ENPM673 - Perception for Autonomous Robots

Project 4

Lucas Kanade Template tracker

Authors:
Trevian Jenkins (116781381)
Karan Sutradhar (117037272)
Markose Jacob (117000269)

Date: April 20th, 2020

Objective:

Our objective is to implement the Lucas-Kanade (LK) template tracker. Also, we have to evaluate the tracker for three video sequences. Then finally check the robustness of the tracker to illumination.

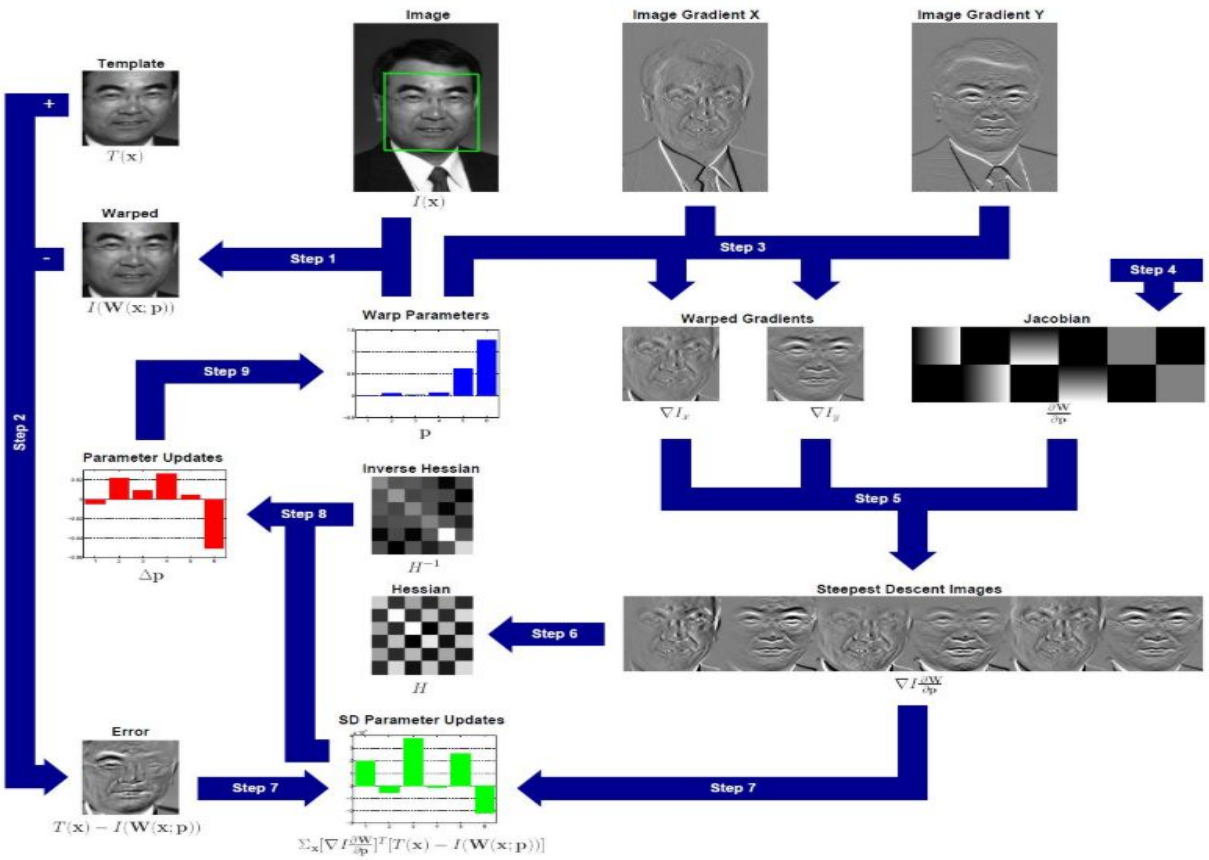
Theory:

The Lucas-Kanade algorithm originally was an image alignment algorithm and can be used for tracking (associate image patch cross frames). The goal of LucasKanade is to align a template image T(x) to an input image I(x), where x = (x ; y) T is a column vector containing the pixel coordinates.
The goal of the Lucas-Kanade algorithm is to minimize the sum of squared error between two images, the template T and the image I warped back onto the coordinate frame of the template:

$$\sum_x [\, I\,(W\,(x;\, p)) - T\,(x)\,]^2$$

Pipeline Followed:

An overview of steps followed in the algorithm is given below.

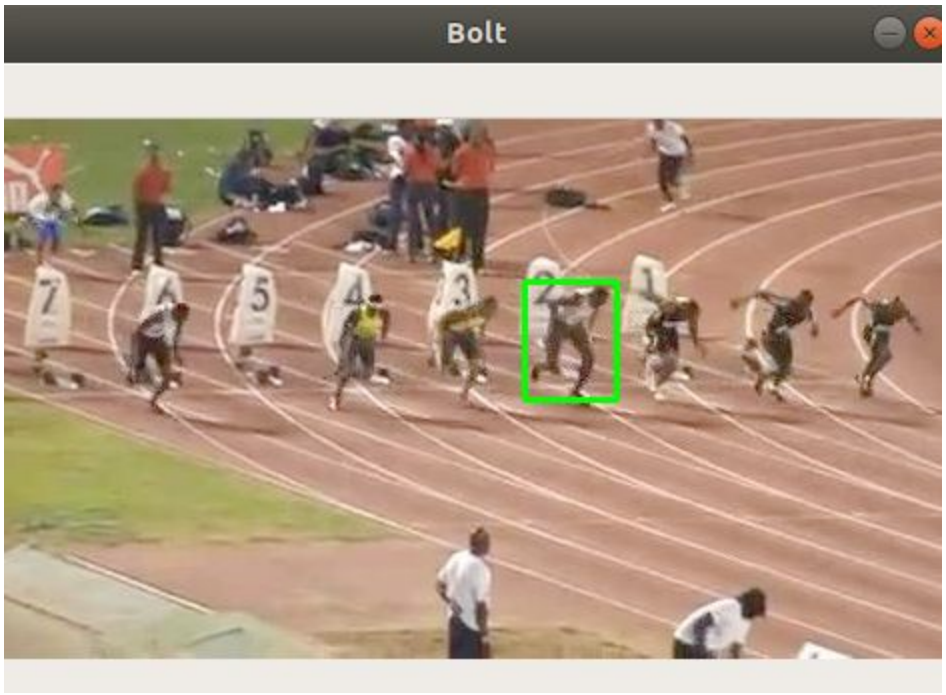


Reference : Lucas-Kanade 20 Years On: A Unifying Framework: Part 1 , Simon Baker and Iain Matthews, CMU-RI-TR-02-16

Method:

Plotting of The Template

The size of the template is decided based on the object which has to be tracked and the template is plotted on the first frame. The coordinates are stored and passed to the core function for every new frame.



Bounding box for Bolt video

Template for Bolt video

STEP 1 : Warping I with W(x; p) to compute I (W(x; p))

The image I is warped with the current estimate of the warp. The result of the wrap is to be subtracted from the template

STEP 2 : Computing the error image T (x) - I (W(x; p))

Here we evaluate the error in the image. Using the coordinates from step one, The portion that has to be tracked is cropped from the first frame and stored as the template.

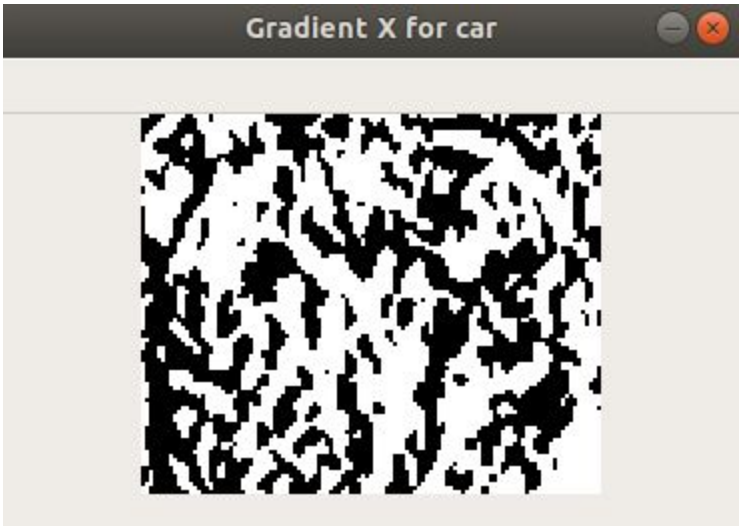
In order to wrap the image we created a numpy array of zeros in the size of the template and then found out the position of the warped template pixel in the current frame. Finally, we copied the values of these pixel locations to the corresponding pixel in the empty numpy array.

The error matrix is calculated by subtracting the warped image from the template image.

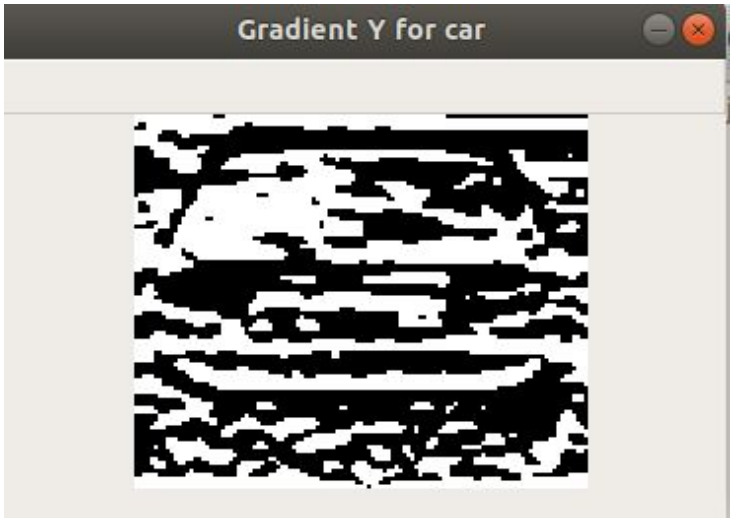
Error for Car video

STEP 3 : Warping the gradient ∇I with W(x; p)

The Gradient along X and Y axis is found using the Sobel filter which has a kernel of size 5. The Gradient is then warped and cropped as per the template.



Gradient along X axis for Car video



Gradient along Y axis for Car video

STEP 4 : Evaluating the Jacobian ($\partial W/\partial p$) at (x; p)

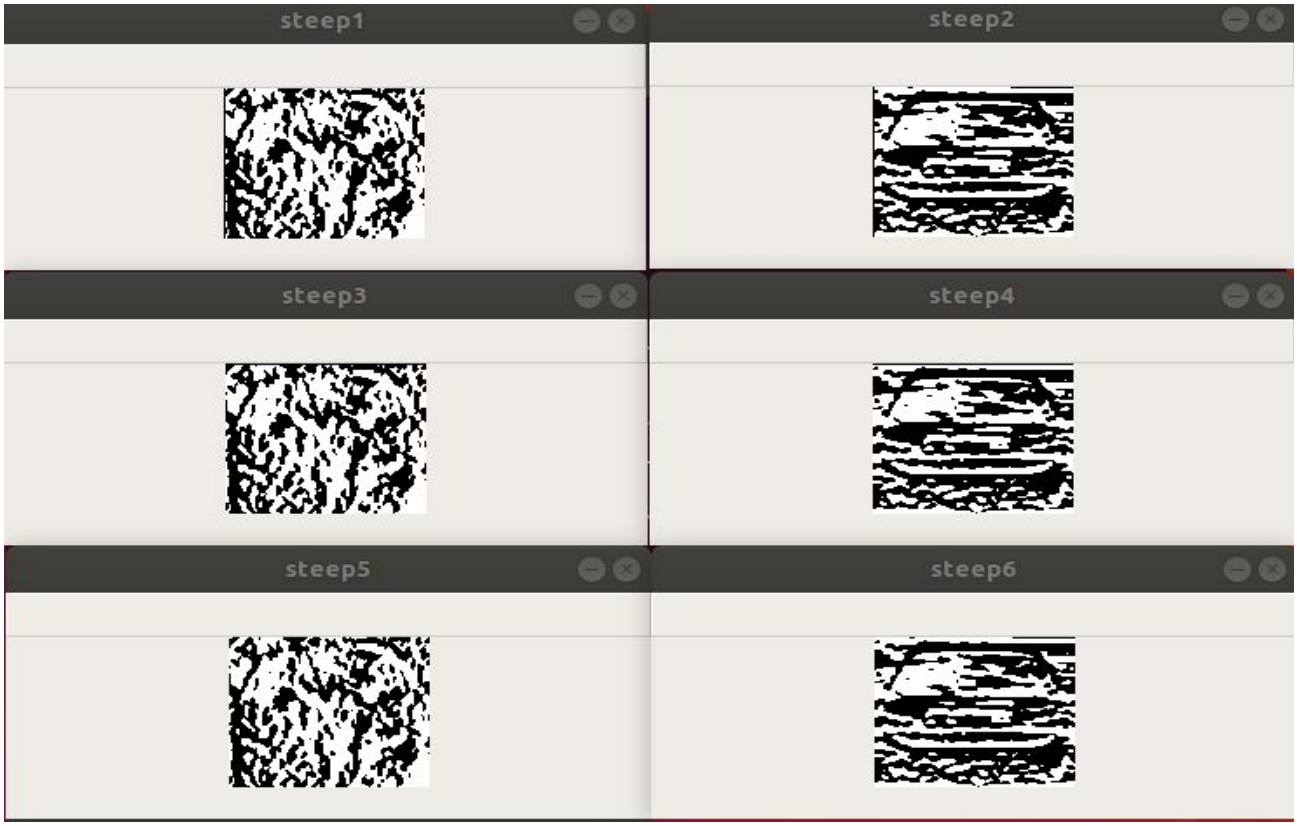
The Jacobian we used is -

$$\begin{bmatrix} j & 0 & i & 0 & 1 & 0 \\ 0 & j & 0 & i & 0 & 1 \end{bmatrix}$$

STEP 5 : Computing the Steepest Descent of the image $\nabla I(\partial W/\partial p)$

The steepest Descent is calculated by multiplying the gradient along X with gradient along Y with the Jacobian matrix which gives us 6 different images in the size of the template.

Basically the two different aforementioned components are combined in this step to compute the steepest descent of the image.



Steep Decent Images for Car video

STEP 6 : Computing the Hessian matrix using Equation as below

$$H = \sum_x [\nabla I(\partial W/\partial p)]^T [\nabla I(\partial W/\partial p)]$$

The Hessian is computed from the steepest descent images.

STEP 7 : SD Parameters Update

Next step is to compute $\sum_x [\nabla I(\partial W/\partial p)]^T [T(x) - I(W(x; p))]$

The steepest descent parameter updates are computed by calculating the dot product between the error image with the steepest descent images.

STEP 8 : Computing Δp

In this step we compute Δp using the equation

$$\Delta p = H^{-1} \sum_x [\nabla I(\partial W/\partial p)]^T [T(x) - I(W(x; p))]$$

The Hessian is inverted and multiplied by the steepest descent parameter updates to get the final parameter, updates the parameter Δp.

STEP 9 : Parameter Updates

Updating the parameters Δp to parameter P.

$$P \dashleftarrow p + \Delta p$$

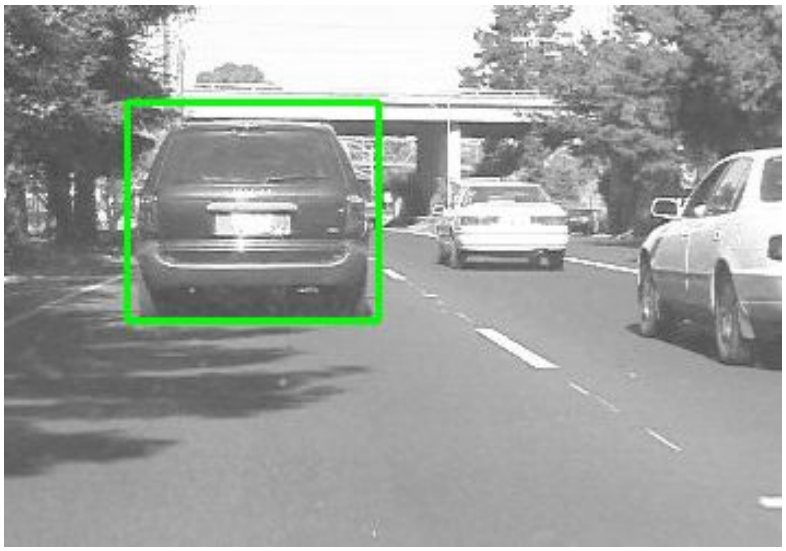
Robustness to Illumination:

To implement this we scaled the brightness of pixels in each frame. This will make the average brightness of pixels in the tracked region and the average brightness of pixels in the template remain the same.

To apply gamma correction, the image pixel intensity is to be scaled from the limit of [0, 255] to [0, 1]. By applying the equation $O = I\ (1/G)$ we get the gamma-corrected image. 'I' is the input image and 'G' is the gamma value. The output image 'O' is then scaled back to the range [0, 255]. Gamma values less than 1.0 will shift the image towards the darker region and gamma value greater than 1.0 will make the image brighter. A gamma value of 1.0 will not affect the input image. We had the best output after applying gamma correction. In our program we tried using different gamma values and found that the value of 3.0 gives the best result. We concluded that gamma correction gives the best output to make the image brighter and hence make the tracker robust against illumination.

Result:

We obtained the desired output as given below:



Snap shot from car video

Snap shot from bolt video

Output Video:

The video output of our program can be found at the link below:
https://drive.google.com/open?id=1-1eIuCFEIfzBZNmDBYJTU2wg4xnQqIkl