Computer Vision Lab #7: Stereo Vision and Epipolar Geometry

Your Name

March 20, 2025

Part I: Disparity Map from Stereo Images

Introduction

3D perception is a critical aspect of computer vision. In this lab, you will calculate a **disparity map** from a pair of stereo images to estimate depth. The images are already rectified, so you do not need to compute intrinsic or extrinsic parameters.

Learning Objectives

- Understand how disparity map calculation works.
- Implement a disparity map algorithm in Python.
- Experiment with hyperparameters and techniques to improve results.

Background

The two images in Figure 1 exhibit a horizontal shift. Similar to how our eyes perceive depth, the scene is captured from two slightly different viewpoints. The principle behind stereo vision is straightforward: objects closer to the camera undergo a larger horizontal displacement than objects farther away. For example, the green cones in Figure 1 exhibit a wider horizontal displacement than the red cones. The goal of this lab is to quantify (in pixels) the displacement of each pixel in the image.

Figure 1: Pair of rectified stereo images. Note the subtle differences between them.

The algorithm works as follows: for each pixel in the left image, extract a patch and search for the best matching patch along the corresponding horizontal line in the right image. However, this process is computationally expensive. To optimize it, we can use two key tricks:

Trick 1: Confine the Search Area

If you know which image is the left and which is the right, you can limit the search to the leftward direction. For example, a pixel at coordinates $(x = 100, y = 100)$ in the left image will never appear at $x > 100$ in the right image. Therefore, you can restrict your search to the range $[1, x]$.

Trick 2: Limit the Search Range

Stereo matching works best when the displacement between the two cameras is small (e.g., the distance between our eyes). This allows you to assume a small search range for matching patches. For instance, if the search area is 64 pixels, you can limit the search to $[x - 64, x]$. For a pixel at $x = 100$, you would search from 36 to 100.

Implementation Tips

- Use an odd number for the patch size (e.g., 3, 5, 7, etc.).
- Ensure the search area does not exceed the image boundaries.

Figure 2 illustrates these concepts.

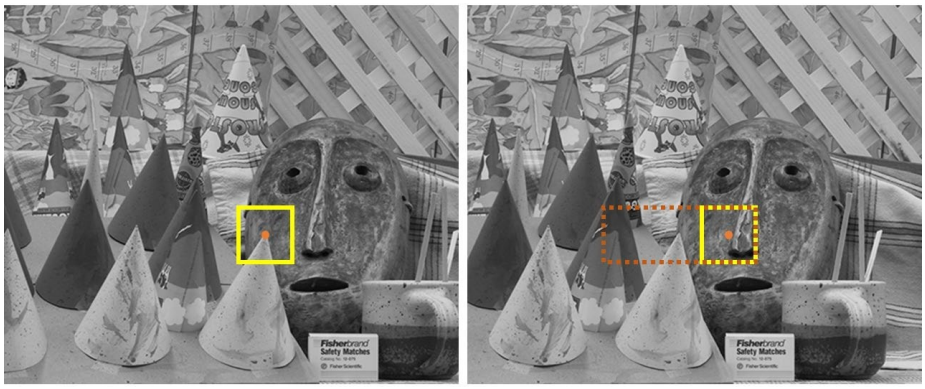

Figure 2: The orange dot represents the pixel for which disparity is calculated. The yellow square denotes the patch size, and the brown rectangle indicates the search area. Note that the search area extends only to the left in the right image.

A Python template is provided to handle image loading, grayscale conversion, and result visualization. Your task is to implement the disparity map calculation.

Task 1: Implement a Disparity Map Calculator

Algorithm

1. Use two nested loops:
   - Outer loop: Iterate over rows ($1 + H$ to HEIGHT $- H$).
   - Inner loop: Iterate over columns ($1 + H$ to WIDTH $- H$).
2. Extract a patch from the left image centered at (x, y).
3. For each pixel in the search window ($x -$ search_area to x):
   - Extract the corresponding patch from the right image.
   - Ensure the search does not exceed image boundaries.
4. Calculate the difference between the patches using **Sum of Absolute Differences (SAD)**:

```
difference = np.sum(np.abs(left_patch - right_patch))
```

5. If the difference is smaller than the previous best, update the best match and its displacement.
6. Store the displacement (in pixels) as the disparity value for the current pixel.

Result

Figure 3 shows an example disparity map. Note the frame around the image (where disparity is zero) due to the patch size, and the shading on the left side caused by the search area limitation.

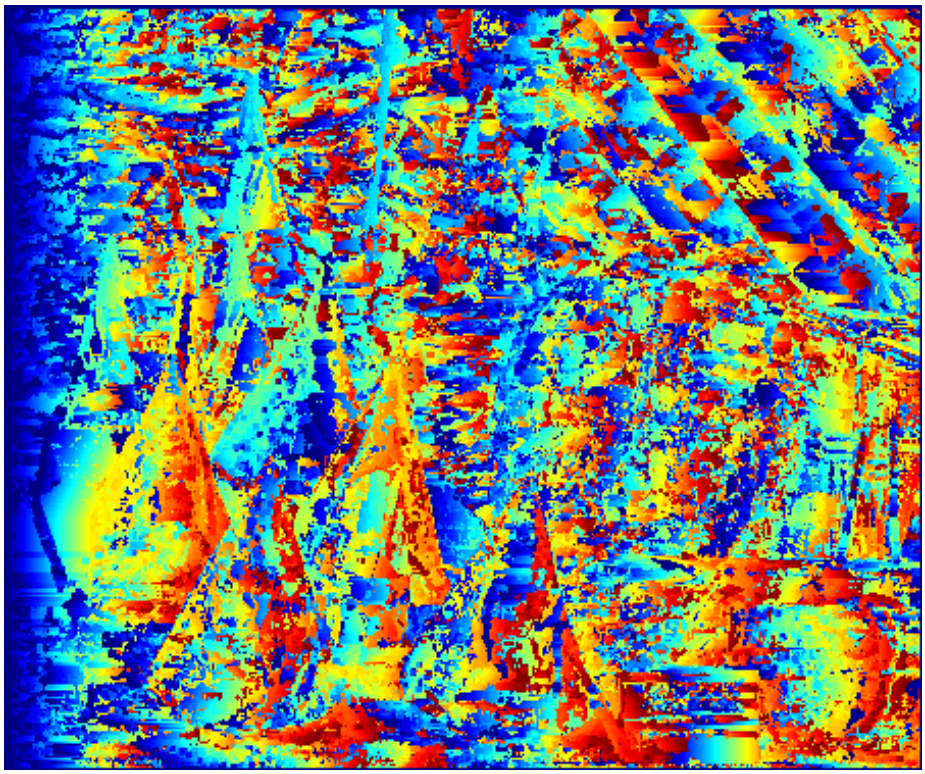

Figure 3: Example disparity map.

Task 2: Improve the Disparity Map

Suggestions for Improvement

- Experiment with hyperparameters (patch size, search area).
- Apply smoothing techniques to the images or disparity map.
- Use alternative patch comparison methods (e.g., normalized cross-correlation).
- Work with color images instead of grayscale.

Part II: Epipolar Geometry and Rectification

Introduction

In this section, you will explore epipolar geometry and image rectification. Specifically, you will compute the fundamental matrix, epipolar lines, and rectify uncalibrated stereo images using OpenCV functions.

Figure 4: Pair of unrectified stereo images. Note the differences in perspective.

Task 1: Epipolar Line Computation

Steps

1. **Find Matching Points**:
   - Use SIFT (Scale-Invariant Feature Transform) to detect and match keypoints between the two images. SIFT is a feature detection algorithm that identifies keypoints and computes their descriptors, which are invariant to scale, rotation, and illumination changes.
2. **Compute Fundamental Matrix**:
   - Use the OpenCV function `cv2.findFundamentalMat()` to compute the fundamental matrix F. This matrix encapsulates the epipolar geometry between the two images.

     ```
     F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC
         )
     ```

   - **Parameters**:
     - `pts1, pts2`: Matching points from the two images.

- `cv2.FM_RANSAC`: Method for robust estimation of the fundamental matrix using RANSAC (Random Sample Consensus).

- Filter out outliers using the mask:

```
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]
```

3. **Compute Epipolar Lines**:

   - Use the OpenCV function `cv2.computeCorrespondEpilines()` to compute the epipolar lines corresponding to the matched points.

```
lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1, 1,
    2), 2, F)
lines1 = lines1.reshape(-1, 3)
```

   - **Parameters**:
     - `pts2`: Points in the second image.
     - `2`: Index of the second image (1 for the first image).
     - `F`: Fundamental matrix.

4. **Visualize Results**:

   - Draw epipolar lines and matching points using `cv2.line()` and `cv2.circle()`.
   - Random colors are assigned to each line and point pair for visualization.

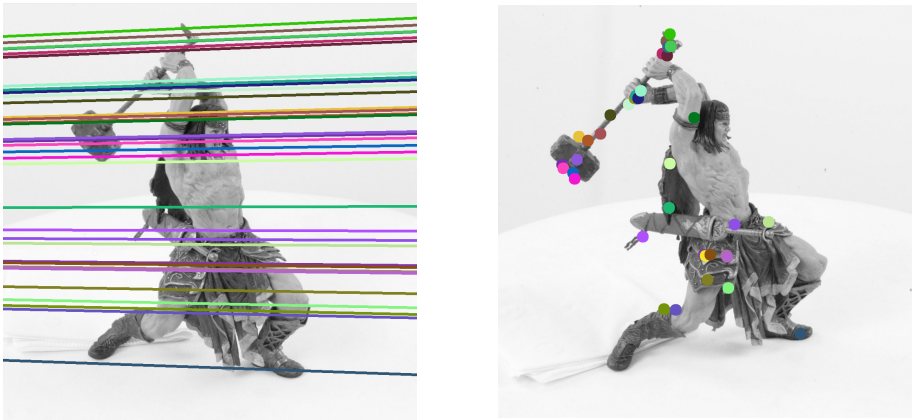

Figure 5: Visualization of epipolar lines and matching points.

Task 2: Image Rectification

Steps

1. **Rectify Images**:
   - Use the OpenCV function `cv2.stereoRectifyUncalibrated()` to compute rectification transformations for the two images.

```
res, H1, H2 = cv2.stereoRectifyUncalibrated(pts1, pts2, F,
     img1.shape, 10)
```

   - **Parameters**:
     - `pts1, pts2`: Matching points from the two images.
     - `F`: Fundamental matrix.
     - `img1.shape`: Size of the input images.
     - `10`: Threshold for filtering outliers.

2. **Warp Images**:
   - Use the OpenCV function `cv2.warpPerspective()` to apply the rectification transformation to the images.

```
imgWarp1 = cv2.warpPerspective(img1, H1, img1.shape)
```

   - **Parameters**:
     - `img1`: Input image.
     - `H1`: Homography matrix for the first image.
     - `img1.shape`: Size of the output image.

3. **Transform Points**:
   - Use the OpenCV function `cv2.perspectiveTransform()` to transform the matching points using the homography matrix.

```
pts1warp = cv2.perspectiveTransform(np.float32(pts1).
    reshape(-1, 1, 2), H1)
pts1warp = np.int32(pts1warp.reshape(-1, 2))
```

4. **Visualize Rectified Images**:
   - Draw matching points and horizontal epipolar lines on the rectified images.

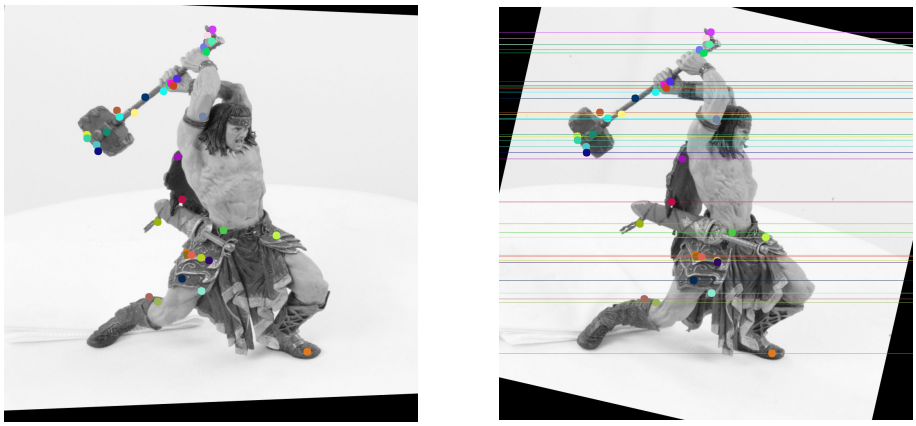

Figure 6: Rectified images with matching points and epipolar lines.

Summary

This lab covers:

- Disparity map calculation from stereo images.
- Epipolar geometry and image rectification using OpenCV.