Pyramidal Lucas-Kanade Optical Flow

Mostafa Hazem Mostafa (2205174)
Khaled Walid Ghalwash (2205018)

April 2025

Abstract

Estimating motion between two frames using Pyramidal Lucas-Kanade Method

This project implements the Pyramidal Lucas-Kanade Optical Flow method to estimate motion between consecutive video frames. By utilizing image pyramids and iterative refinement, we achieve accurate tracking of pixel displacements across various image resolutions, even in the presence of large motion. We demonstrate the effectiveness of the approach through visualization of motion fields.

Contents

1 Introduction

Optical Flow refers to the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera. Among various optical flow methods, the Lucas-Kanade algorithm is particularly effective for small motions. However, to handle larger motions, we extend it using pyramidal techniques.

2 Background

2.1 Optical Flow Estimation

Optical Flow estimation is fundamental in Computer Vision, particularly in motion detection, video tracking, and 3D reconstruction.

2.2 Lucas-Kanade Method

The Lucas-Kanade method assumes that the flow is essentially constant in a local neighborhood of the pixel under consideration, solving for the displacement vector using a system of equations.

2.3 Pyramidal Lucas-Kanade

To deal with larger motions, an image pyramid is built: a multi-scale representation where motion is first estimated on the coarsest level and progressively refined at finer levels.

3 Methodology

3.1 Data Preparation

he dataset used in this project is the **Dimetrodon** sequence from the Middlebury Vision repository. It consists of two frames, denoted as `img0` and `img1`, showing slight motion between them towards upright.

Figure 1: Visualization of the two input frames: `img0` (left) and `img1` (right)

3.2 Gradient Computation

The image gradients in the x and y directions (I_x and I_y) were computed using the Sobel operator. The temporal gradient (I_t) was the difference between frames.

3.3 Image Pyramid Construction

Each frame was downscaled to create multiple levels of resolution. Typically, a 3-level pyramid was constructed with a scaling factor of 0.5.

3.4 Pyramidal Lucas-Kanade Algorithm

The optical flow was first estimated at the lowest resolution. The displacement estimates were then upsampled and refined at higher resolutions. Iterative warping and optical flow updates were performed at each level.

3.5 Visualization of Optical Flow

Flow vectors were plotted using quiver plots where arrows represent the magnitude and direction of motion at sampled grid points.

4 Results

The optical flow was successfully estimated between consecutive frames. The motion was correctly captured, including relatively large displacements thanks to the pyramidal approach. Visualization confirmed the effectiveness of the method, especially compared to single-scale optical flow, which failed to capture larger motions.

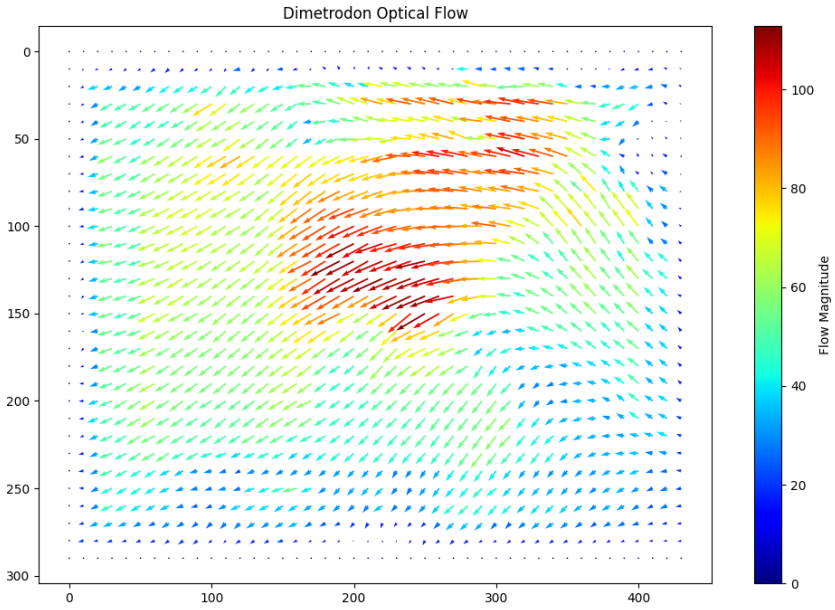

Figure 2: Visualization of Optical Flow using Quiver Plot, Here there's motion exaggeration factor by 50 for visualization.

5 Conclusion

The Pyramidal Lucas-Kanade Optical Flow method provided a robust approach for estimating motion in image sequences, particularly when dealing with larger displacements. The hierarchical refinement strategy greatly enhanced the accuracy of flow estimation compared to traditional methods. Future improvements could involve incorporating feature-based tracking and exploring real-time optimizations.

References

- B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” *Proceedings of Imaging Understanding Workshop*, 1981.
- J. Y. Bouguet, “Pyramidal Implementation of the Lucas-Kanade Feature Tracker,” Intel Corporation, 1999.
- R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2010.
- J. Barron, D. Fleet, and S. Beauchemin, ”Performance of Optical Flow Techniques,” *International Journal of Computer Vision*, vol. 12, no. 1, pp. 43-77, 1994.
- C. Tomasi and T. Kanade, “Detection and Tracking of Point Features,” *Carnegie Mellon University Technical Report CMU-CS-91-132*, 1991.
- J. Shi and C. Tomasi, ”Good Features to Track,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1994.
- S. Avidan and A. Shashua, ”Twodimensional Visual Motion Estimation by Using Linear Information,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 9, pp. 1001-1014, 1996.

A Appendix A: Algorithm Overiview

Algorithm 1 Pyramidal Lucas-Kanade Optical Flow

Input: Images I_1, I_2; levels L; window size w; iterations K
Output: Flow fields u, v
Initialize $u \leftarrow 0$, $v \leftarrow 0$ (same size as I_1) Build Gaussian pyramids $\{P_1^0, \ldots, P_1^L\}$ and $\{P_2^0, \ldots, P_2^L\}$ **for** $l = L$ **to** 0 **do**
  Set $I_1^l \leftarrow P_1^l$, $I_2^l \leftarrow P_2^l$ **if** $l < L$ **then**
    Upsample and scale u, v by 2
  end
  Compute gradients I_x, I_y of I_1^l **for** *each pixel* (x, y) *in* I_1^l **do**
    Initialize $\Delta u \leftarrow 0$, $\Delta v \leftarrow 0$ **for** $k = 1$ **to** K **do**
      Warp I_2^l at $(x + u + \Delta u, y + v + \Delta v)$ Compute $I_t = I_2^l(x + u + \Delta u, y + v + \Delta v) - I_1^l(x, y)$ Collect I_x, I_y, I_t in $w \times w$ window Solve for $(\Delta u, \Delta v)$:

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

      Update: $\Delta u \leftarrow \Delta u + \delta u$, $\Delta v \leftarrow \Delta v + \delta v$
    end
    Update global flow: $u(x, y) \leftarrow u(x, y) + \Delta u$, $v(x, y) \leftarrow v(x, y) + \Delta v$
  end
end
return u, v

A.1 Pyramidal Lucas-Kanade Flow Example

```
def pyramidal_lucas_kanade(I1, I2, levels=4, window_size=15,
  ↪iterations=3):
    # Check for input validation
    if I1.shape != I2.shape:
        raise ValueError("Input images must have the same shape")
    if len(I1.shape) != 2:
        raise ValueError("Input images must be grayscale")
    if window_size % 2 == 0:
        raise ValueError("window_size must be odd")

    # Build Gaussian pyramids for both images
    pyr1 = build_pyramid(I1, levels)
    pyr2 = build_pyramid(I2, levels)

    # Initialize flow at the coarsest level
    u = np.zeros_like(pyr1[0])
    v = np.zeros_like(pyr1[0])

    # Iterate over the pyramid levels
    for level in range(levels):
        I1_l = pyr1[level]
```

```
        I2_l = pyr2[level]

        # Upsample flow at finer levels
        if level != 0:
            u = cv2.pyrUp(u) * 2
            v = cv2.pyrUp(v) * 2
            h, w = I1_l.shape
            u = cv2.resize(u, (w, h), interpolation=cv2.
               ↪INTER_LINEAR)
            v = cv2.resize(v, (w, h), interpolation=cv2.
               ↪INTER_LINEAR)

        # Compute gradients at current level
        Ix, Iy = compute_gradients(I1_l)

        # Iterate for flow refinement
        for _ in range(iterations):
            # Warp second image towards the first using the
               ↪current flow
            I2_warped = warp_image(I2_l, u, v)
            It = I2_warped - I1_l

            # Update flow using least-squares method
            for y in range(hw, I1_l.shape[0] - hw):
                for x in range(hw, I1_l.shape[1] - hw):
                    Ix_win = Ix[y - hw:y + hw + 1, x - hw:x + hw
                       ↪+ 1].flatten()
                    Iy_win = Iy[y - hw:y + hw + 1, x - hw:x + hw
                       ↪+ 1].flatten()
                    It_win = It[y - hw:y + hw + 1, x - hw:x + hw
                       ↪+ 1].flatten()

                    A = np.vstack((Ix_win, Iy_win)).T
                    b = -It_win.reshape(-1, 1)

                    ATA = A.T @ A
                    if np.min(np.linalg.eigvals(ATA)) <
                       ↪eigen_thresh:
                        continue

                    try:
                        nu = np.linalg.lstsq(A, b, rcond=None)[0]
                        u[y, x] += nu[0, 0]
                        v[y, x] += nu[1, 0]
                    except np.linalg.LinAlgError:
                        continue

    return u, v
```

B Appendix B: Mathematical Foundations

The Sobel Operator The Sobel operator computes image gradients in the x and y directions by convolving the image with two kernels. The horizontal Sobel kernel G_x is:

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

and the vertical Sobel kernel G_y is:

$$G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

The gradients are computed by:

$$S_x(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} G_x(m,n) \cdot I(i+m, j+n)$$

$$S_y(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} G_y(m,n) \cdot I(i+m, j+n)$$

where S_x and S_y are the gradients in x and y directions respectively.
The magnitude and direction of the gradient are given by:

$$\text{Magnitude} = \sqrt{I_x^2 + I_y^2}$$

$$\text{Direction} = \arctan\left(\frac{I_y}{I_x}\right)$$

These gradients are crucial for detecting edges and structural information in the images.

The function `warp_image(img, u, v)` The `warp_image` function warps an image by shifting it according to displacement vectors u and v. The mapping is calculated as:

$$\text{map}_x = \text{grid}_x + u, \tag{1}$$
$$\text{map}_y = \text{grid}_y + v \tag{2}$$

This remapping is performed using `cv2.remap` with linear interpolation and reflection at borders (`cv2.BORDER_REFLECT`).

The function `build_pyramid(img, levels)` The `build_pyramid` function creates an image pyramid, which is a sequence of images with progressively lower resolutions. It starts with the original image:

$$\text{pyramid} = [\text{img}]$$

and at each level:

$$\text{img} = \text{cv2.pyrDown(img)}$$

After generating all levels, the list is reversed:

$$\text{pyramid[::-1]}$$

This structure supports multi-scale analysis essential for handling large motion between frames.

The function `pyramidal_lucas_kanade(I1, I2, levels=4, window_size=15, iterations=3, eigen_thresh=1e-4)` The pyramidal Lucas-Kanade method estimates optical flow by iteratively refining flow estimates at multiple resolutions.

First, the input images I_1 and I_2 are normalized:

$$I_1 = \frac{I_1}{255.0}, \quad I_2 = \frac{I_2}{255.0}$$

Image pyramids for both frames are constructed. The initial flow fields are set to zero:

$$u = 0, \quad v = 0$$

At each pyramid level, the flow is upsampled to the current level size using `cv2.pyrUp`. Gradients I_x and I_y are calculated with the Sobel operator, and the flow is estimated by solving the optical flow constraint:

$$A = \begin{pmatrix} I_x & I_y \end{pmatrix}, \quad b = -I_t$$

The least-squares solution for the flow $\nu = (u, v)^T$ at each pixel is:

$$\mathbf{A}^T\mathbf{A}\nu = \mathbf{A}^T\mathbf{b}$$

The process is iterated at each level until convergence or reaching the maximum number of iterations.

C Conclusion

Through the combination of multi-scale analysis and the Lucas-Kanade method, robust optical flow estimation is achieved. The detailed mathematical formulation and careful handling of gradient information are crucial to the success of the algorithm.