CS7.505.S22: Computer Vision | Assignment 2

Author: Aditya Kumar Singh
ID: 2021701010

March 22, 2022

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Contents

Question 2: Single-Scale Lucas-Kanade Optical Flow **1**
Keypoint Selection: Selecting Pixels to Track 1
Forward-Additive Sparse Optical Flow 2

Question 3: Multi-Scale Coarse-to-fine Optical Flow **4**

Question 2: Single-Scale Lucas-Kanade Optical Flow

Keypoint Selection: Selecting Pixels to Track

Implement (i) Harris Corner Detector and (ii) Shi-Tomasi Corner Detector and visualize the feature set of pixels obtained by both algorithms that will be tracked by the sparse LK method. Visualize the detected pixels superimposed on the images for at least one image from each of the given sequences.

This part of the assignment is based on the 1988 work, A Combined Corner and Edge Detector, by Chris Harris and Mike Stephens, and 1994 CVPR paper, Good Features to Track, by Jianbo Shi and Tomasi.

Answer:

Harris corner detection as well as *Shi-Tomasi* corner detection are both gradient-based algorithm that compute gradients for a point locally (through a small window) and using that try to figure out whether that point is a corner point or not (meaning are there any two prevalent directions around that point where maximum intensity change is happening $\implies$ "which more or less denote are there any edge crossing happening $\longrightarrow$ which is nothing but a corner"). The eigen-values of the co-occurence derivative matrix ($= \begin{bmatrix} I_x.I_x & I_x.I_y \\ I_y.I_x & I_y.I_y \end{bmatrix}$) helps us to do so in deciding whether to consider that point or discard it altogether.
We'll just highlight the crux of both the method that decides which points to choose.

1. For Harris-corner based method:
$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$
where W denotes the window. Then we calculate *Harris response* i.e., the smallest eigenvalue of M which is computed using an approximation,
$$\lambda_{\min} \approx \frac{\lambda_1\lambda_2}{\lambda_1+\lambda_2} = \frac{\det(M)}{trace(M)}$$

2. Whereas in case of Shi-Tomasi, the only difference is that they don't do any approximations, rather they directly take $\lambda_{\min}$.
But to avoid eigen-value calculation, I came up with a nice formulation as shown below:
$$\begin{aligned} \min(\lambda_1,\lambda_2) &= \frac{\lambda_1+\lambda_2-|\lambda_1-\lambda_2|}{2} \\ &= \frac{trace(M)-|\lambda_1-\lambda_2|}{2} \\ &= \frac{trace(M)-|\sqrt{trace(M)^2-4\det(M)}|}{2}, \text{ as} (a-b)^2 = (a+b)^2-4ab \end{aligned}$$

Now let's hop to the results we got from both the methods.
(*Disclaimer: The results we got are more or less similar as both of them follow similar mrthodology*)

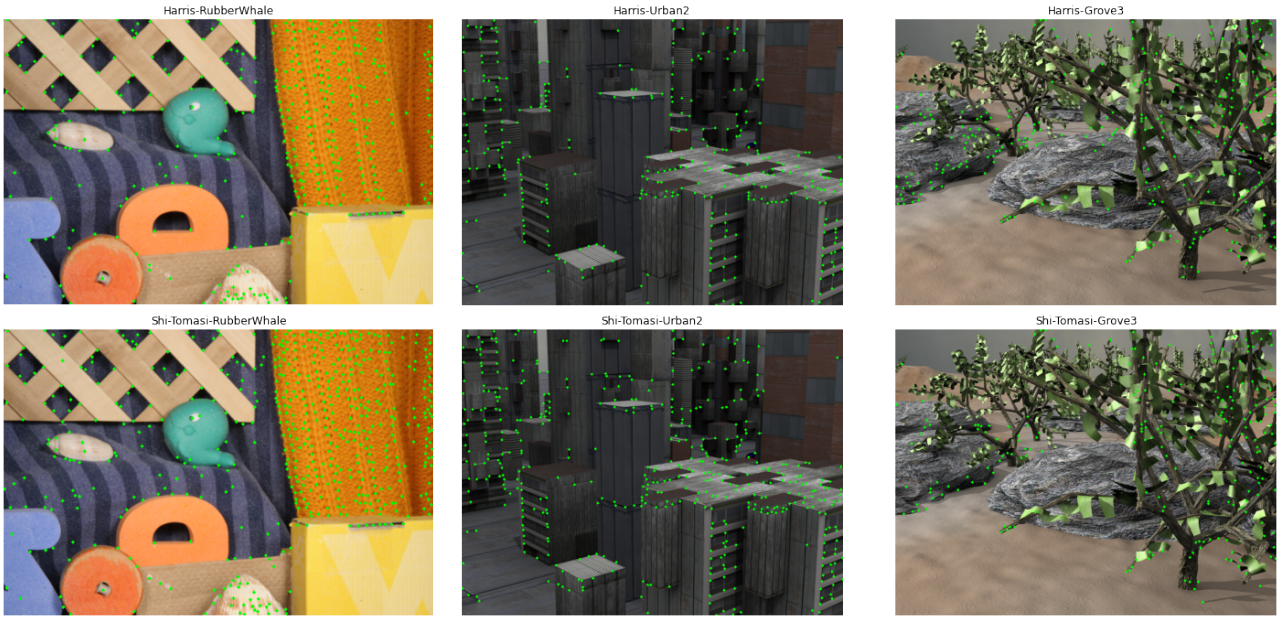

Figure 1: With threshold = 850 for responses, we obtained above corners. The top one being for Harris; while the bottom one for Shi-Tomasi

Forward-Additive Sparse Optical Flow

Implement the single-scale Lukas-Kanade (LK) algorithm based on the work described in the classic 1981 IJCAI paper An iterative image registration technique with an application to stereo vision by Bruce D. Lucas and Takeo Kanade. This involves finding the motion (u, v) that minimizes the sum-squared error of the brightness constancy equations for each pixel in a window. Your algorithm will be implemented as a function with the following inputs,
`LukasKanadeForwardAdditive(Img1, Img2, windowSize)`
You will use all the given image sequences from Middlebery Optical Flow dataset throughout part two to test your algorithm. First, compile the ground truth optical flow into a video to see it. Extract the vertical and horizontal flow and repeat the same. *Before running your code on the images, you should first convert your images to grayscale and map intensity values to the range* $[0, 1]$.

Algorithm:
(*Do note that all our algorithms from hereon are based upon sparse feature points.*)

`Step 01: Normalize the gray scale images by dividing it with` $2^8 - 1 = 255$.

`Step 02: Apply Gaussian blurring on both normalized images with scale` $= \sigma = \frac{k-1}{6}$, `where` k `= filter size.`

`Step 03: To estimate` I_x `and` I_y `we convolved Scharr filters on the obtained images.`

$$\text{scharr\_x} = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}, \ \text{scharr\_y} = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

And to compute I_t, we subtract image at t from image at $t + 1$.

`Step 04: We initialized horizontal component of velocity array,` $\mathbf{u}$, `(which consists of velocity information for each pixel along x-direction) to zeros. Same goes for` $\mathbf{v}$, `the vertical component of velocity.`

`Step 05: For each interest point (i.e., a feature point) detected through Harris corner detection or Shi-Tomasi,`

> `Step 05.1: Pick a window around it whose size is defined by user` $\longrightarrow$ `check if that window is totally contained in the image.`

Step 05.2: Corresponding to this window location obtain all the values that can be covered through this window in I_x, I_y, and I_t and flatten it.

Step 05.3: Construct A matrix by stacking up the flattened I_x and I_y horizontally, and denote $b =$ flattened subwindow of I_t. Compute $\begin{bmatrix} u \\ v \end{bmatrix}$ from $(A^T A)^{-1} A^T b$

Step 05.7 Update the corresponding location in **u** and **v** matrix with the values obtained above.

Let's see the results (OF) we obtain on the data (`Grove3, Urban2, and RubberWhale`) provided to us.

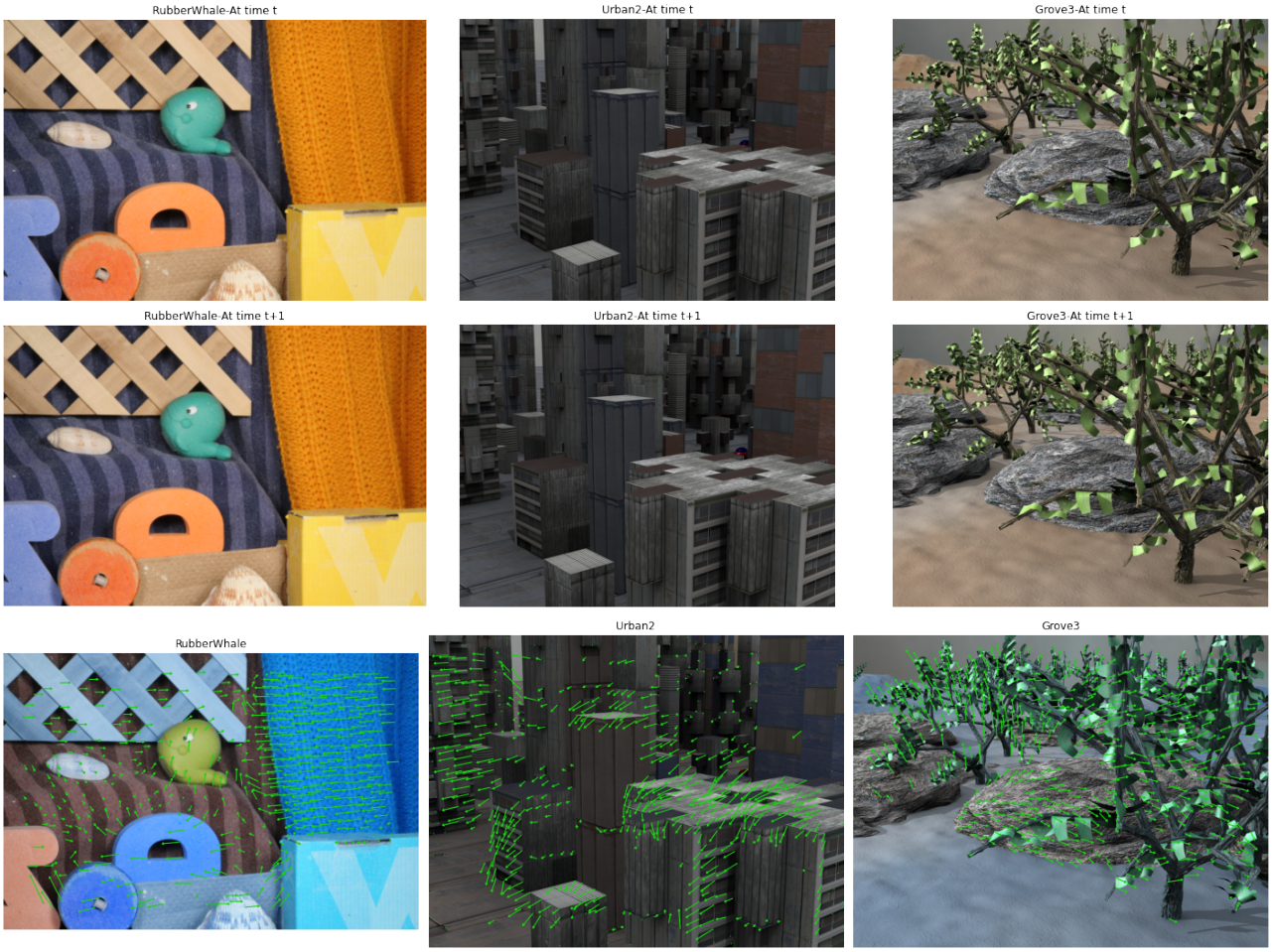

Figure 2: Optical Flow denoted in Green arrows at bottom

Further Analysis:

We've clearly mentioned the effect of increasing/decreasing the window size in Part 3 of Question 2.3 (in `solution.pdf`). Let's see the results here.

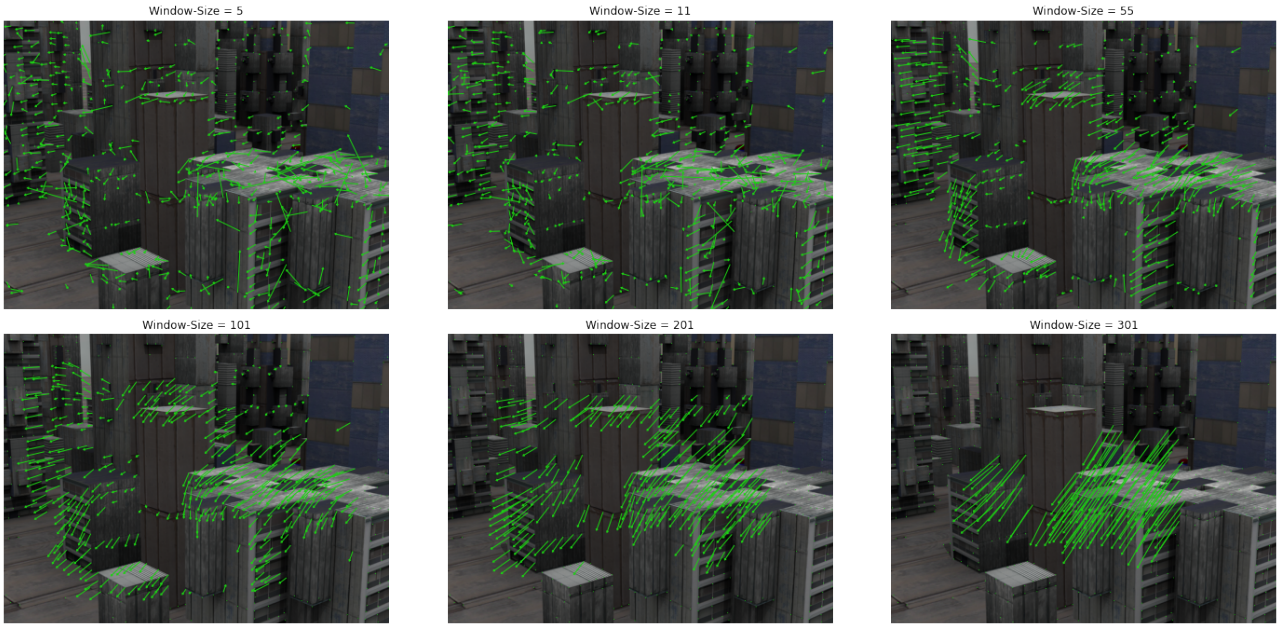

Figure 3: For different window size, how OF varies

Now why window-size = 55 serves the purpose better is due to the reason that the actual displacement happened for corner points for two subsequent images is close to **50** pixels. While for bigger window sizes all corner points show a uniform direction as majority of corner points point towards left-bottom corner that subdue the effect of current centre pixel's actual motion while estimating for velocity.

Further we experimented with τ with values ranging from 10^{-4} to 1.5. And the results we got are all similar which in itself is explainable as we have chose corner-like points (i.e., sparse feature points from Shi-Tomasi detector) whose both eigen values can never get nearer to zero (i.e., always stays positive, even > 1).

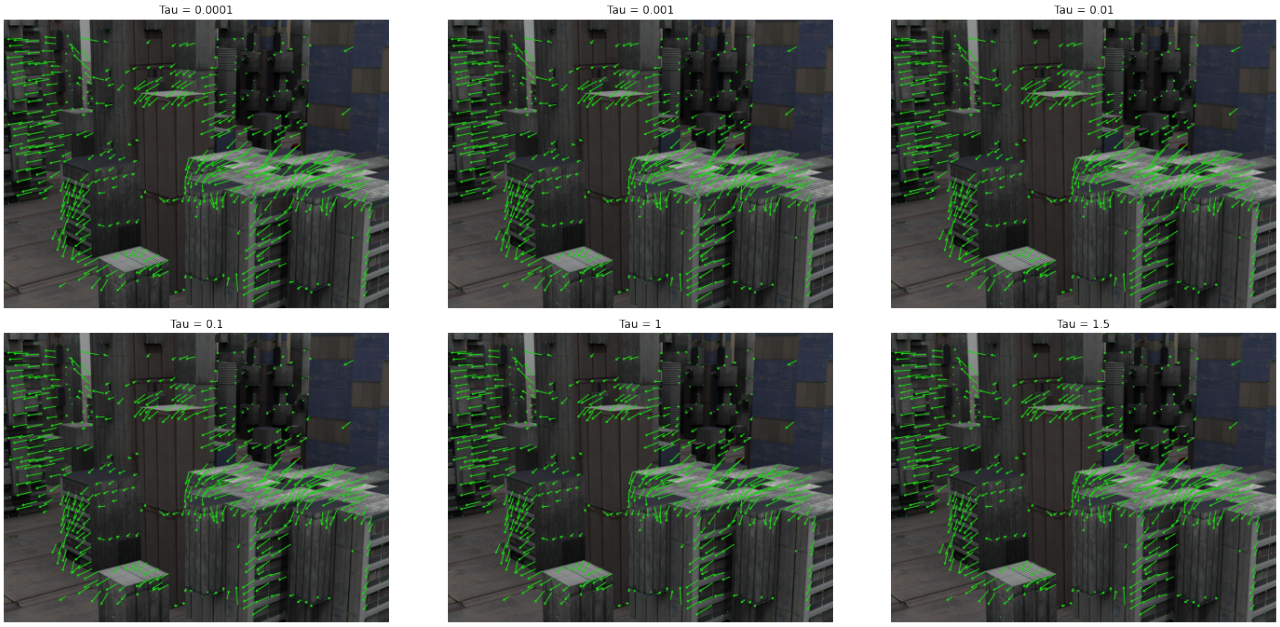

Figure 4: OF with different τ

Question 3: Multi-Scale Coarse-to-fine Optical Flow

Modify your algorithm to iteratively refine the optical flow estimate from multiple image resolutions. The basic idea is to initially compute the optical flow at a coarse image resolution. The obtained optical flow can then be upsampled and refined at successively higher resolutions, up to the resolution of the original input images. By computing the optical flow in this manner, we can circumvent the aperture problem to some degree – because the window size remains fixed across all resolutions, the algorithm effectively computes the optical flow using

successively smaller apertures. This approach also works well on images with large displacements, something the single-scale algorithm is unable to handle.

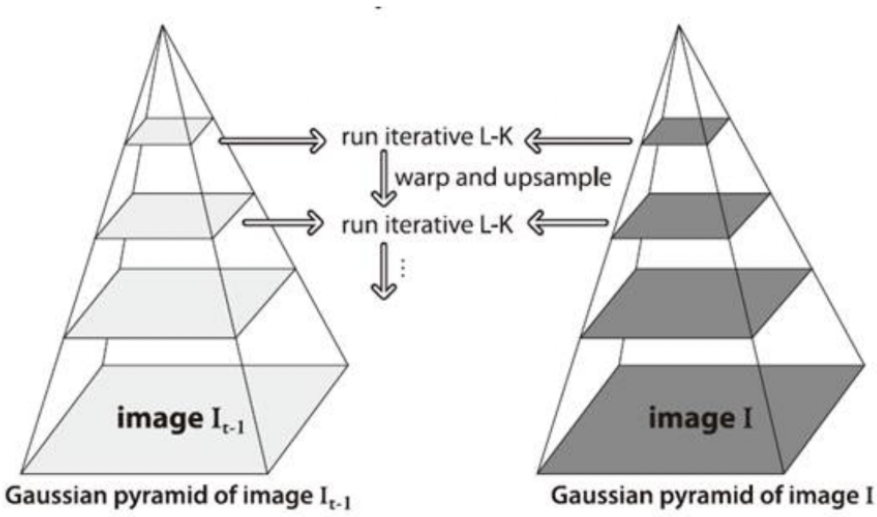

Figure 5: Multi-scale coarse-to-fine optical flow estimation.

In terms of implementation, first modify your code from part two so that it accepts and refines parameters u_0 and v_0, which corresponds to the initial estimates of the optical flow,
`OpticalFlowRefine(Img1, Img2, windowSize, u0, v0)`
Once `OpticalFlowRefine` is working, you should write another function,
`MultiScaleLucasKanade(Img1, Img2, windowSize, numLevels)`
which calls the refinement function. `MultiScaleLucasKanade` should implement the following pseudo-code,

```
Step 01.  Gaussian smooth and scale Img1 and Img2 by a factor of 2^(1 - numLevels).

Step 02.  Compute the optical flow at this resolution.

Step 03.  For each level,

  a.  Scale Img1 and Img2 by a factor of 2^(1 - level)
  b.  Upscale the previous layer's optical flow by a factor of 2
  c.  Compute u and v by calling OpticalFlowRefine with the previous level's optical flow
```

Run the algorithm on the image sequences from the Middlebery dataset. Compare against part two and analyze the results. Show a case where multi-scale performs better than single-scale. This method is based on the 2001 article, Pyramidal Implementation of Affine Lucas Kanade Feature Tracker, by J. Bouguet.

Algorithm for `OpticalFlowRefine`:
(*Note: All the steps in this algorithm will be same except for some steps*)

`Step 01:` Normalize the gray scale images by dividing it with $2^8 - 1 = 255$.

`Step 02:` Apply Gaussian blurring on both normalized images with scale $= \sigma = \frac{k-1}{6}$, where k = filter size.

`Step 03:` To estimate I_x and I_y we convolved Scharr filters on the obtained images.

$$\text{scharr\_x} = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}, \quad \text{scharr\_y} = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

And to compute I_t, we subtract image at t from image at $t + 1$.

`Step 04:` We don't initialize any velocity array, rather we take it as user input in terms of **u0** and **v0** which denotes the initial estimate that needs to be rectified.

Step 05: For each interest point (i.e., a feature point) detected through Harris corner detection or Shi-Tomasi,

Step 05.1: Pick a window around it whose size is defined by user $\longrightarrow$ check if that window is totally contained in the image.

Step 05.2: Corresponding to this window location obtain all the values that can be covered through this window in I_x, I_y and flatten it. But for temporal-difference term (i.e., b =) we take Image2 patch at a shifted location (i.e., $(x + u0, y + v0)$) − Image1 patch at (x, y).

Step 05.3: Pre-compute A matrix. For $i = 1$ to Total iterations defined by user to have a refined (u, v)

Step 05.3.1: Compute $\begin{bmatrix} \partial u \\ \partial v \end{bmatrix}$ from $(A^T A)^{-1} A^T b$

Step 05.3.2: If $abs(\partial u) + abs(\partial v) <$ threshold: then $break$

Step 05.3.3: Update **u0** and **v0** by adding $(\partial u, \partial v)$ and shift the window in Image2 by $(\partial u, \partial v)$.

Step 05.3.4: Update b.

Results time:

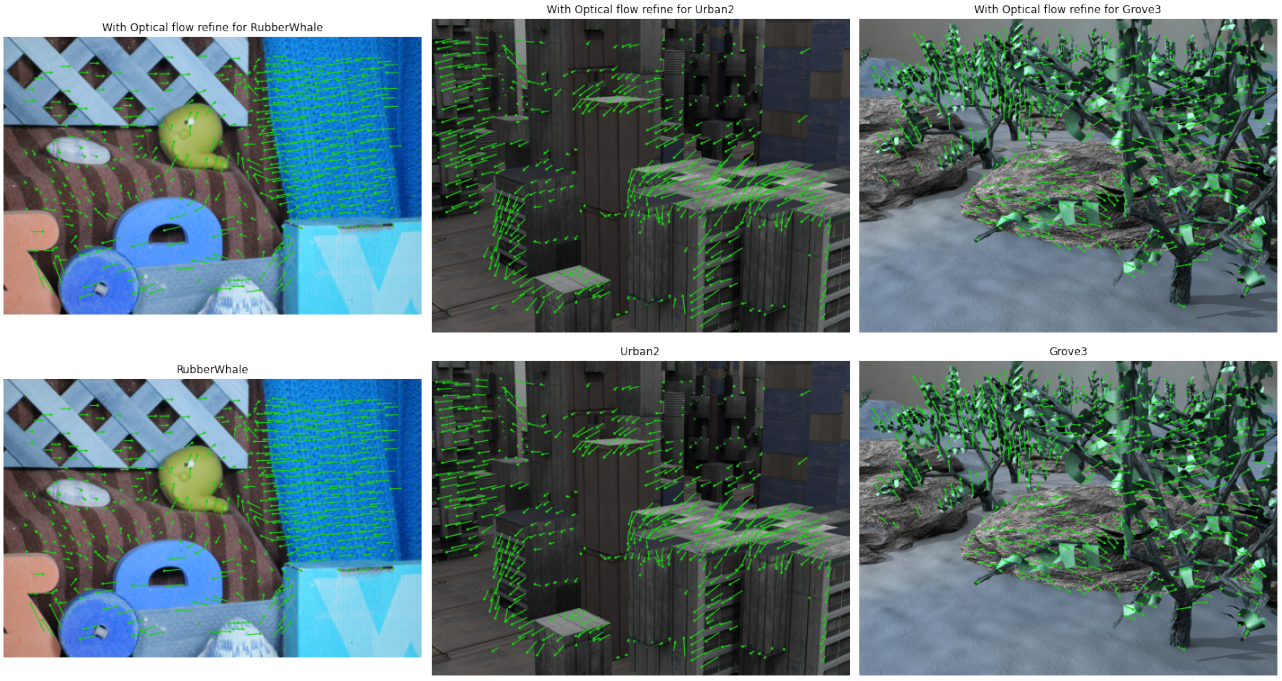

Figure 6: Upper one obtained through OF refine with iteration = 15; while second one is from normal LK

Changing iteration values change the nature of optical flow and since we ourselves taking the responsibility for convergence of OF through iteration, **15** seems to be optimal.

Multi-Scale LK using OF refinement

It's algorithm is clearly mentioned in the question itself, so let's directly hop to the results and compare it with all three.
(*Disclaimer: The results we got for every type of images with three methods are almost comparable. But if one observes minutely an increasing trend in performance can be seen, starting from* `LucasKanade` $\longrightarrow$ `OpticalFlowRefinement` $\longrightarrow$ `MultiScaleLucasKanade`) **Results time:**

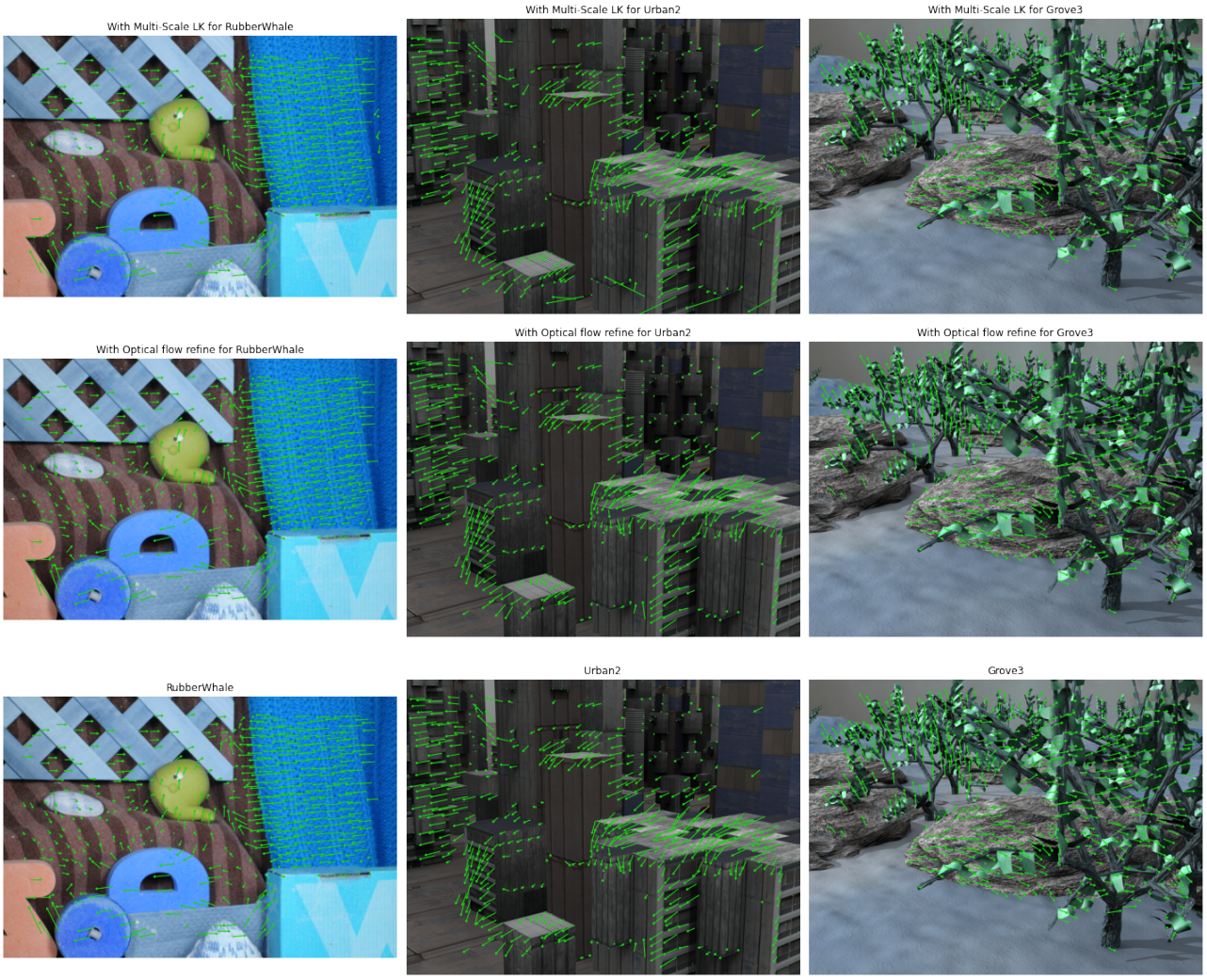

Figure 7: Top: Multi-Scale; Middle: Optical Flow Refine; Bottom: Simple LK

Remarks:

Answer to theoretical questions can be found in `solution.pdf`. Please refer that.