Output:** 2D image coordinates (x, y).
- **Algorithm Applied:** Use the pinhole projection equations:

$$x = f \cdot \frac{X}{Z}, \quad y = f \cdot \frac{Y}{Z}$$

  Implement in Python using NumPy and test with sample points (e.g., $(1, 1, 2)$, $f = 1$).

Exercise 2: Intrinsic Camera Matrix

- **Objective:** Construct an intrinsic camera matrix K and project a 3D point to 2D.
- **Input Data:** Focal length f, principal point (p_x, p_y), and a 3D point (X, Y, Z).
- **Expected Output:** 2D image coordinates (x, y).
- **Algorithm Applied:** Construct K as:

$$K = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

  Project using $\mathbf{x} = K \cdot [X, Y, Z]^T$, then normalize homogeneous coordinates.

Exercise 3: Extrinsic Parameters and Full Camera Matrix

- **Objective:** Combine intrinsic and extrinsic parameters to project a 3D world point to 2D.
- **Input Data:** Intrinsic matrix K, rotation matrix R, translation vector t, and a 3D point (X_w, Y_w, Z_w).
- **Expected Output:** 2D image coordinates (x, y).
- **Algorithm Applied:** Form the camera matrix $P = K[R|t]$, where:

  $$[R|t] = \begin{bmatrix} R_{3\times 3} & t_{3\times 1} \end{bmatrix}$$

  Project using $\mathbf{x} = P \cdot [X_w, Y_w, Z_w, 1]^T$, then normalize.

Exercise 4: Effect of Focal Length and Pinhole Size

- **Objective:** Explore the effect of varying focal length and simulating pinhole size on projections.
- **Input Data:** A set of 3D points (e.g., a 3D grid) and varying f values (e.g., 0.5, 1.0, 2.0).
- **Expected Output:** Visualizations of 2D projections for different f values using Matplotlib.
- **Algorithm Applied:** Project points with different f:

  $$x = f \cdot \frac{X}{Z}, \quad y = f \cdot \frac{Y}{Z}$$

  Simulate larger pinhole size by adding random noise or averaging nearby projections.

Exercise 5: Real-World Application - Projecting 3D Points onto an Image

- **Objective:** Project a set of 3D points onto a real image using a camera matrix.
- **Input Data:** A camera matrix P, a set of 3D points, and an image (e.g., `checkerboard.jpg`).
- **Expected Output:** The image with projected 2D points marked (using OpenCV).
- **Algorithm Applied:** Compute $\mathbf{x} = P \cdot [X, Y, Z, 1]^T$, normalize to get (x, y), and draw points with `cv2.circle`.

Extensions

- Experiment with different rotation matrices R or translation vectors t and observe their effects.
- Implement camera calibration using OpenCV's `cv2.calibrateCamera` with a checkerboard pattern.
- Simulate lens distortion effects and correct them using additional parameters in the camera matrix.

4 Lecture 10: Geometric Camera Models (cont.)

These exercises cover geometric camera models (Lecture 10), focusing on reviewing the camera matrix, understanding perspective distortion, exploring alternative camera models, and performing pose estimation. Students will implement these concepts using Python, NumPy, and OpenCV. The input data may include synthetic 3D points, 2D-3D correspondences, or real-world images such as checkerboard patterns. The expected output consists of computed camera matrices, projected points, visualizations, or pose estimates. Students are required to submit their Python code along with a report that includes input/output results and brief explanations for each exercise.

Programming Exercises

Exercise 1: Perspective Projection with Different Focal Lengths

- **Objective:** Simulate perspective projection with varying focal lengths to observe magnification effects.
- **Input Data:** A set of 3D points (e.g., a 3D grid) and focal lengths $f = [100, 500, 1000]$.
- **Expected Output:** 2D projections visualized using Matplotlib for each focal length.
- **Algorithm Applied:** Use the perspective projection:

$$x = f \cdot \frac{X}{Z}, \quad y = f \cdot \frac{Y}{Z}$$

  Implement in Python with NumPy and plot the results to demonstrate perspective distortion.

Exercise 2: Weak Perspective vs. Perspective Projection

- **Objective:** Compare perspective and weak perspective projections for a set of 3D points.
- **Input Data:** A set of 3D points at varying depths and a fixed focal length f.
- **Expected Output:** 2D projections for both models, visualized side-by-side.
- **Algorithm Applied:** - Perspective: $x = f \cdot \frac{X}{Z}, y = f \cdot \frac{Y}{Z}$ - Weak Perspective: Assume constant depth Z_{avg}, then $x = f \cdot \frac{X}{Z_{\text{avg}}}, y = f \cdot \frac{Y}{Z_{\text{avg}}}$ Implement in Python and visualize differences.

Exercise 3: Orthographic Projection Simulation

- **Objective:** Simulate an orthographic projection and compare it with perspective projection.
- **Input Data:** A set of 3D points and an orthographic camera matrix.
- **Expected Output:** 2D projections visualized for orthographic and perspective models.
- **Algorithm Applied:** Orthographic projection matrix (assuming no scaling or shift):

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

  Project points with $\mathbf{x} = P \cdot [X, Y, Z, 1]^T$ and compare with perspective projection.

Exercise 4: Pose Estimation with DLT

- **Objective:** Estimate a camera matrix P using Direct Linear Transform (DLT) from 3D-2D correspondences.
- **Input Data:** At least 6 pairs of 3D points (X_i, Y_i, Z_i) and 2D image points (x_i, y_i).
- **Expected Output:** Estimated 3×4 camera matrix P.
- **Algorithm Applied:** For each point correspondence, set up:

$$\begin{bmatrix} \mathbf{X}_i^\top & \mathbf{0}^\top & -x_i \mathbf{X}_i^\top \\ \mathbf{0}^\top & \mathbf{X}_i^\top & -y_i \mathbf{X}_i^\top \end{bmatrix} \mathbf{p} = \mathbf{0},$$

  where $\mathbf{X}_i = [X_i, Y_i, Z_i, 1]^\top$, $\mathbf{p} = [p_1, \ldots, p_{12}]^\top$, and $\mathbf{0} = [0, 0, 0, 0]^\top$. Solve $A\mathbf{p} = 0$ using SVD, taking the smallest singular vector and reshaping it into P.

Exercise 5: Camera Matrix Decomposition

- **Objective:** Decompose a camera matrix P into intrinsic K, rotation R, and translation t.
- **Input Data:** A 3×4 camera matrix P (e.g., from Exercise 4 or synthetic data).
- **Expected Output:** Matrices K, R, and vector t.
- **Algorithm Applied:** - Extract $M = P[:, 0 : 3]$ and $t = -M^{-1}P[:, 3]$ (camera center). - Use QR decomposition on M^{-1} to get K (upper triangular) and R (orthogonal). Implement in Python with NumPy's `linalg.svd` and `linalg.qr`.

Exercise 6: Pose Estimation with Real Data

- **Objective:** Estimate camera pose using OpenCV from a checkerboard image.
- **Input Data:** An image of a checkerboard and known 3D coordinates of its corners.
- **Expected Output:** Rotation vector, translation vector, and visualized reprojection.
- **Algorithm Applied:** Use `cv2.findChessboardCorners` to detect 2D points, define corresponding 3D points, and solve with `cv2.solvePnP`. Visualize reprojected points with `cv2.projectPoints`.

Extensions

- Experiment with radial distortion correction using OpenCV's `cv2.undistort`.
- Implement a dolly zoom (vertigo effect) simulation by adjusting focal length and camera position.
- Extend pose estimation to multiple frames for camera tracking.

General Instructions

- **Objective:** These exercises cover two-view geometry (Lecture 11), including triangulation, epipolar geometry, essential matrix, fundamental matrix, and the 8-point algorithm, helping students implement these concepts using Python, NumPy, and OpenCV.
- **Tools:** Use Python 3.x, NumPy, OpenCV (`cv2`), and Matplotlib for visualization. Install via:

```
pip install numpy opencv-python matplotlib
```

- **Input Data:** Use pairs of images with known 2D point correspondences or synthetic data (e.g., 3D points and camera matrices).
- **Expected Output:** 3D point reconstructions, epipolar lines, essential/fundamental matrices, and visualizations.
- **Submission:** Submit Python code and a report with input/output results and brief explanations for each exercise.

5 Lecture 11: Two-view Geometry

The exercises in the list are designed to achieve the following specific objectives: computing the 3D coordinates of a point in space from 2D projections using backprojection and SVD (Triangulation); visualizing epipolar geometry by drawing epipolar lines using the fundamental matrix (Epipolar Line Visualization); estimating the essential matrix from 2D point correspondences and camera intrinsics using the normalized 8-point algorithm (Essential Matrix Estimation); computing the fundamental matrix from at least 8 pairs of matched points using the 8-point algorithm with rank-2 constraint (Fundamental Matrix); finding the epipole through the kernel space of the fundamental matrix using SVD (Epipole Computation); and finally, applying the complete real-world two-view geometry pipeline including feature detection and matching, fundamental matrix estimation, epipolar line visualization, and triangulation for 3D structure reconstruction (Real-world Two-view Geometry).

Programming Exercises

Exercise 1: Triangulation with Backprojection

- **Objective:** Compute the 3D point from two 2D correspondences using triangulation.
- **Input Data:** Two camera matrices P and P', and 2D points $\mathbf{x}$ and $\mathbf{x}'$.
- **Expected Output:** 3D point coordinates $\mathbf{X}$.
- **Algorithm Applied:** Set up the system $A\mathbf{X} = 0$ using cross products:

$$\mathbf{x} \times (P\mathbf{X}) = 0, \quad \mathbf{x}' \times (P'\mathbf{X}) = 0$$

  Solve using SVD to find $\mathbf{X}$.

Exercise 2: Epipolar Line Visualization

- **Objective:** Compute and visualize epipolar lines given a fundamental matrix F.
- **Input Data:** Two images, a fundamental matrix F, and a set of 2D points in the first image.
- **Expected Output:** Second image with epipolar lines drawn.
- **Algorithm Applied:** Compute $\mathbf{l}' = F\mathbf{x}$ for each point $\mathbf{x}$, then plot the line $l'_1 x + l'_2 y + l'_3 = 0$.

Exercise 3: Essential Matrix Estimation

- **Objective:** Estimate the essential matrix E from known camera intrinsics and point correspondences.
- **Input Data:** Intrinsic matrices K and K', and 2D point correspondences $\mathbf{x}$ and $\mathbf{x}'$.
- **Expected Output:** Essential matrix E.
- **Algorithm Applied:** Normalize points ($\hat{\mathbf{x}} = K^{-1}\mathbf{x}$), apply the 8-point algorithm to compute F, then compute $E = K'^T F K$.

Exercise 4: Fundamental Matrix with 8-Point Algorithm

- **Objective:** Compute the fundamental matrix F using the 8-point algorithm.
- **Input Data:** At least 8 pairs of 2D point correspondences $\{\mathbf{x}_m, \mathbf{x}'_m\}$.
- **Expected Output:** Fundamental matrix F.
- **Algorithm Applied:** Construct A where each row is $[x_m x'_m, x_m y'_m, x_m, y_m x'_m, y_m y'_m, y_m, x'_m, y'_m, 1]$, solve $A\mathbf{f} = 0$ using SVD, reshape $\mathbf{f}$ into F, and enforce rank 2.

Exercise 5: Epipole Computation

- **Objective:** Compute the epipole from a fundamental matrix F.
- **Input Data:** Fundamental matrix F.
- **Expected Output:** Epipole coordinates in homogeneous form.
- **Algorithm Applied:** Compute the right null space of F using SVD; the epipole $\mathbf{e}'$ is the singular vector corresponding to the smallest singular value.

Exercise 6: Real-world Two-view Geometry

- **Objective:** Apply two-view geometry to real images using feature matching and OpenCV.
- **Input Data:** Two overlapping images.
- **Expected Output:** Fundamental matrix F, epipolar lines, and triangulated 3D points visualized.
- **Algorithm Applied:** Detect and match features (e.g., SIFT), compute F with `cv2.findFundamentalMat`, draw epipolar lines, and triangulate points with `cv2.triangulatePoints`.

6 Lecture 12: Stereo

In Lecture 12 on Stereo Vision, students will engage in several programming exercises to gain a deeper understanding of how to reconstruct 3D information from images. Specifically, students will review the technique of computing 3D spatial coordinates from a stereo image pair using corresponding points and camera matrices. Next, they will compute the disparity between rectified image pairs and use this information to estimate object distances. The lab also includes stereo rectification using intrinsic and extrinsic camera parameters with OpenCV. Following this, students will generate disparity maps using a basic block-matching method. Finally, the session introduces an advanced stereo matching technique that enhances smoothness between neighboring pixels by applying a dynamic programming algorithm that balances accuracy and continuity.

Programming Exercises

Exercise 1: Revisiting Triangulation

- **Objective:** Reconstruct 3D points from stereo images using triangulation.
- **Input Data:** Left and right stereo images, camera matrices P and P', and matched 2D points.
- **Expected Output:** 3D point coordinates $\mathbf{X}$.
- **Algorithm Applied:** For each pair of points $\mathbf{x}$ and $\mathbf{x}'$, solve $A\mathbf{X} = 0$ using SVD, where A is formed from $\mathbf{x} \times (P\mathbf{X}) = 0$ and $\mathbf{x}' \times (P'\mathbf{X}) = 0$.

Exercise 2: Disparity Computation

- **Objective:** Compute disparity from rectified stereo images and estimate depth.
- **Input Data:** Rectified left and right stereo images, focal length f, and baseline b.
- **Expected Output:** Disparity map and depth map.
- **Algorithm Applied:** Compute disparity $d = x_l - x_r$ for each pixel, then depth $Z = \frac{f \cdot b}{d}$.

Exercise 3: Stereo Rectification

- **Objective:** Rectify a pair of stereo images to align epipolar lines horizontally.
- **Input Data:** Left and right stereo images, camera intrinsics K, K', and extrinsic parameters.
- **Expected Output:** Rectified left and right images.
- **Algorithm Applied:** Use `cv2.stereoRectify` to compute rectification homographies, then apply `cv2.warpPerspective` to rectify images.

Exercise 4: Stereo Matching with Block Matching

- **Objective:** Compute a disparity map using block matching.
- **Input Data:** Rectified left and right stereo images.
- **Expected Output:** Disparity map.
- **Algorithm Applied:** Use `cv2.StereoBM` with SSD cost function to compute disparity along horizontal epipolar lines.

Exercise 5: Improving Stereo Matching with Smoothness

- **Objective:** Improve stereo matching by enforcing smoothness using an energy minimization approach.
- **Input Data:** Rectified left and right stereo images.
- **Expected Output:** Enhanced disparity map.
- **Algorithm Applied:** Implement a simple dynamic programming approach per scanline with data term (SSD) and smoothness term (e.g., $|d_p - d_q|$).

Exercise 6: Structured Light Simulation

- **Objective:** Simulate structured light for depth estimation using a projector and camera.
- **Input Data:** A single image with a projected pattern (e.g., stripes) and camera/projector calibration data.
- **Expected Output:** Depth map.
- **Algorithm Applied:** Simulate stripe projection, detect pattern edges, and triangulate depth using known projector-camera geometry.

Extensions

- Experiment with different block sizes in stereo matching and analyze their effects.
- Implement normalized cross-correlation (NCC) instead of SSD for stereo matching.
- Use structured light with real hardware (e.g., a projector) and compare with stereo results.

References

- Lecture 7: 2D Transformations, VNU-IS, Spring 2025.
- Lecture 8: Image Homographies, VNU-IS, Spring 2025.
- Lecture 9: Geometric Camera Models, VNU-IS, Spring 2025.
- Lecture 10: Geometric Camera Models (cont.), VNU-IS, Spring 2025.
- Lecture 11: Two-view Geometry, VNU-IS, Spring 2025.
- Lecture 12: Stereo, VNU-IS, Spring 2025.
- OpenCV Documentation, `https://docs.opencv.org/`.
- Hartley, R., & Zisserman, A., "Multiple View Geometry in Computer Vision".