Visión Artificial VIAR25

Lab 1: PyTorch Setup, Camera Calibration & Image Transformations

Prof. David Olivieri

Uvigo

September 8, 2025

Lab 1 Overview

Lab Goals

Transform mathematical theory from Lecture 1 into working code. By the end of this lab, you'll have implemented the core algorithms that power computer vision systems.

What We'll Implement Today

1. **Environment Setup**: PyTorch + OpenCV + visualization tools
2. **Image I/O & Manipulation**: Loading, transforming, visualizing images
3. **Camera Calibration**: Zhang's method with checkerboard detection
4. **Homography Estimation**: DLT algorithm with RANSAC
5. **Lens Distortion Correction**: Brown-Conrady model implementation
6. **Color Space Transformations**: RGB $\leftrightarrow$ HSV conversion

Lab Structure

- **Demo Code**: Step-by-step implementation with explanations
- **Exercises**: Hands-on coding challenges to test understanding
- **Questions**: Theoretical connections to deepen comprehension
- **Extension Challenges**: Advanced implementations for extra credit

Environment Setup & PyTorch Fundamentals

Required Libraries

```python
import torch
import torchvision
import torchvision.transforms as transforms
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import RANSACRegressor
import glob

```

Exercise 1.1: PyTorch Tensor Operations

Task: Create a function that converts between NumPy arrays and PyTorch tensors

```python
def numpy_to_torch(img_np):
    """Convert HxWxC numpy array to CxHxW torch tensor"""
    # TODO: Implement conversion
    pass

def torch_to_numpy(img_torch):
    """Convert CxHxW torch tensor to HxWxC numpy array"""
    # TODO: Implement conversion
    pass

```

Questions:

- Why do PyTorch and OpenCV use different channel ordering?
- When would you use GPU tensors vs CPU tensors for image processing?

Image I/O and Basic Transformations

Demo: Image Loading and Visualization

```python
def load_image(filepath):
    """Load image using OpenCV and convert to RGB"""
    img_bgr = cv2.imread(filepath)
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
    return img_rgb

def visualize_images(images, titles, figsize=(15, 5)):
    """Display multiple images in a row"""
    fig, axes = plt.subplots(1, len(images), figsize=figsize)
    for i, (img, title) in enumerate(zip(images, titles)):
        axes[i].imshow(img, cmap='gray' if len(img.shape)==2 else None)
        axes[i].set_title(title)
        axes[i].axis('off')
    plt.tight_layout()
    plt.show()

```

Exercise 1.2: Geometric Transformations

Task: Implement the transformation hierarchy from Lecture 1

```python
def apply_transformation(img, transform_matrix):
    """Apply 3x3 transformation matrix to image"""
    # TODO: Use cv2.warpPerspective
    pass

def create_similarity_transform(scale, rotation, translation):
    """Create similarity transformation matrix"""
    # TODO: Implement using lecture equations
    pass

```

Corner Detection for Calibration

Demo: Checkerboard Corner Detection

```python
def detect_checkerboard_corners(img, pattern_size):
    """Detect checkerboard corners with sub-pixel accuracy"""
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # Find corners
    ret, corners = cv2.findChessboardCorners(gray, pattern_size, None)

    if ret:
        # Refine corners to sub-pixel accuracy
        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
                    30, 0.001)
        corners = cv2.cornerSubPix(gray, corners, (11,11), (-1,-1), criteria)

    return ret, corners

def visualize_corners(img, corners, pattern_size):
    """Draw detected corners on image"""
    img_with_corners = img.copy()
    cv2.drawChessboardCorners(img_with_corners, pattern_size, corners, True)
    return img_with_corners

```

Exercise 1.3: Corner Detection Analysis

Questions:

- What happens if the checkerboard is blurry? Test with different image qualities.
- How does corner detection accuracy affect calibration results?
- Implement a function to measure corner detection repeatability across multiple views.

Zhang's Camera Calibration Algorithm

Demo: Complete Calibration Pipeline

```python
def calibrate_camera(calibration_images, pattern_size, square_size):
    """Implement Zhang's camera calibration method"""

    # Prepare object points (3D points in real world space)
    objp = np.zeros((pattern_size[0] * pattern_size[1], 3), np.float32)
    objp[:,:2] = np.mgrid[0:pattern_size[0], 0:pattern_size[1]].T.reshape(-1,2)
    objp *= square_size

    # Arrays to store object points and image points
    objpoints = []  # 3D points in real world space
    imgpoints = []  # 2D points in image plane

    for img_path in calibration_images:
        img = load_image(img_path)
        ret, corners = detect_checkerboard_corners(img, pattern_size)

        if ret:
            objpoints.append(objp)
            imgpoints.append(corners)

    # Perform calibration
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(
        objpoints, imgpoints, img.shape[:2][::-1], None, None)

    return ret, mtx, dist, rvecs, tvecs

```

Calibration Analysis and Validation

Exercise 1.4: Calibration Quality Assessment

Task: Implement calibration quality metrics

```python
def compute_reprojection_error(objpoints, imgpoints, mtx, dist, rvecs, tvecs):
    """Compute RMS reprojection error"""
    # TODO: Project 3D points back to image plane
    # TODO: Compute distance between projected and detected points
    pass

def analyze_calibration_accuracy(mtx, dist, rvecs, tvecs, objpoints, imgpoints):
    """Comprehensive calibration analysis"""
    # TODO: Compute per-image errors
    # TODO: Visualize error distribution
    # TODO: Check for systematic biases
    pass

```

Questions:

- What's a "good" reprojection error? How does it depend on image resolution?
- How many calibration images do you need for stable results?
- What happens if you use only images from one orientation?

Expected Results

- Reprojection error: < 1.0 pixels for good calibration
- Focal lengths should be approximately equal for square pixels
- Principal point should be near image center

Homography Estimation with DLT

Demo: Direct Linear Transform Implementation

```python
def estimate_homography_dlt(src_pts, dst_pts):
    """Estimate homography using Direct Linear Transform"""
    assert len(src_pts) >= 4, "Need at least 4 point correspondences"

    # Normalize points for numerical stability
    src_norm, T1 = normalize_points(src_pts)
    dst_norm, T2 = normalize_points(dst_pts)

    # Set up linear system Ah = 0
    A = []
    for i in range(len(src_norm)):
        x, y = src_norm[i]
        u, v = dst_norm[i]

        A.append([0, 0, 0, -x, -y, -1, v*x, v*y, v])
        A.append([x, y, 1, 0, 0, 0, -u*x, -u*y, -u])

    A = np.array(A)

    # Solve using SVD
    U, S, Vt = np.linalg.svd(A)
    h = Vt[-1, :] / Vt[-1, -1]  # Normalize by last element
    H_norm = h.reshape(3, 3)

    # Denormalize
    H = np.linalg.inv(T2) @ H_norm @ T1

    return H / H[2, 2]  # Normalize by bottom-right element

```

Point Normalization for Numerical Stability

Demo: Hartley Normalization

```python
def normalize_points(points):
    """Normalize points for DLT numerical stability"""
    points = np.array(points, dtype=np.float32)

    # Compute centroid
    centroid = np.mean(points, axis=0)

    # Compute mean distance to centroid
    distances = np.linalg.norm(points - centroid, axis=1)
    scale = np.sqrt(2) / np.mean(distances)

    # Create transformation matrix
    T = np.array([
        [scale, 0, -scale * centroid[0]],
        [0, scale, -scale * centroid[1]],
        [0, 0, 1]
    ])

    # Apply transformation
    points_hom = np.column_stack([points, np.ones(len(points))])
    points_norm = (T @ points_hom.T).T
    points_norm = points_norm[:, :2] / points_norm[:, [2]]

    return points_norm, T

```

Exercise 1.5: Homography Robustness

Task: Compare normalized vs. unnormalized DLT

- Implement DLT without normalization
- Test on points with different scales (e.g., [0,1] vs [0,1000])
- Measure conditioning of matrix $A^T A$

Question: Why does normalization improve numerical stability?

RANSAC for Robust Homography Estimation

Demo: RANSAC Implementation

```python
def ransac_homography(src_pts, dst_pts, threshold=3.0, max_iters=1000):
    """Robust homography estimation using RANSAC"""
    best_H = None
    best_inliers = []
    best_score = 0

    n_points = len(src_pts)

    for _ in range(max_iters):
        # Randomly sample 4 points
        sample_idx = np.random.choice(n_points, 4, replace=False)
        sample_src = src_pts[sample_idx]
        sample_dst = dst_pts[sample_idx]

        # Estimate homography from sample
        H = estimate_homography_dlt(sample_src, sample_dst)

        # Count inliers
        inliers = []
        for i in range(n_points):
            # Transform source point
            src_hom = np.append(src_pts[i], 1)
            dst_pred_hom = H @ src_hom
            dst_pred = dst_pred_hom[:2] / dst_pred_hom[2]

            # Compute reprojection error
            error = np.linalg.norm(dst_pred - dst_pts[i])
            if error < threshold:
                inliers.append(i)

        # Update best model if better
        if len(inliers) > best_score:
            best_score = len(inliers)
            best_inliers = inliers
            best_H = H

    return best_H, best_inliers

```

Lens Distortion Correction

Demo: Brown-Conrady Distortion Model

```python
def undistort_points(points, mtx, dist):
    """Undistort points using camera parameters"""
    # Convert to homogeneous coordinates
    points_hom = np.column_stack([points, np.ones(len(points))])

    # Apply inverse camera matrix to get normalized coordinates
    points_norm = np.linalg.inv(mtx) @ points_hom.T
    points_norm = points_norm[:2, :].T

    # Apply distortion correction
    k1, k2, p1, p2, k3 = dist.ravel()

    points_corrected = []
    for x, y in points_norm:
        r2 = x*x + y*y

        # Radial distortion correction
        radial_factor = 1 + k1*r2 + k2*r2*r2 + k3*r2*r2*r2

        # Tangential distortion correction
        x_corrected = x * radial_factor + 2*p1*x*y + p2*(r2 + 2*x*x)
        y_corrected = y * radial_factor + p1*(r2 + 2*y*y) + 2*p2*x*y

        points_corrected.append([x_corrected, y_corrected])

    # Convert back to pixel coordinates
    points_corrected = np.array(points_corrected)
    points_corrected_hom = np.column_stack([points_corrected, np.ones(len(points_corrected))])
    points_pixel = (mtx @ points_corrected_hom.T).T

    return points_pixel[:, :2]
```

Color Space Transformations

Demo: RGB to HSV Conversion

```python
def rgb_to_hsv_manual(rgb_img):
    """Manual RGB to HSV conversion following lecture algorithm"""
    rgb_normalized = rgb_img.astype(np.float32) / 255.0
    h, w, c = rgb_img.shape
    hsv_img = np.zeros_like(rgb_normalized)

    for i in range(h):
        for j in range(w):
            r, g, b = rgb_normalized[i, j]

            # Value (brightness)
            v = max(r, g, b)
            delta = v - min(r, g, b)

            # Saturation
            s = 0 if v == 0 else delta / v

            # Hue
            if delta == 0:
                h_val = 0
            elif v == r:
                h_val = 60 * ((g - b) / delta % 6)
            elif v == g:
                h_val = 60 * ((b - r) / delta + 2)
            else:  # v == b
                h_val = 60 * ((r - g) / delta + 4)

            hsv_img[i, j] = [h_val, s, v]

    return hsv_img

```

Exercise 1.6: Color Space Analysis

Task: Compare manual vs. OpenCV color conversion

- Implement HSV to RGB conversion
- Measure numerical differences between implementations
- Visualize color space distributions for different image types

Histogram Equalization Implementation

Exercise 1.7: Advanced Image Enhancement

Task: Implement adaptive histogram equalization

```python
def histogram_equalization(img):
    """Global histogram equalization from scratch"""
    # TODO: Implement algorithm from lecture
    pass

def adaptive_histogram_equalization(img, tile_size=(8, 8)):
    """CLAHE (Contrast Limited Adaptive Histogram Equalization)"""
    # TODO: Implement tile-based equalization
    # TODO: Apply contrast limiting
    # TODO: Bilinear interpolation between tiles
    pass

```

Questions:

- When does global histogram equalization fail?
- How does tile size affect adaptive equalization results?
- What are the trade-offs between enhancement and noise amplification?

Advanced Exercises & Extension Challenges

Challenge Problems (Extra Credit)

For students seeking deeper understanding and implementation experience.

Challenge 1: Multi-Camera Calibration

- Implement stereo camera calibration
- Estimate relative pose between two cameras
- Validate epipolar geometry constraints
- **Theory Connection**: How does this relate to fundamental matrix estimation?

Challenge 2: Real-time Calibration

- Implement live camera calibration using webcam
- Add automatic checkerboard detection and quality assessment
- Visualize calibration progress in real-time
- **Theory Connection**: What's the minimum number of views needed?

Challenge 3: Custom Calibration Patterns

- Implement calibration using circular dots instead of checkerboards
- Compare accuracy and robustness vs. checkerboard patterns
- Handle partial pattern visibility
- **Theory Connection**: How does pattern choice affect corner detection accuracy?

Theoretical Questions for Deeper Understanding

Conceptual Questions

1. **Camera Model**: Why is the pinhole camera model insufficient for real cameras? When would you use fisheye vs. perspective models?
2. **Calibration Theory**: Prove that you need at least 6 point correspondences to estimate all camera parameters. What happens with exactly 6 points?
3. **Homography Constraints**: Under what conditions is a homography undefined? How many degrees of freedom does a homography have and why?
4. **Distortion Models**: Why do lens distortion coefficients have different effects? When would higher-order terms be necessary?
5. **Numerical Stability**: Explain why the DLT algorithm becomes unstable without normalization. What is the condition number measuring?

Discussion Questions

- How would you modify Zhang's method for a moving camera setup?
- What calibration accuracy is needed for different applications (AR, robotics, measurement)?
- How do you validate calibration quality without ground truth?

Lab Assessment & Deliverables

What to Submit

1. **Implemented Functions**: All exercise functions with proper documentation
2. **Calibration Results**: Camera parameters from your own calibration dataset
3. **Analysis Report**:
   - Reprojection error analysis
   - Comparison of different algorithms (DLT vs RANSAC)
   - Discussion of failure cases and limitations
4. **Visualization**: Clear plots showing calibration quality and algorithm behavior

Grading Criteria

- **Correctness (40%)**: Algorithms produce mathematically correct results
- **Code Quality (25%)**: Clean, well-documented, efficient implementation
- **Analysis (25%)**: Thoughtful discussion of results and theoretical connections
- **Creativity (10%)**: Extension challenges and novel insights

Next Lab Preview

Lab 2: Custom convolution operations, data augmentation, preprocessing pipelines - building the foundation for deep learning approaches.