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

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 $\theta = 45°$. 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

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:

$$[R|t] = \begin{bmatrix} R_{3\times3} & t_{3\times1} \end{bmatrix}$$

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.

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.
- OpenCV Documentation, `https://docs.opencv.org/`.

- Hartley, R., & Zisserman, A., "Multiple View Geometry in Computer Vision".