CS484: Introduction to Computer Vision
Homework Assignment 3: Assignment Report

Due: December 21, 2024

1.0 Introduction

This report focuses on implementing and analyzing two pivotal computer vision algorithms: the Lucas-Kanade Forward Additive Alignment and the Kanade-Lucas-Tomasi (KLT) Tracker. Both algorithms are essential for tasks such as motion estimation and object tracking, and this assignment aims to highlight their fundamental principles, implementation, and practical applications. The Lucas-Kanade method employs an iterative approach to minimize the sum of squared intensity differences between a template and its target, making it particularly suitable for small displacements. In contrast, the KLT tracker extends Lucas-Kanade by incorporating corner detection and local patch alignment, enabling robust feature tracking over multiple frames.

This report also explores the impact of hyperparameters, such as the convergence threshold (ε) and iteration limits, on performance. Experimenting with these parameters helps uncover the trade-offs between computational efficiency and tracking accuracy. By comparing the algorithms' responses to different configurations, the study provides insights into their strengths and limitations, particularly in handling various motion dynamics and convergence scenarios. Through detailed experimentation and evaluation, this work contributes to a deeper understanding of the interplay between algorithm design and performance in real-world computer vision tasks.

2.0 Methodology

2.1 Lucas-Kanade Forward Additive Alignment

2.1.1 Mathematical Formulation of the Algorithm

We assume a simple translational warp in this questions's version of the "Lucas-Kanade Forward Additive Alignment" algorithm:

$$W(x; p) = x + p$$

where $x = (x, y)^T$ is a pixel coordinate, and $p = (p_x, p_y)^T$ is the translation vector that we wish to estimate.

Given a template region $N \subset \Re^2$ in the reference image (frame) I_t , the goal is to find **p** that best aligns this template onto the next image I_{t+1} . We do this by minimizing the sum of squared intensity differences:

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \sum_{\mathbf{x} \in \mathcal{N}} \left\| I_{t+1}(\mathbf{x}+\mathbf{p}) - I_t(\mathbf{x}) \right\|^2.$$

Here, I_t (x) is the intensity at pixel **x** in frame I_t , and I_{t+1} (**x+p**) is the intensity at the displaced coordinate **x + p** in the next frame I_{t+1} .

Iterative Update via Linearization

Since **p** is typically solved iteratively, let us denote the current estimate of **p** as $p^{(k)}$. We define a small increment $\Delta\mathbf{p}$ and use a first-order Taylor expansion of I_{t+1} around x +$p^{(k)}$:

$$I_{t+1}\left(\mathbf{x}+\mathbf{p}^{(k)}+\Delta\mathbf{p}\right) \approx I_{t+1}\left(\mathbf{x}+\mathbf{p}^{(k)}\right) + \nabla I_{t+1}\left(\mathbf{x}+\mathbf{p}^{(k)}\right) \frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}} \Delta\mathbf{p}.$$

Because our warp is simply **x + p**,

$$\frac{\partial W(\mathbf{x};\mathbf{p})}{\partial \mathbf{p}} = I_{2\times 2},$$

the 2 × 2 identity matrix.

Least-Squares Problem

Substituting the linear approximation back into the objective, we obtain a standard least-squares problem:

$$\arg\min_{\Delta\mathbf{p}} \left\| A\,\Delta\mathbf{p} - \mathbf{b} \right\|^2,$$

where A is constructed from image gradients ∇I_{t+1} at the relevant pixels, and **b** encodes the difference

$$I_t(\mathbf{x}) - I_{t+1}\left(\mathbf{x}+\mathbf{p}^{(k)}\right).$$

The solution is obtained by the normal equations:

$$\Delta \mathbf{p} = \left(A^\top A\right)^{-1} A^\top \mathbf{b}.$$

Parameter Update and Convergence

At each iteration,

$$\mathbf{p}^{(k+1)} \;=\; \mathbf{p}^{(k)} + \Delta \mathbf{p}.$$

We repeat this until $\| \Delta \mathbf{p} \|$ is smaller than a threshold ε or until a maximum iteration limit is reached. The final p (i.e., p^*) is the translation that best aligns the template in I_t to its corresponding region in I_{t+1} .

2.1.2 Explanation of the Implementation

I define a generic convolution function for tasks such as Gaussian smoothing and gradient calculation. Gaussian smoothing is performed using a fixed 3x3 kernel normalized to (1,2,1; 2,4,2; 1,2,1) in order to reduce noise. Then I compute image gradients in the x and y directions using Sobel filters, yielding two gradient images, Ix and Iy. To handle sub-pixel displacements, I implement bilinear interpolation so that each interpolated pixel value is derived from the four surrounding integer sample points. Next, I build an image pyramid by repeatedly blurring and downsampling by a factor of 2, enabling a coarse-to-fine approach.

In the single-scale Lucas-Kanade step, I solve for a 2D displacement p that aligns the template T to the current frame I. At each iteration, I warp coordinates by adding the current estimate of p to the template's grid, interpolate the current frame and its gradients at these warped coordinates, compute the difference between warped intensities and the template, and assemble a least-squares system H dp = b, where H = A^T A (the approximate Hessian) and b = A^T error. A regularization term lambda_reg I can be added to H to ensure invertibility. Solving for dp and updating p continues until the displacement magnitude drops below epsilon or until the maximum number of iterations is reached.

For the pyramidal approach, the same method is applied starting at the coarsest level of the image pyramid; once convergence is achieved at that level, I upsample the displacement estimate p and use it as the initialization for the next finer level. This

strategy significantly improves speed and robustness. Finally, for each frame in a sequence, I apply the pyramidal Lucas-Kanade method to update the bounding box, tracking the object over time, and optionally visualizing each step.

2.1.3 Challenges Faced and How They Are Addressed

The Lucas-Kanade alignment algorithm struggles with fast-moving objects due to its inherent assumption of small displacements between frames. Large displacements caused by rapid motion violate this assumption, leading to misalignment. Specifically, the algorithm uses gradient-based optimization, which relies on small, smooth changes in pixel intensities. When the object moves quickly, the displacement can exceed the algorithm's search region, causing it to converge to incorrect locations or fail altogether. Additionally, the object may appear blurred or distorted due to motion, and if the object has a uniform texture or small size, the image gradients become weak, resulting in an ill-conditioned Hessian matrix. These factors collectively limit the algorithm's effectiveness for fast-moving objects.

A potential mitigation strategy is the **pyramidal approach**, which addresses large displacements by performing alignment at multiple resolutions in a coarse-to-fine manner. This method constructs a Gaussian pyramid where each level represents a progressively downsampled version of the image. At the coarser levels, the resolution is reduced, making large displacements appear smaller and easier to align. The displacement estimate is computed at the coarser level and propagated to finer levels, refining the alignment at higher resolutions for pixel-level precision. By iteratively aligning from coarse to fine resolutions, the pyramidal approach effectively captures large-scale motion while ensuring accuracy, making it a robust solution for tracking fast-moving objects. I have implemented this approach in my solution and in second and third video sequence frames, it helped to increase accuracy but the runtime increased significantly. While it really improved the results for the third dataset, it improved the accuracy a little for the second dataset. I think this is because, in the second dataset, the ball (template object) is moving very quickly, but in the third dataset, the book is moving in a much slower phase. This helps with its alignment accuracy with the pyramidal approach.

I also experimented with regularizing the Hessian matrix using different regularization coefficients to address potential issues with it being ill-conditioned, as an ill-conditioned Hessian can lead to numerical instability and poor updates for the displacement vector. Regularization, by adding a small constant to the diagonal of the Hessian, was intended to stabilize the inversion process during optimization. However, this adjustment did not significantly impact the alignment results, likely because the

primary challenges in alignment were caused by other factors such as large displacements, rather than numerical instability in the Hessian, which is generally a result of distortion or blur available in the images. However, the original images do not include much blur or they are not distorted. Therefore, this approach did not improve the accuracy at all.

I also tried three more approaches—expanded search region, dynamic template updates, and refinement at finer scales—to address the alignment challenges in the Lucas-Kanade pyramidal implementation. However, none of these approaches yielded significant improvements. The expanded search region failed because it introduced unnecessary noise and complexity at coarser levels, making the algorithm prone to mismatches. Dynamic template updates struggled as the object's appearance changed rapidly or inconsistently, leading to the propagation of errors over time. Lastly, the refinement step at finer scales did not help as the misalignments from higher levels were too large to be corrected in a few iterations. These challenges suggest that the underlying assumptions of small displacements and consistent template appearance might not hold for the given dataset, limiting the effectiveness of these enhancements.

2.1.4 Approach to Experimenting with Different ε and Iteration Values

I experimented with epsilon values of 1e-3, 1e-4, and 1e-5 because using 1e-6 mostly resulted in excessively long runtimes across the entire dataset, leading to significantly reduced efficiency. For these epsilon values, I set the max_iter value as high as possible to allow the algorithm to converge based on the first suitable epsilon, and I tested the accuracy under these conditions. Subsequently, I adjusted the max_iter value to better balance the trade-off between accuracy and efficiency.

2.2 Kanade-Lucas-Tomasi (KLT) Tracker

2.2.1 Mathematical Formulation of the Algorithm

Harris Corner Detector

A common approach for selecting features (corners) to track is the **Harris detector**. For each pixel $\mathbf{x} = (x, y)^\top$ in an image I, we define the structure tensor (or second-moment matrix)

$$M(\mathbf{x}) = \begin{pmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{pmatrix},$$

where Σ indicates a summation over a small window (e.g., a 3 × 3 or 5 × 5 patch) centered at **x**, and I_x , I_y are partial derivatives of I w.r.t. x and y. Then the **Harris response** is computed as:

$$R(\mathbf{x}) = \det\big(M(\mathbf{x})\big) - k\ \big(\mathrm{trace}\big(M(\mathbf{x})\big)\big)^2,$$

where k is a constant (commonly around 0.04). Locations **x** with large R(**x**) are considered corners or high-contrast features. After computing R(**x**) for all pixels, we perform thresholding and non-maximum suppression to select strong corners.

Per-Corner Local Alignment

Once corners are detected, each corner is *tracked* in the subsequent frame(s) using a local **Lucas-Kanade** approach, but restricted to a small patch around the corner.

Local Warp and Objective

We assume a pure translation warp for each corner:

$$W(\mathbf{x}; \mathbf{p}) = \mathbf{x} + \mathbf{p},$$

where $\mathbf{p} = (p_x, p_y)^\top$ now represents the per-corner displacement. For a corner initially at $\mathbf{x} = (x, y)^\top$ in frame It, we look at a small window (e.g., 15 × 15) around *(x, y)* and define an objective to minimize the intensity difference in the next frame I_{t+1} :

$$\arg\min_{\mathbf{p}} \sum_{\mathbf{u} \in \Omega} \big\| I_{t+1}(\mathbf{u} + \mathbf{p}) - I_t(\mathbf{u}) \big\|^2,$$

where Ω is the set of pixel coordinates in the local patch around *(x, y)*.

Iterative Update

As with Lucas-Kanade, we solve for **p** iteratively. Let $p^{(k)}$ be the current estimate. If we assume a small increment Δ**p**, we do a first-order Taylor expansion:

$$I_{t+1}(\mathbf{u} + \mathbf{p}^{(k)} + \Delta\mathbf{p}) \ \approx\ I_{t+1}(\mathbf{u} + \mathbf{p}^{(k)}) + \nabla I_{t+1}(\mathbf{u} + \mathbf{p}^{(k)})\ \Delta\mathbf{p}.$$

Collecting terms, we arrive at a least-squares problem

$$\arg\min_{\Delta\mathbf{p}} \big\| A\,\Delta\mathbf{p} - \mathbf{b} \big\|^2,$$

where A depends on the local image gradients, and **b** depends on the intensity differences between I_t and I_{t+1} (warped by $p^{(k)}$)). Solving the normal equations,

$$\Delta \mathbf{p} = (A^\top A)^{-1} A^\top \mathbf{b},$$

and updating $p^{(k+1)} = p^{(k)} + \Delta \mathbf{p}$ iteratively until $\| \Delta \mathbf{p} \| < \varepsilon$ or a max iteration limit is reached.

Bounding Box Update

Let us assume we have N corners $\{x_i\}_{i=1}^{N}$ in the bounding box. After each corner is tracked from frame t to $t + 1$, we obtain displacements $(\Delta x_i, \Delta y_i)$. A simple way to update the bounding box is to compute the average displacement across all corners:

$$\Delta x_{\text{box}} = \frac{1}{N} \sum_{i=1}^{N} \Delta x_i, \quad \Delta y_{\text{box}} = \frac{1}{N} \sum_{i=1}^{N} \Delta y_i.$$

Then the bounding box shifts by $(\Delta x_{box}, \Delta y_{box})$. Mathematically, if the bounding box in frame t is at (x, y, w, h), the new bounding box in frame $t + 1$ is $(x + \Delta x_{box}, y + \Delta y_{box}, w, h)$.

Re-Detection of Corners

Over time, corners may drift out of view or become unreliable. Hence, KLT trackers often *re-detect* corners periodically or whenever the number of tracked corners falls below a threshold. Formally:

if (#{corners} < M) then re-run Harris on I_{t+1} to add new corners.

We only keep new corners that lie within the updated bounding box, maintaining a stable set of tracked features.

2.2.2 Explanation of the Implementation

I start by importing necessary libraries like cv2, numpy, and matplotlib. Then I define the following utilities:

gaussian_blur uses cv2.GaussianBlur with a 3x3 kernel to reduce noise.

compute_image_gradients calculates Ix and Iy via cv2.Sobel for horizontal and vertical gradients.

bilinear_interpolate does sub-pixel sampling of the image by blending the values at the four closest integer pixel coordinates.

build_pyramid repeatedly blurs and down-samples the image to build a coarse-to-fine pyramid.

harris_corners implements Harris corner detection by computing the structure tensor from Ix and Iy, then using the corner response function R. It selects local maxima above a threshold.

lucas_kanade_point performs single-scale optical flow for a single point. It iterates by comparing the patch from the previous frame (T) to the current frame (Iw) in a least-squares sense, forming H dp = b, where H = A^T A, b = A^T (T - Iw), and dp is the incremental motion update. Iteration continues until dp is under epsilon or max_iterations is reached.

lucas_kanade_point_pyramid does the same thing but starts from a coarse pyramid level, computes the displacement, and refines at finer levels, making the approach more robust for larger motions.

track_corners_klt is the main function. It loads the first frame, detects Harris corners in a bounding box, and tracks them across each subsequent frame with lucas_kanade_point_pyramid. Median-based filtering removes outliers. Periodically, or if corner count is low, it re-detects corners in the bounding box. The bounding box position is updated by taking the median displacement of the tracked corners.

visualize_tracking draws bounding boxes and tracked corners on each frame and displays the results.

Hyperparameters include:

- harris_thresh and harris_k for Harris corner detection
- corner_count for the maximum corners to keep
- epsilon and max_iterations for Lucas-Kanade convergence
- window_size for the local patch size
- pyramid_levels for the multi-scale approach
- re_detection_rate for how often corners are re-detected

However, I have only experimented with epsilon and max_iterations and re_detection_rate, and keep the others the same with reasonable values, since we are not asked to experiment with these parameters. Overall, this way, the KLT pipeline tracks salient corners within a bounding box, updates their positions frame to frame, and provides a stable bounding-box estimate over a video sequence.

2.1.3 Challenges Faced and How They Are Addressed

To address runtime performance challenges in the Lucas-Kanade tracker, I optimized the algorithm by replacing manual convolution and Gaussian blurring with OpenCV's highly efficient cv2.GaussianBlur and cv2.Sobel functions. These native-compiled operations eliminated the need for slow, nested Python loops. Additionally, I computed image gradients once per frame instead of recalculating them for each corner, reducing redundant computations. These changes significantly improved the algorithm's efficiency, leveraging optimized functions and minimizing unnecessary calculations, resulting in faster processing without sacrificing accuracy. The runtime dropped significantly.

Another challenge was to prevent outlier corners from distorting the bounding box update, ensuring robust and accurate tracking by relying only on stable, inlier corners. Initially, I only removed corners that drifted outside the bounding box and then updated the bounding box using the average displacement. However, this could still let outlier corners influence the bounding box update. Then, I first compute displacements and use a median-based approach to detect outliers before updating the bounding box. By doing so, I ensure that only stable, inlier corners affect the bounding box shift. Additionally, after outlier removal and bounding box updating, I still ensure I have enough corners by re-detecting them if needed. These changes improve robustness by tackling the challenge of drifting and outlier corners more effectively. This refined approach reduces the impact of erroneous corners and produces a more stable and accurate tracking result.

Again pyramidal implementation of Lucas-Kanade Alignment is leveraged here. However, in the tracking algorithm, it did not improve the tracking accuracy at all. It just improved the results for the second dataset a little. Therefore, I just applied pyramidal levels to get the results for the second dataset and just worked on the original scale for the other two video sequences.

2.1.4 Approach to Experimenting with Different ε and Iteration Values

I experimented with epsilon values of 1e-3, 1e-4, and 1e-5 because using 1e-6 mostly resulted in excessively long runtimes across the entire dataset, leading to significantly reduced efficiency. For these epsilon values, I set the max_iter value as high as possible to allow the algorithm to converge based on the first suitable epsilon, and I tested the accuracy under these conditions. Subsequently, I adjusted the max_iter value to better balance the trade-off between accuracy and efficiency.

3.0 Experimental Results

3.1 Visualization of Tracking Results

3.1.1 Lucas-Kanade Forward Additive Alignment

3.1.1.1 Visualizations of Tracked Bounding Boxes Overlaid on Video Frames (Best Results Obtained)

3.1.1.1.1 Dataset 1

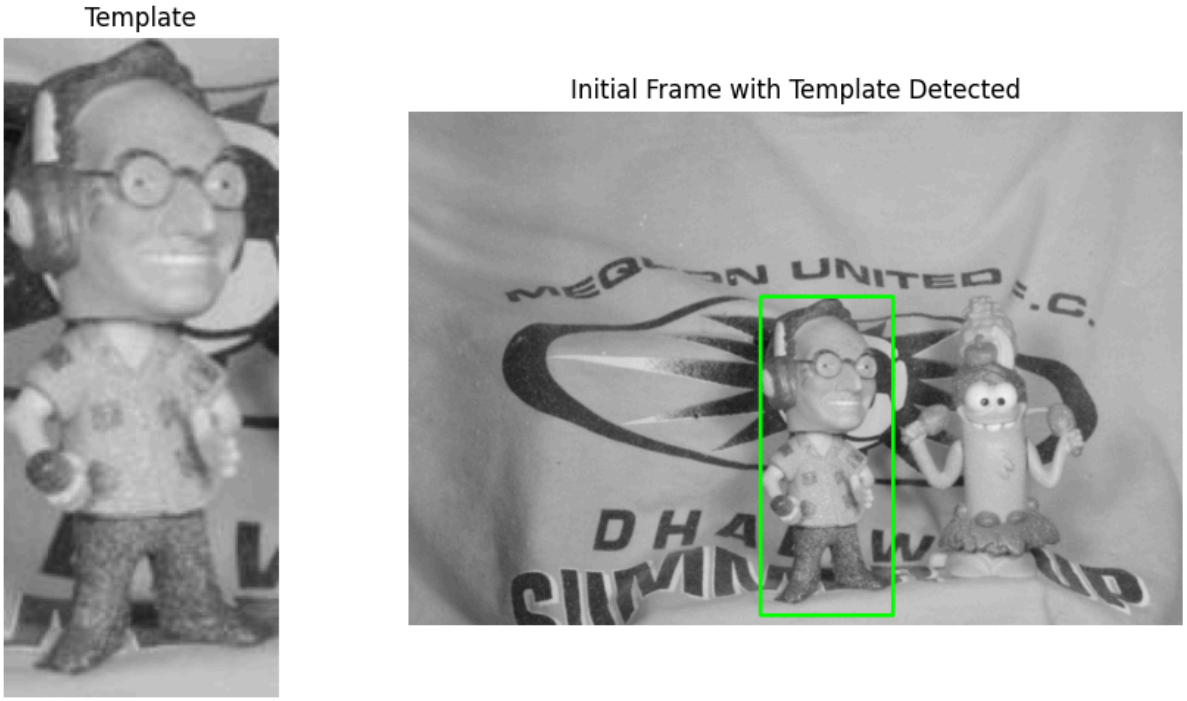

Figure 1: Template Image for Dataset 1 and Template Bounded in the Initial Frame Using Given Metadata

Best Result Obtained with Parameter Values:

epsilon = 1e-5
max_iterations = 300
pyramid_levels = 1

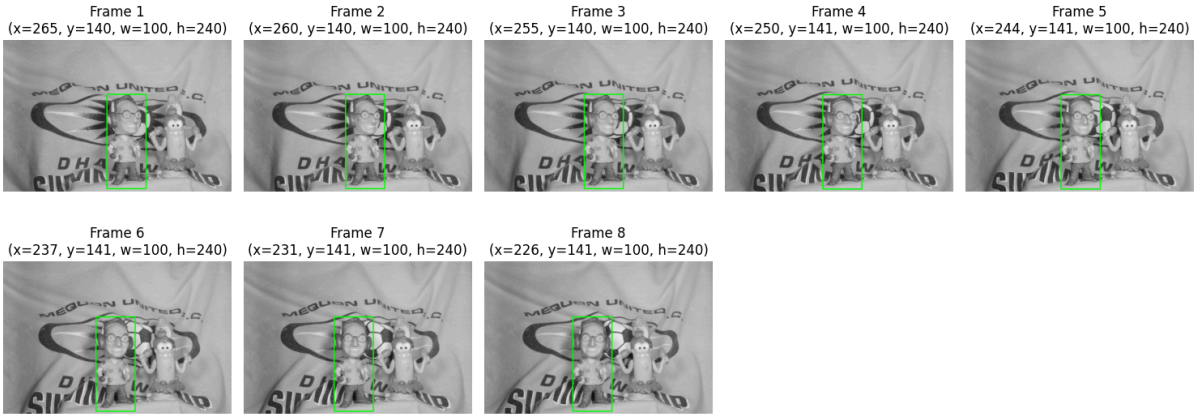

Figure 2: Visualizations and Coordinates of Tracked Bounding Boxes Overlaid on Video Frames of Dataset 1

3.1.1.1.2 Dataset 2

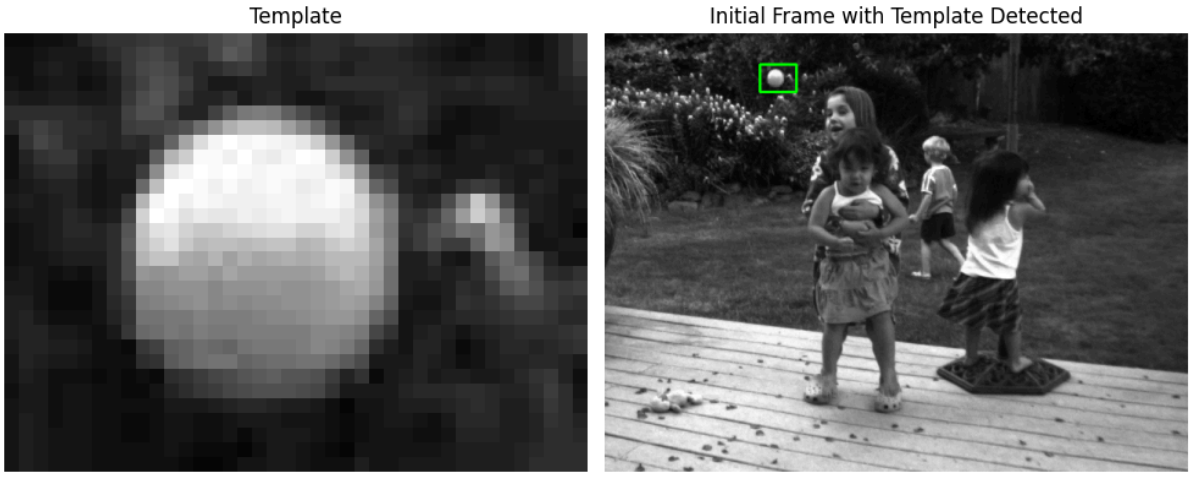

Figure 3: Template Image for Dataset 2 and Template Bounded in the Initial Frame Using Given Metadata

Best Result Obtained with Parameter Values:

epsilon = 1e-6
max_iterations = 700
pyramid_levels = 5

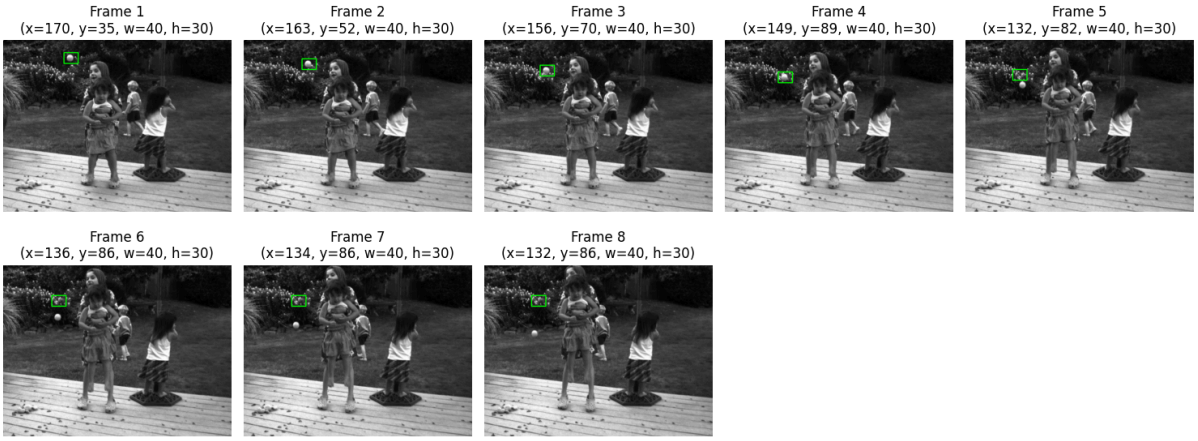

Figure 4: Visualizations and Coordinates of Tracked Bounding Boxes Overlaid on Video Frames of Dataset 2

3.1.1.1.3 Dataset 3

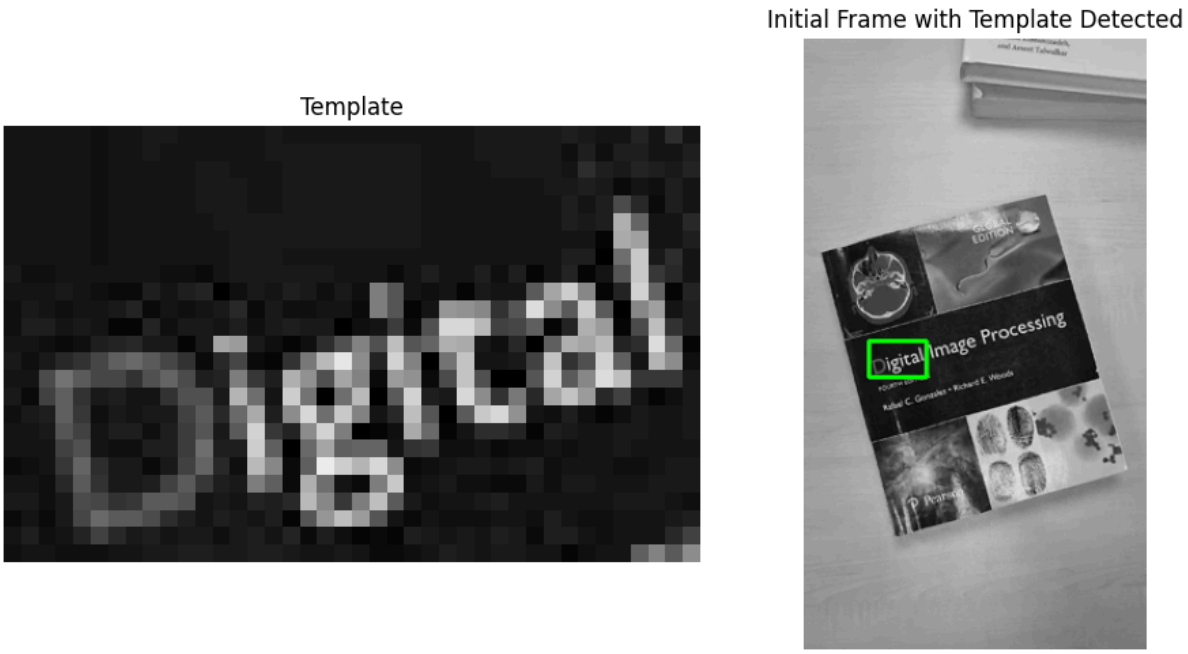

Figure 5: Template Image for Dataset 3 and Template Bounded in the Initial Frame Using Given Metadata

Best Result Obtained with Parameter Values:

epsilon = 1e-5
max_iterations = 200
pyramid_levels = 3

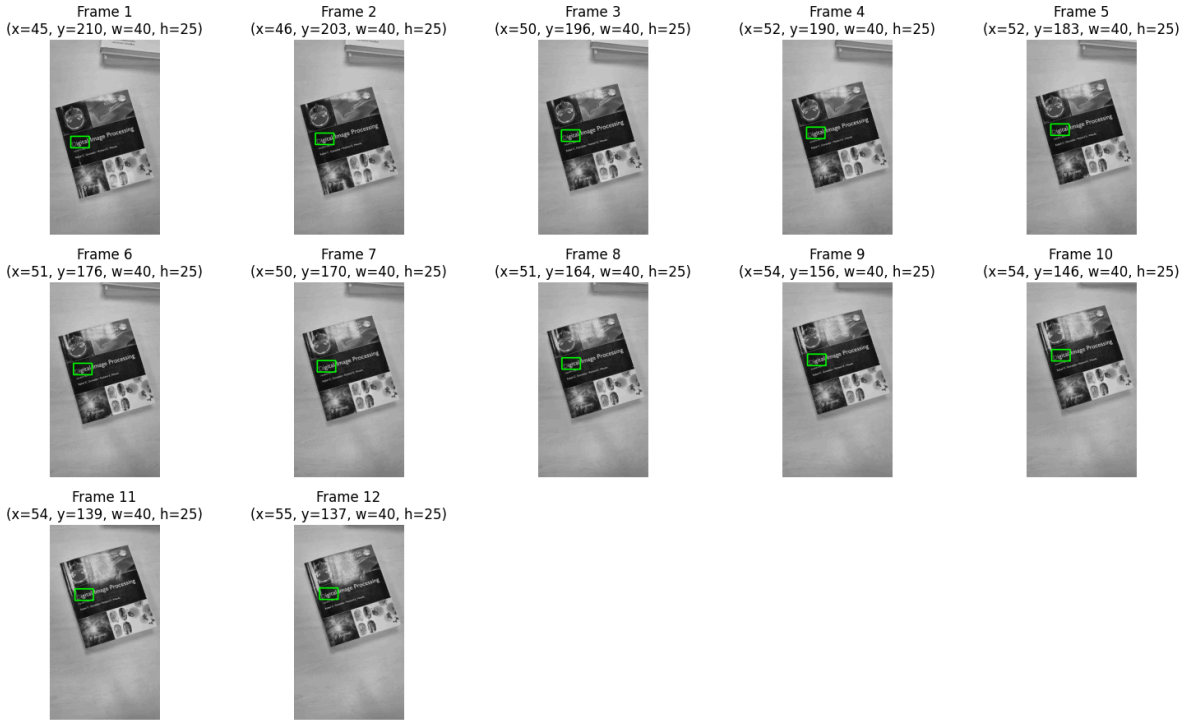

Figure 6: Visualizations and Coordinates of Tracked Bounding Boxes Overlaid on Video Frames of Dataset 3

3.1.1.2 Number of Iterations Required for Convergence Under Different ε Selections

3.1.1.2.1 Dataset 1

Frame Number (as named in the dataset)/ Epsilon Value	1e-3	1e-4	1e-5
08	81	110	139
09	106	148	191
10	165	220	276
11	173	220	268
12	170	225	281
13	162	223	284
14	148	209	270

Table 1: Number of Iterations Required for Convergence Under Different ε Selections for Dataset 1

3.1.1.2.2 Dataset 2

Frame Number (as named in the dataset)/ Epsilon Value	**Pyramidal Levels**	**1e-3**	**1e-5**	**1e-6**
08	**Level 0**	36	90	121
	Level 1	37	84	108
	Level 2	25	55	69
	Level 3	14	30	38
	Level 4	23	38	45
09	**Level 0**	18	120	178
	Level 1	44	114	150
	Level 2	28	71	90
	Level 3	32	69	89
	Level 4	46	85	104
10	**Level 0**	58	164	217
	Level 1	33	88	117
	Level 2	34	72	94
	Level 3	46	83	103
	Level 4	54	123	158
11	**Level 0**	120	328	435
	Level 1	162	304	372

	Level 2	43	128	168
	Level 3	21	40	49
	Level 4	24	66	87
12	**Level 0**	111	477	674
	Level 1	126	293	378
	Level 2	51	104	130
	Level 3	24	48	60
	Level 4	22	63	84
13	**Level 0**	92	461	668
	Level 1	132	317	412
	Level 2	57	132	171
	Level 3	30	53	64
	Level 4	37	53	62
14	**Level 0**	94	442	637
	Level 1	130	316	411
	Level 2	51	102	125
	Level 3	31	57	70
	Level 4	35	65	77

Table 2: Number of Iterations Required for Convergence Under Different ε Selections for Dataset 2

3.1.1.2.3 Dataset 3

Frame Number (as named in the dataset)/	**Pyramidal Levels**	**1e-3**	**1e-4**	**1e-5**

Epsilon Value				
2	**Level 0**	14	25	38
	Level 1	16	29	42
	Level 2	31	41	51
3	**Level 0**	10	16	22
	Level 1	16	27	37
	Level 2	34	45	55
4	**Level 0**	12	20	28
	Level 1	19	29	40
	Level 2	28	40	52
5	**Level 0**	17	26	35
	Level 1	15	23	33
	Level 2	32	40	49
6	**Level 0**	11	18	25
	Level 1	10	23	34
	Level 2	32	44	57
7	**Level 0**	14	22	31
	Level 1	11	21	32
	Level 2	26	34	43
8	**Level 0**	14	21	27
	Level 1	8	16	25
	Level 2	28	40	53
9	**Level 0**	21	36	51
	Level 1	9	23	37

	Level 2	38	52	66
10	**Level 0**	27	39	52
	Level 1	26	39	52
	Level 2	61	73	85
11	**Level 0**	22	29	36
	Level 1	49	66	82
	Level 2	39	53	67
12	**Level 0**	127	93	105
	Level 1	83	123	161
	Level 2	39	53	68

Table 3: Number of Iterations Required for Convergence Under Different ε Selections for Dataset 3

In general, smaller epsilon values lead to more iterations because the convergence criterion is stricter. For instance, comparing 1e-3 to 1e-4 or 1e-5, the iteration counts consistently rise. This pattern holds across different frames and different pyramid levels, reflecting that the algorithm must perform additional iterations to achieve the finer error tolerance. In pyramidal approaches, coarser levels (larger level index in the tables) often converge in fewer iterations, since the displacements are initially estimated at a smaller resolution. As the alignment moves to finer levels (lower level index), the patch size is larger in pixel terms and subtler motion remains to be refined, sometimes increasing the iteration count. However, overall, the use of pyramids still reduces the total alignment time versus a single-scale approach because large displacements are captured at coarse levels. Finally, in certain frames, when motion complexity is higher (e.g., significant perspective change or texture variation), even larger iteration values are observed for smaller epsilon thresholds, showing how difficult alignment can become for those specific frames under strict accuracy constraints.

3.1.2 Kanade-Lucas-Tomasi (KLT) Tracker

3.1.2.1 Visualizations of Tracked Bounding Boxes and Corner Points Overlaid on Video Frames (Best Results Obtained)

3.1.2.1.1 Dataset 1

Best Result (with Paying Attention to Accuracy vs. Efficiency Trade-Off) Obtained with Parameter Values:

epsilon = 1e-5
max_iterations = 1200
pyramid_levels = 1

Figure 7: Visualizations and Coordinates of Tracked Bounding Boxes and Overlaid on Video Frames of Dataset 1

3.1.2.1.2 Dataset 2

Best Result (with Paying Attention to Accuracy vs. Efficiency Trade-Off) Obtained with Parameter Values:

epsilon = 1e-5
max_iterations = 2000
pyramid_levels = 3

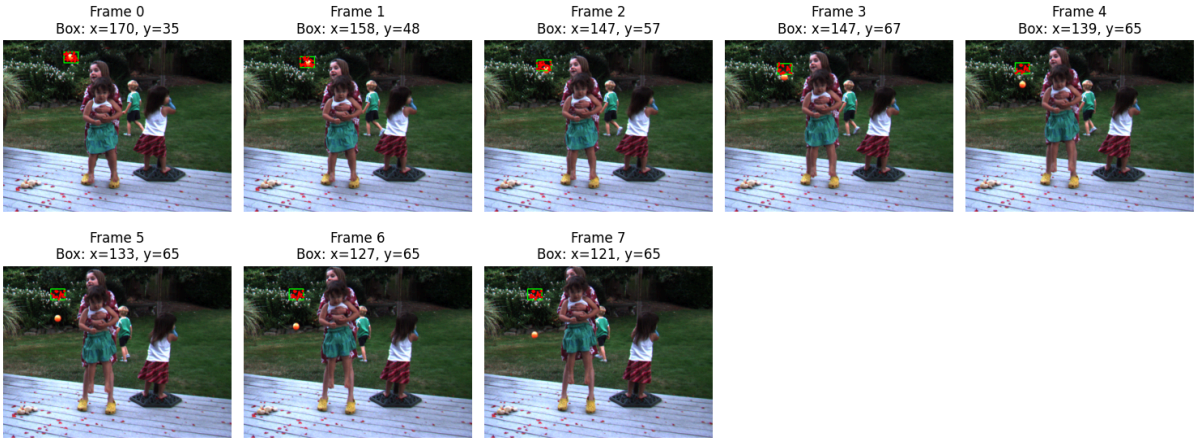

Figure 8: Visualizations and Coordinates of Tracked Bounding Boxes and Overlaid on Video Frames of Dataset 2

3.1.2.1.3 Dataset 3

Best Result (with Paying Attention to Accuracy vs. Efficiency Trade-Off) Obtained with Parameter Values:

epsilon = 1e-5
max_iterations = 3000
pyramid_levels = 1

Figure 9: Visualizations and Coordinates of Tracked Bounding Boxes and Overlaid on Video Frames of Dataset 3

3.1.2.2 Number of Iterations Required for Convergence Under Different ε Selections

3.1.2.2.1 Dataset 1

Frame Number (as named in the dataset)/ Epsilon Value	**1e-3**	**1e-4**	**1e-5**
08	149	176	217
09	99	113	129
10	580	604	687
11	179	245	313
12	493	809	1141
13	345	547	757
14	142	172	227

Table 4: Maximum Number of Iterations Required for Convergence Among All Detected Corners Under Different ε Selections for Dataset 1

3.1.2.2.2 Dataset 2

Frame Number (as named in the dataset)/ Epsilon Value	**1e-3**	**1e-4**	**1e-5**
08	753	765	834
09	1354	1552	1663
10	526	580	614
11	289	466	646
12	46	66	493
13	45	60	76

14	45	75	100

Table 5: Maximum Number of Iterations Required for Convergence Among All Detected Corners and All Pyramid Levels Under Different ε Selections for Dataset 2

3.1.2.2.3 Dataset 3

Frame Number (as named in the dataset)/ Epsilon Value	**1e-3**	**1e-4**	**1e-5**
2	361	594	573
3	664	763	594
4	666	539	683
5	880	1460	2088
6	1271	1508	1511
7	449	442	461
8	204	1198	2593
9	438	657	839
10	591	583	695
11	571	1143	1513
12	44	61	77

Table 6: Maximum Number of Iterations Required for Convergence Among All Detected Corners Under Different ε Selections for Dataset 3

In the KLT tracker, each corner is tracked independently, and the “Maximum Number of Iterations” reported reflects the slowest corner to converge in each frame. As expected, using a smaller epsilon makes the convergence criterion stricter, causing certain corners to demand many additional iterations before meeting that condition. The data show that some frames exhibit particularly large iteration counts, which often correlates with scenes having higher motion complexity, occlusions, or less distinct corner features. Additionally, even within the same frame, different corners may converge at widely

varying rates. Thus, while lower epsilon can yield more precise alignment, it also significantly extends runtime, especially if some corners are difficult to track. This behavior underscores the inherent trade-off in KLT tracking between setting a small epsilon for accuracy and keeping iteration counts manageable for efficiency.

3.2 Comparative Analysis

The Lucas-Kanade Forward Additive Alignment and KLT Tracker each have strengths and weaknesses. Lucas-Kanade aligns an entire template and performs well with moderate motion and stable object appearance, especially when using a pyramidal approach to handle larger displacements. However, it struggles with rapid motion or occlusions due to its reliance on a single global template. In contrast, the KLT Tracker is more robust to occlusions and dynamic changes by tracking multiple corner features and periodically re-detecting lost ones, though this introduces computational overhead.

Both methods are sensitive to the convergence threshold (ε) and iteration limits. Smaller ε values improve accuracy but significantly increase iteration counts, especially in complex motion. Lucas-Kanade's iterations are more consistent, while KLT may be slowed by challenging corners requiring many iterations. Higher iteration limits allow more refinement but can result in diminishing accuracy returns, while too-low limits risk incomplete alignment.

In tests, Lucas-Kanade worked better in slower motion scenarios, achieving stable alignment, while KLT excelled in dynamic environments like rapid motion, adapting to changes in corner features. The trade-off between accuracy and efficiency makes Lucas-Kanade preferable for stable objects with manageable motion and KLT ideal for dynamic settings with occlusions. Parameter tuning is critical to balancing performance and computational cost for each task.

4.0 Discussion

The Lucas-Kanade method excels in tracking stable objects with moderate motion, particularly when using a pyramidal approach to handle large displacements. However, it struggles with rapid motion or occlusions due to reliance on a static template, and smaller ε values, while increasing accuracy, significantly inflate runtimes. The KLT Tracker is more robust to occlusions and dynamic changes by tracking multiple corners and re-detecting lost ones but suffers from inefficiency due to outlier corners and the computational cost of re-detection.

Improvements for Lucas-Kanade include adaptive templates to handle appearance changes and dynamic search regions for rapid motion. For KLT, robust outlier filtering and faster corner re-detection can enhance efficiency. Both methods could benefit from predictive models for motion estimation and hierarchical frameworks to efficiently manage large displacements and appearance changes. These strategies would improve accuracy and robustness while balancing computational cost.

5.0 Conclusion

In summary, this report highlights the nuanced interplay between accuracy and efficiency in the Lucas-Kanade and KLT algorithms. Experimenting with varying ε and iteration values revealed critical thresholds impacting convergence and runtime. While the pyramidal approach improved robustness against large displacements, its benefits were context-dependent, as observed in the datasets. Challenges such as handling rapid motion, outlier removal, and runtime optimization underscored the practical complexities of implementing these algorithms. Overall, this assignment not only deepened understanding of alignment and tracking methodologies but also provided a framework for future exploration of advanced computer vision techniques.

6.0 References

1. OpenCV, "Harris Corner Detection," [Online]. Available: https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html. [Accessed: 21-Dec-2024].

2. OpenCV, "OpenCV," [Online]. Available: https://opencv.org. [Accessed: 21-Dec-2024].

3. Carnegie Mellon University, "Alignment and Tracking: Lucas-Kanade Algorithm," [Online]. Available: https://www.cs.cmu.edu/~16385/s17/Slides/14.4_Alignment__LucasKanade.pdf. [Accessed: 21-Dec-2024].

4. MathWorks, "Lucas-Kanade Tracker with Pyramid and Iteration," [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/30822-lucas-kanade-tracker-with-pyramid-and-iteration. [Accessed: 21-Dec-2024].

5. Course Slides found in the Moodle page of the course.