NOTE: A large part of my code (which may be relevant for checking) is in .py files in the 'src' folder. Please do not restrict evaluation to .ipynb files. My handwritten/typed answers SECTION WISE can be found in the 2-3 PDFs in the main folder.

Also, I had used function names (different from the ones mentioned in the assignment PDF) in all of the notebooks and changing it to the officially mentioned names might have introduced bugs in case I forgot to replace names even at one place. Hence, please mind the following remapping.

- LukasKanadeForwardAdditive(Img1, Img2, windowSize) IS PRESENT in "lk_helper.py" AS
  - def apply_lk_algo(img_1, img_2, windowSize, sum_mode='avg', recv_sigma_val=-1)
- OpticalFlowRefine(Img1, Img2, windowSize, u0, v0) is PRESENT in "lk_helper.py" AS:
  - def apply_pyramid_lk(img1, img2, windowSize, numLevels)

Contents and some other details regarding directory structure:

- "harris_helper.py" and "lk_helper.py" contain all the interesting important functions used in the assigmment
- "videos' ' of the 3 scenes with quiverplots is in the "./videos" directory.
- The 3 notebooks starting with the word "Harris" are the results of the Harris Corner Detector on the 3 scenes.
  - The main function used is :
    - find_corners(img_array, kernel_x, kernel_y, do_gaussian=False, **algo="harris",** sigma_val=1, k_val=0.04, window_size_for_aggregating=3, window_size_for_suppression=11)
    - THE PARAMETER MEANINGS HAVE been described in the latter part of the report.
    - Psi-Tomasi can be run by simply changing the parameter named "algo" to "shi-tomasi"
- The 3 notebooks starting with "Running Optical Flow" contain the results of the experiments on using different params on the 3 scenes.
- "Comparing naive LK algorithm with Pyramidal implementation.ipynb" has my attempts to find where can the "pyramidal implementation" produce better results
- "lk_method.ipynb": contains just one running instance of the Lucas Kanade algorithm (DOES NOT CONTAIN ANY EXPERIMENTS)
- "lk_pyramid.ipynb": contains just one running instance of the PYRAMIDAL IMPLEMENTATION of the Lucas Kanade algorithm (DOES NOT CONTAIN ANY EXPERIMENTS)
- The folder "./rough_notebooks" contains the notebooks I used to experiment with the official OpenCV functions and to verify the correctness of my own algorithms by talling with the library results

---

Regarding Harris Corner detector

Parameters and Experimental Settings

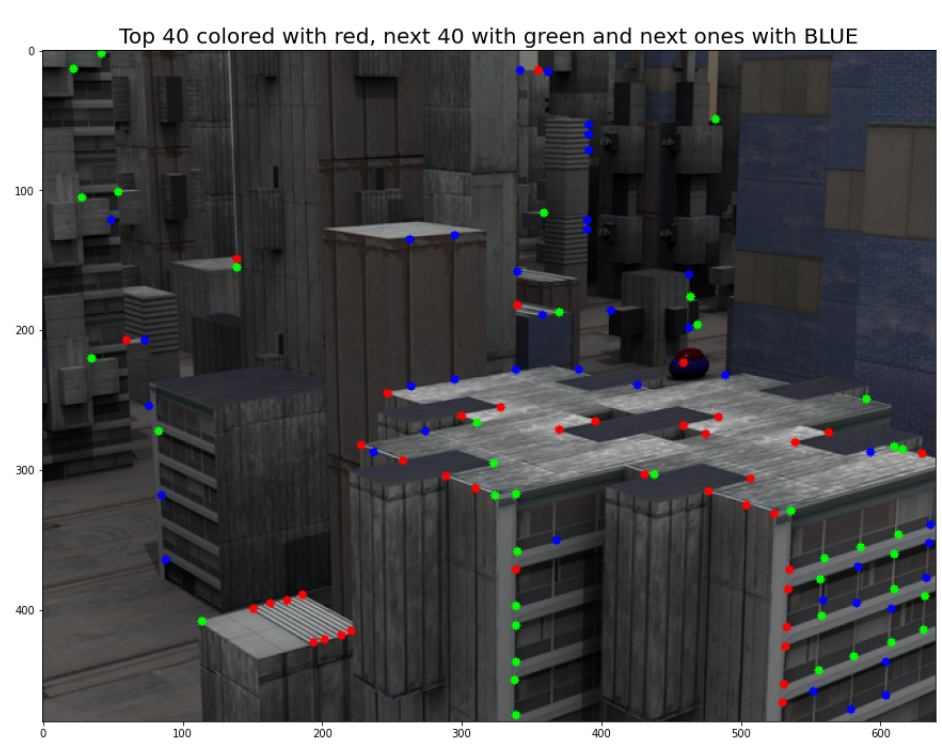
Harris corner detector has a lot of parameters which can be varied. They are:-

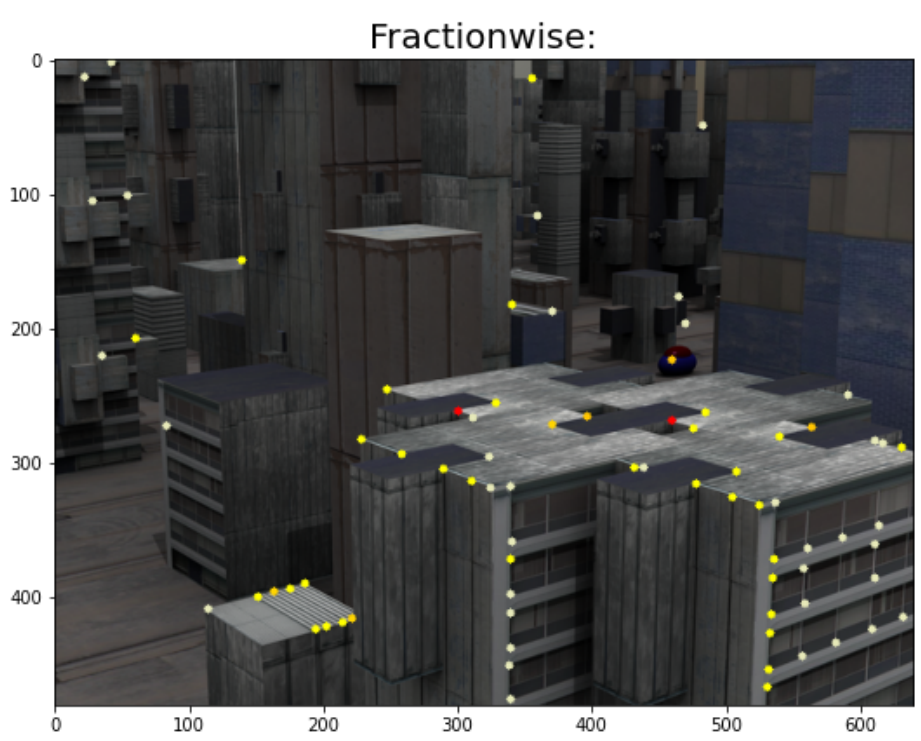
- The choice of the Sobel operator used for calculating gradients for a pixel
- The block size (neighboring pixels included) while calculating the gradient covariance matrix for a pixel
- Method used while calculating the gradient covariance matrix (gaussian contribution or uniform contribution).
  - As per this link, OpenCV implementation seems to give equal weightage to all neighbors and the center pixel.

    - The paper by Harris recommends a Gaussian weightage instead
- k ie Harris detector free parameter.
- Sigma value to be used (in case Gaussian weightage is assigned while calculating gradients)
- Non-maximum suppression ie Maxima constraint for being a corner: As per the official paper, a pixel is to be a designated a corner only if it's E(x,y) is maximum in a 3 x 3 neighboring square
- *What sort of value threshold to be kept on E(x,y) for a point to be a corner*
    - Threshold can be absolute for all images
    - Or, it can depend on the highest E(x,y) of the entire image
    - Or, we can always display the top "n" pixels with the highest scores

I experimented with variations in almost all of the above parameters.

Below: Window size for suppression=11, gaussian=True, window for Gaussian = 3 pixels, k_val=0.04





In above image, red points are the top 40 points, the green ones are the next 40 and the blue ones are ranked 80-120. It is evident that:

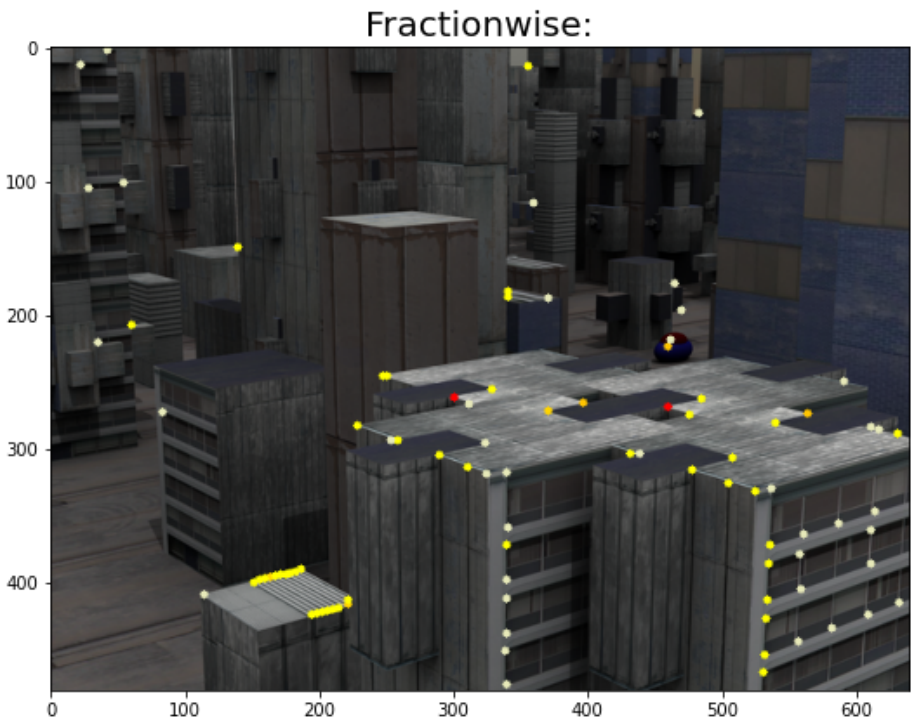
- Red points include both corners and points on edges.
- Green points also include corners and edges
- Blue points are mostly on edges

In the right image, points are colored on basis of relative score to the highest rated pixel. Red = close the highest rated pixel, WHITE=large difference in score, YELLOW=between red and white

- We see that on a score basis, even though points on edges rank well, their relative score when compared to HIGHEST RATED PIXEL is small.

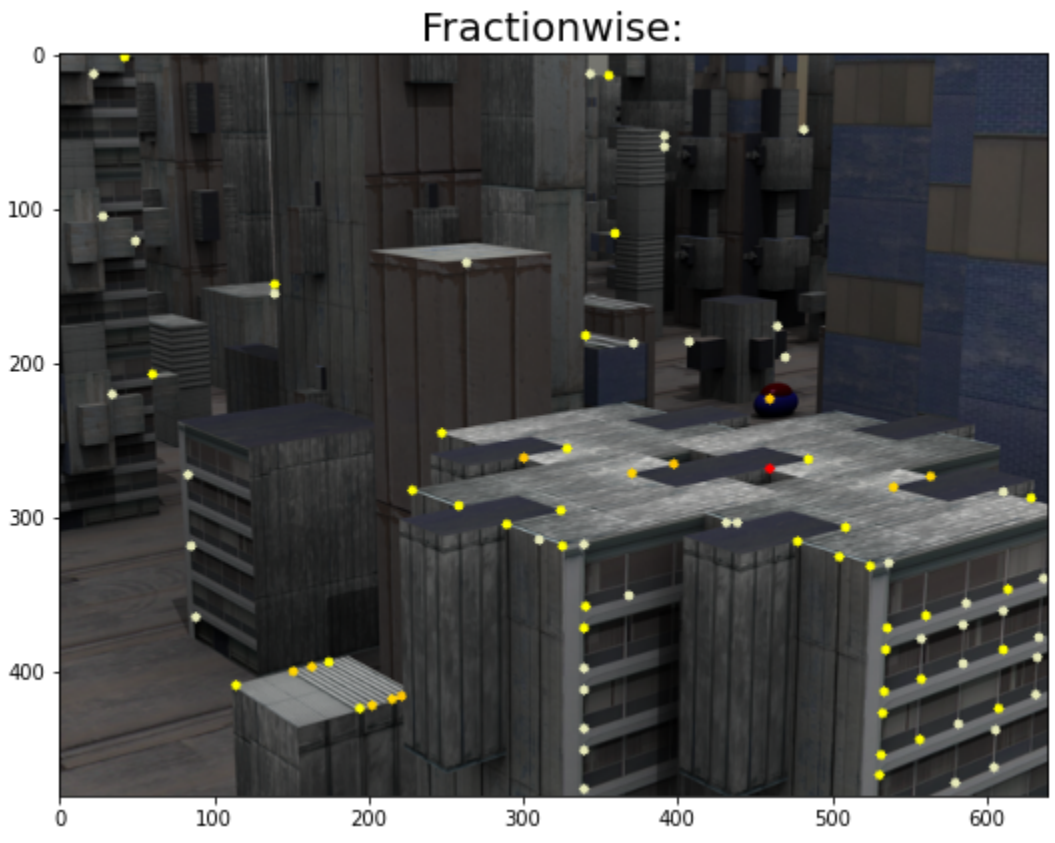
Decreasing window size for maximal suppression from 11 to 3:

The number of points in neighborhood of each other increased (edge points adjacent to a corner).



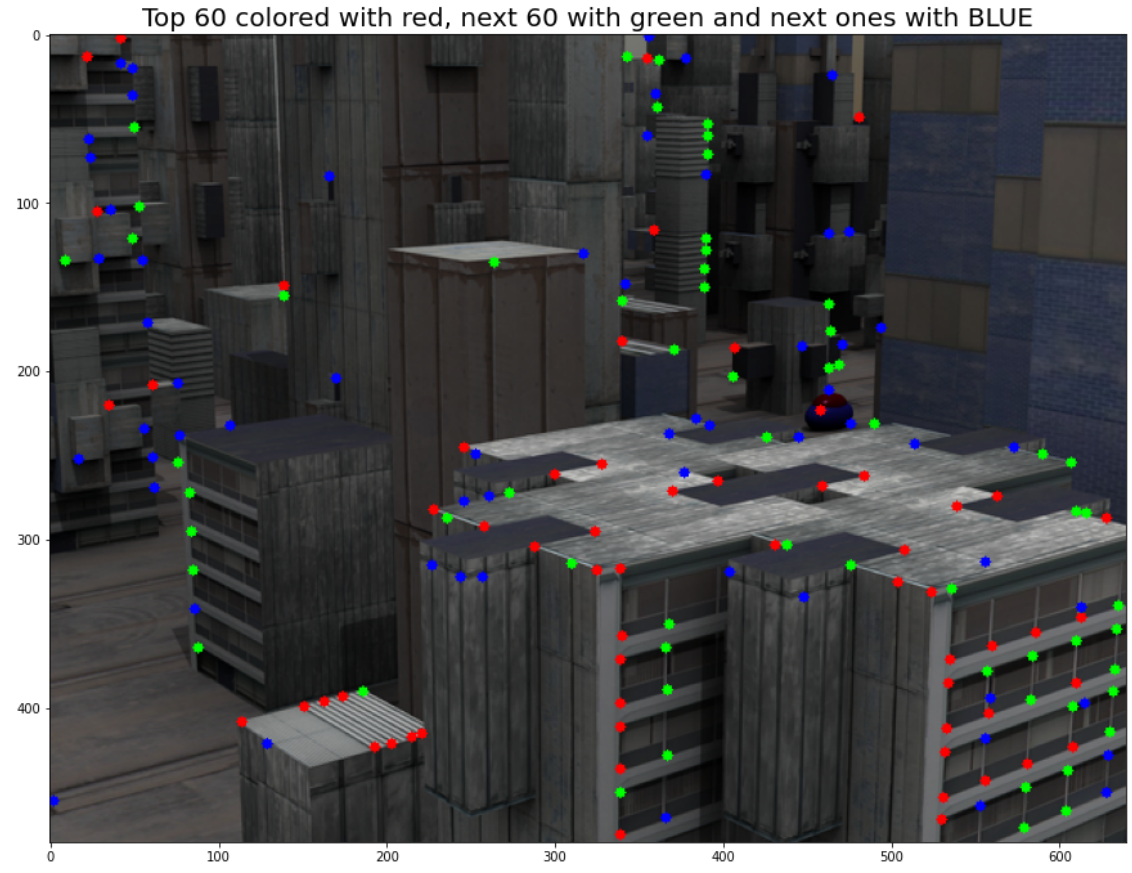
Increasing K value from 0.04 to 0.1:

No major change observed. It seems that the score of the edge points decreased slightly though.



Increasing SIGMA value in Gaussian smoothing from 1 to 20

No major changes observed.



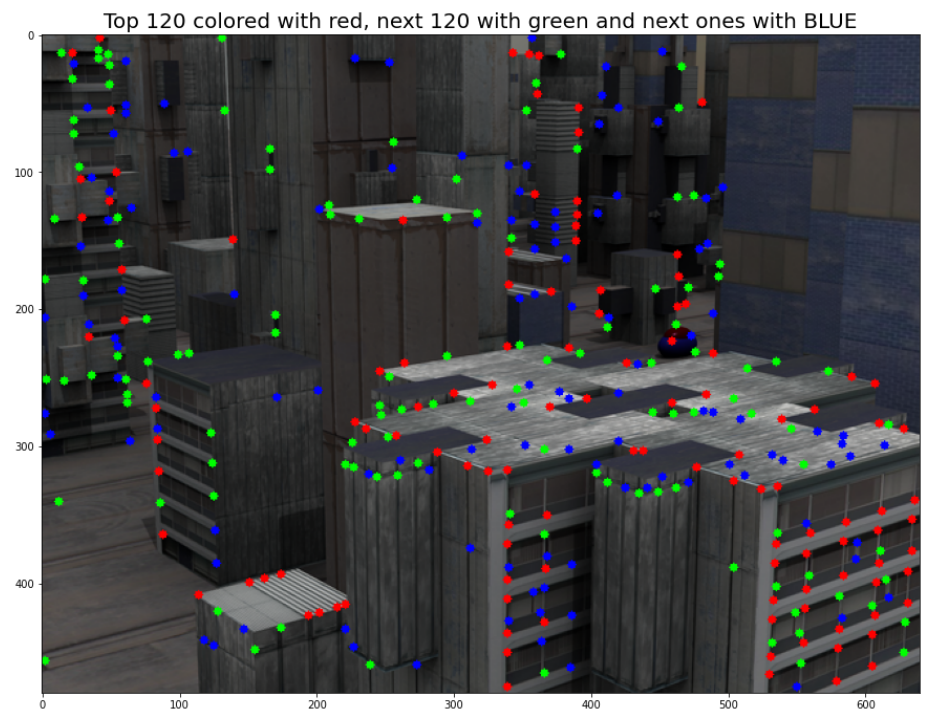
Comparison with Shi-Tomasi

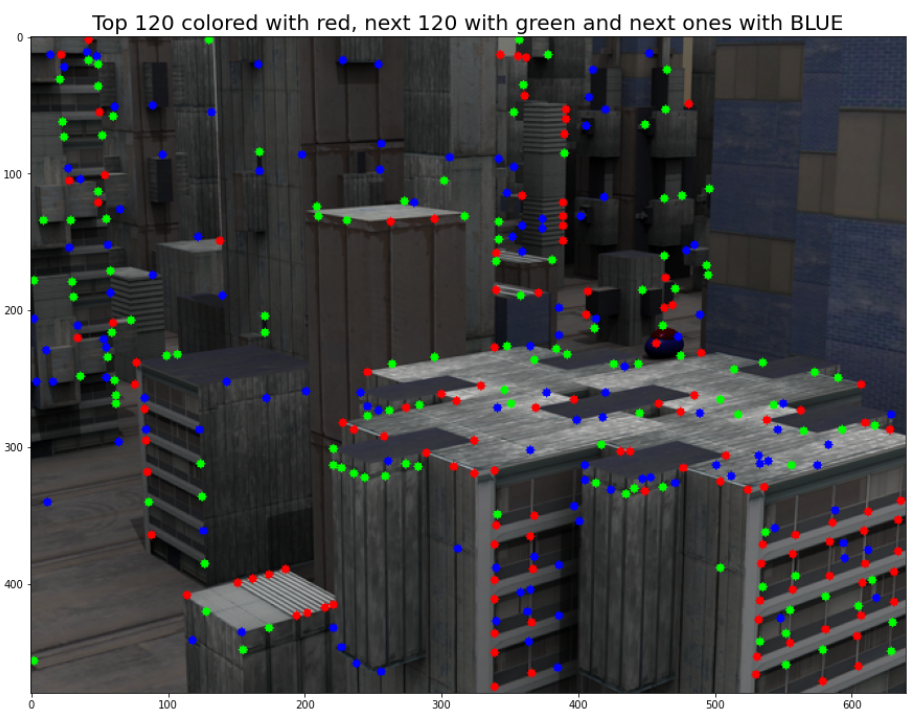
Left=Harris, Right=Psi-Tomasi

Top 120=red

Next 120=green

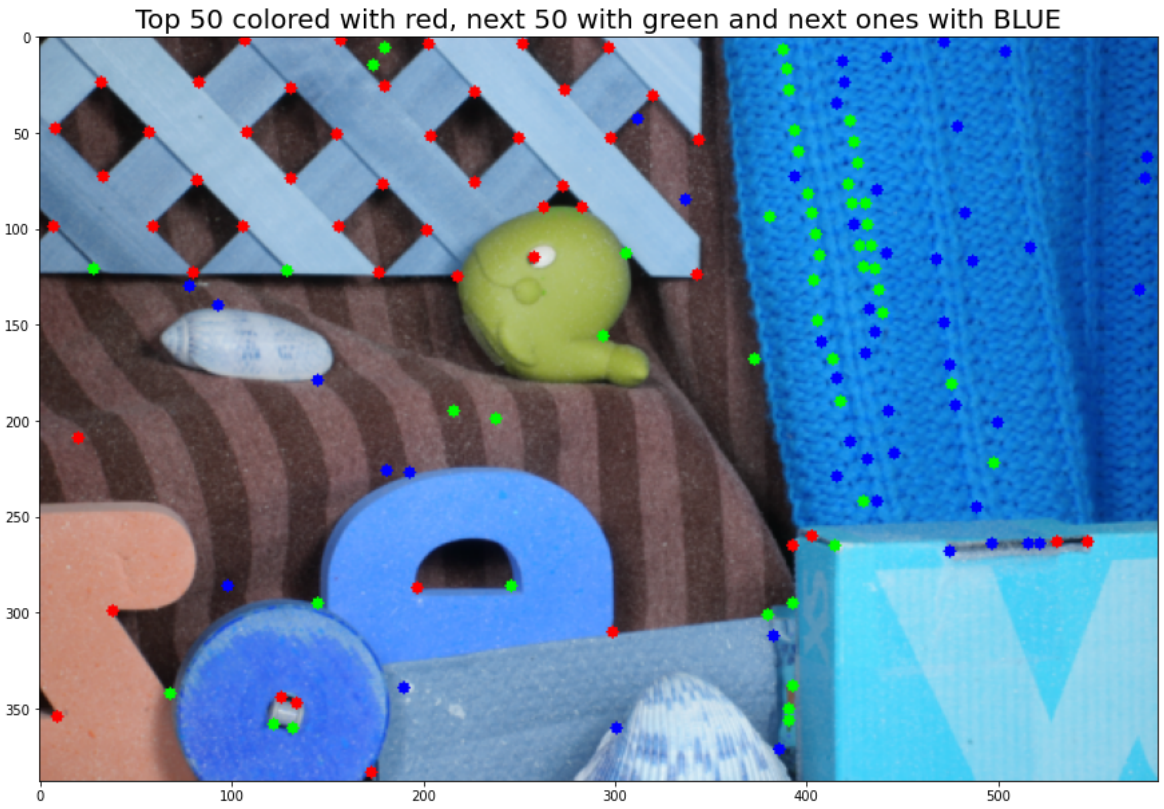
Next 120=blue



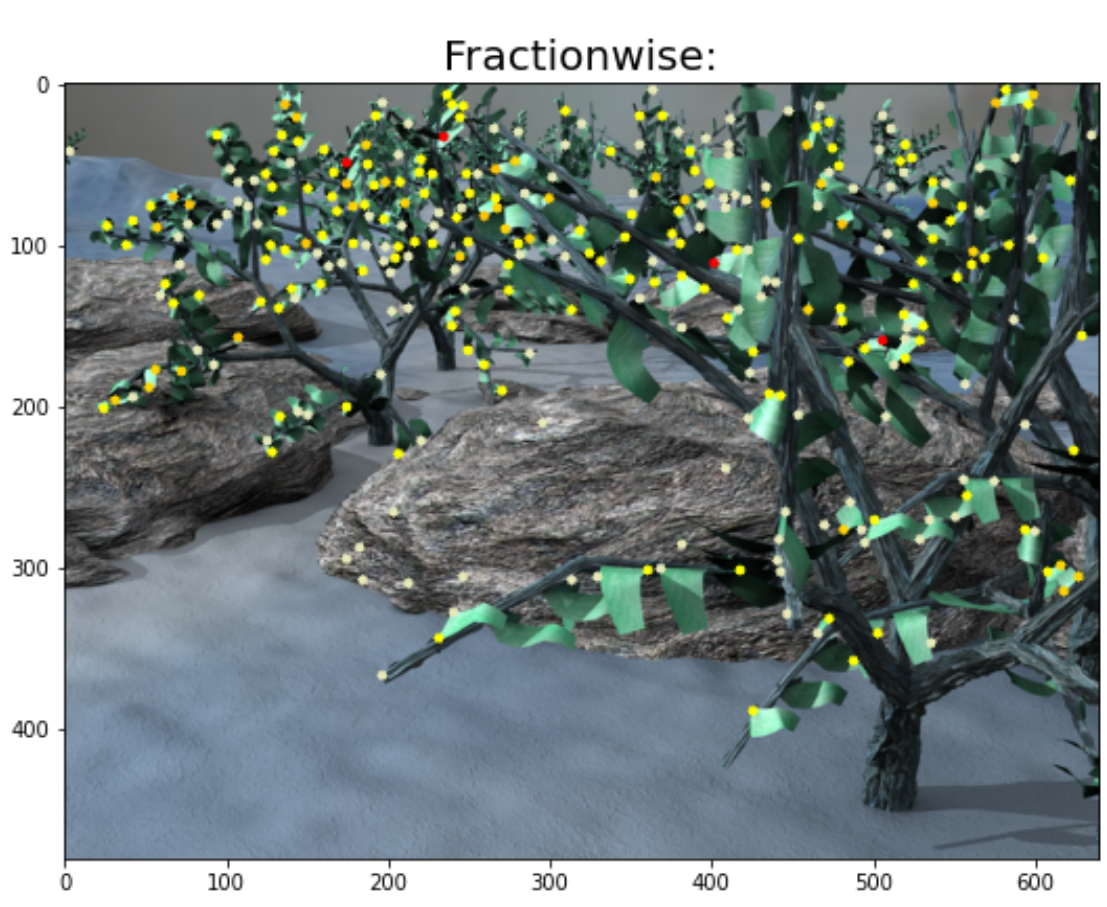


Almost negligible change observed, but surprisingly, Harris seems slightly better (some of the edges which are green in left and red in right. Also, some corners which are red in left and green in right) although this could be only for the specific hyperparams used.

Results for other scenes



Fractionwise:



Discussion

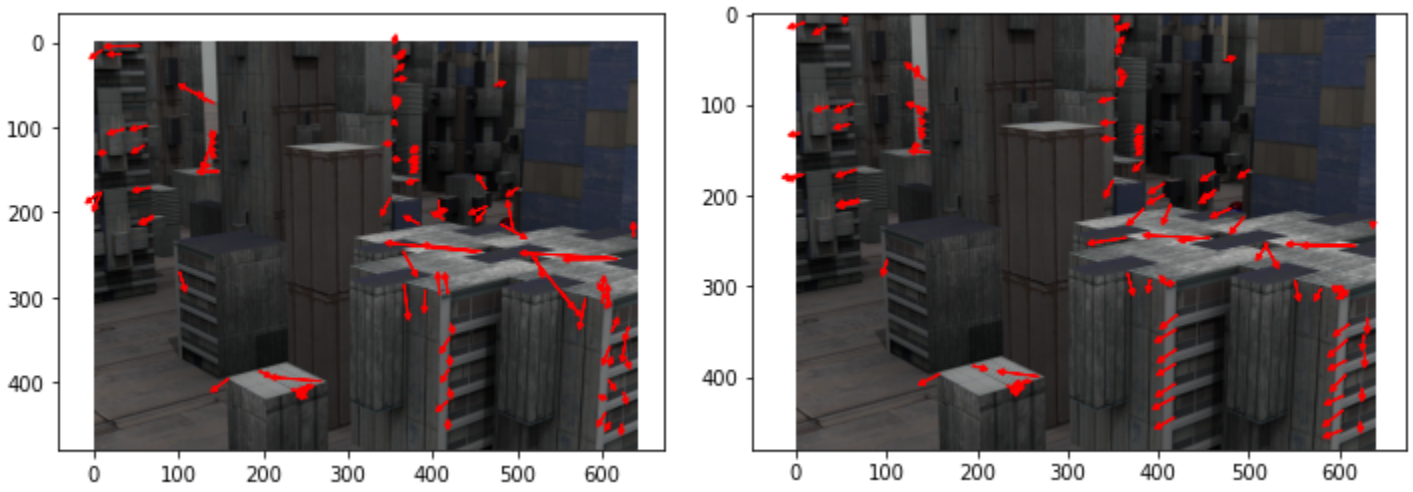
In summary, the Harris corner detector (and the Shi-Tomasi detector) seems to correctly detect the expected corner points in the image. But there are still cases in which the detector classifies points as corners, even though they are in a flat region or as a part of an edge. However, with sufficient experimentation with the hyperparameters (different tailored hyper-parameters for each test-case), I believe reasonable accuracy can be achieved.

Answer to Section 2.3 (Regarding tradeoffs for window sizes)

- Intuitively, a small window would be preferable in order not to ``smooth out'' the details contained in the images (i.e. small values of w_x and w_y). This helps better handle cases where even the pixels in close proximity have different movements. Small => better for accuracy
- Intuitively, a large window helps handle sensitivity and noise with respect to changes of lighting, size of image motion. ANy outliers are neutralized by the effect of their neighboring pixels.

Observations

Increasing the WINDOW SIZE from 5 (left) to 11 (right) bettered the result in case of Urban2



- More detailed analysis has been done on a scene by scene basis

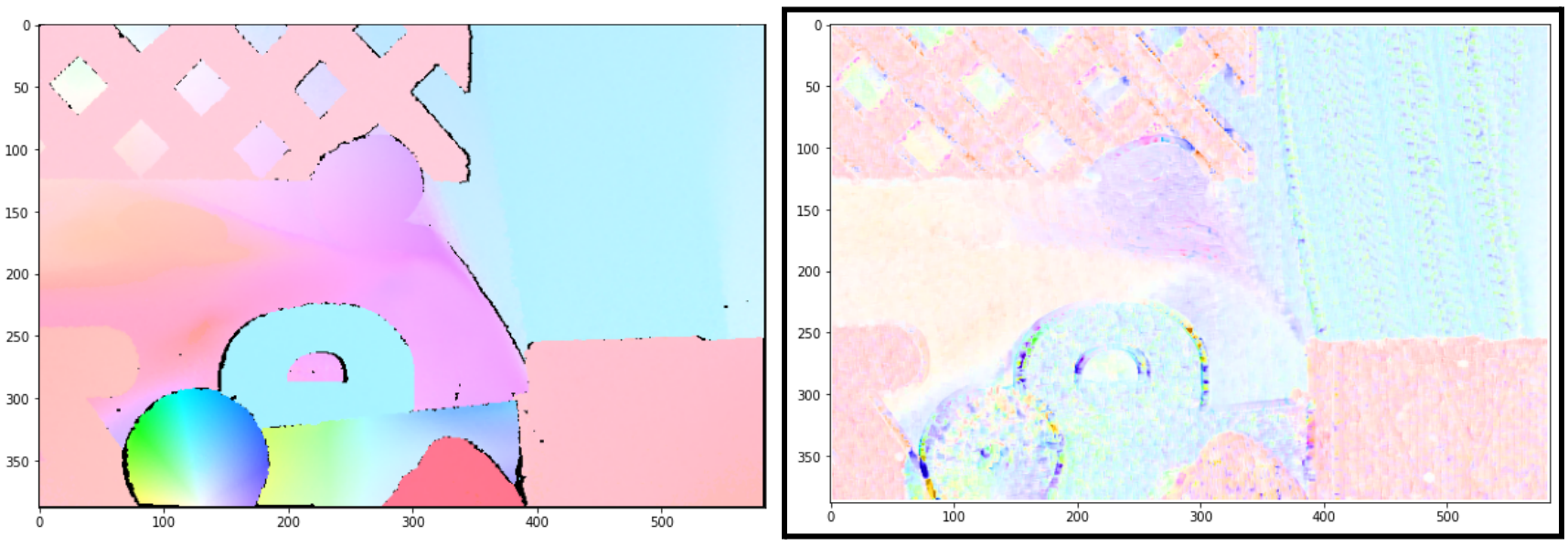
---

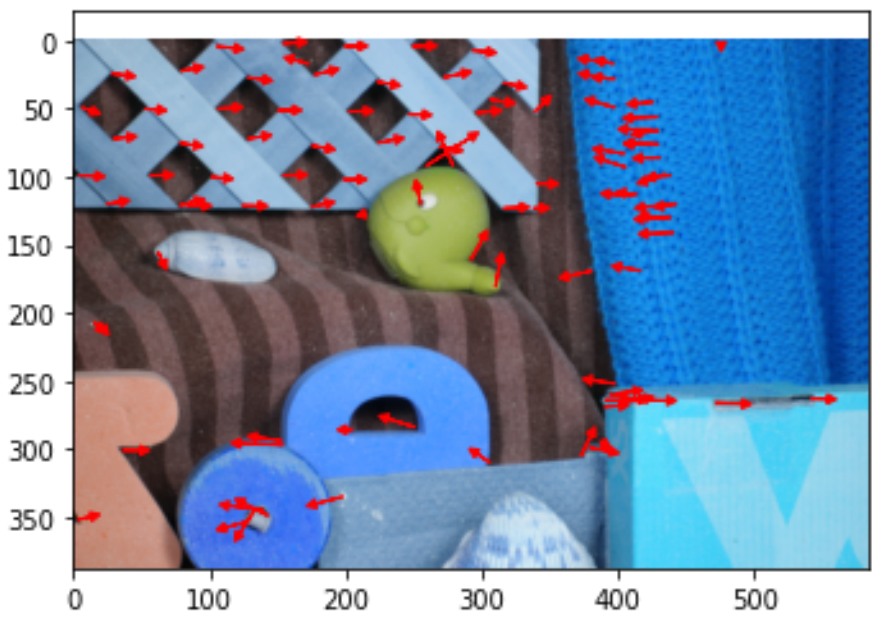
Regarding Lucas Kanade's algorithm:

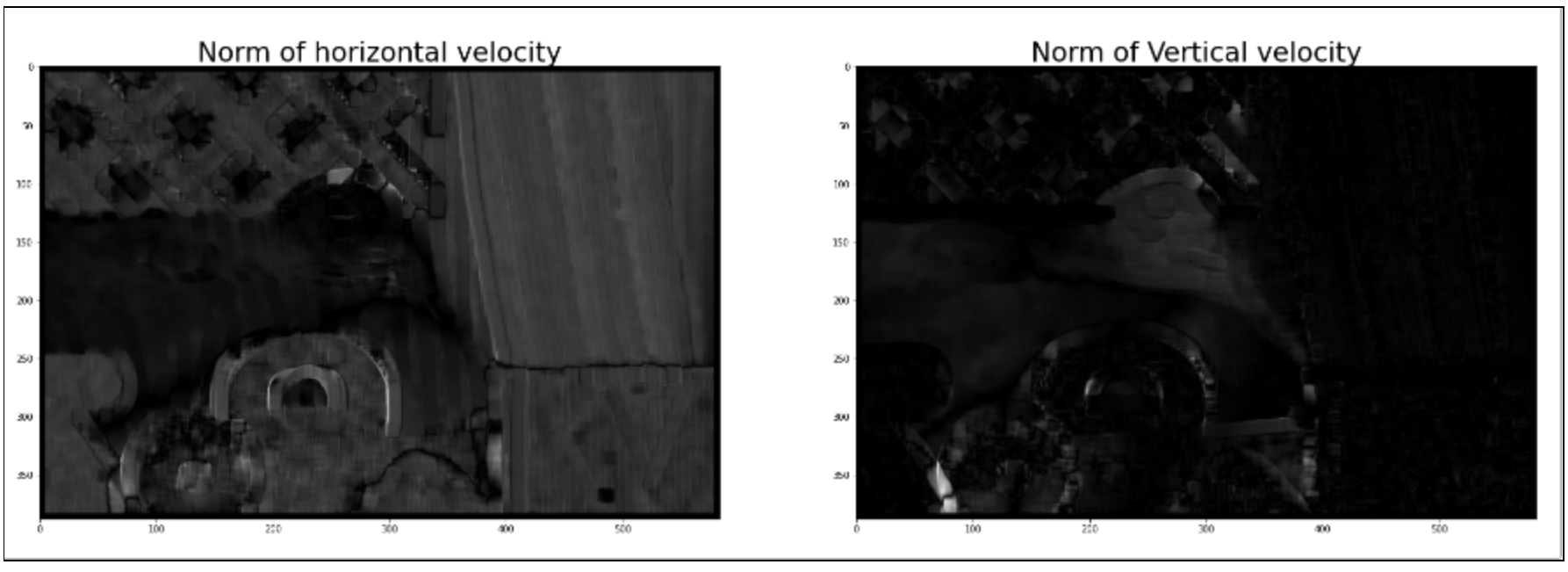
There are a lot of parameters which can be varied. They are:-

- The sobel operator used to calculate the gradient
- Window size over which the summation for the gradient covariance matrix is done
- The weightage/contributions of the different neighbors to the gradient covariance matrix
- Policy to decide sparse points for whom flows needs to be calculated

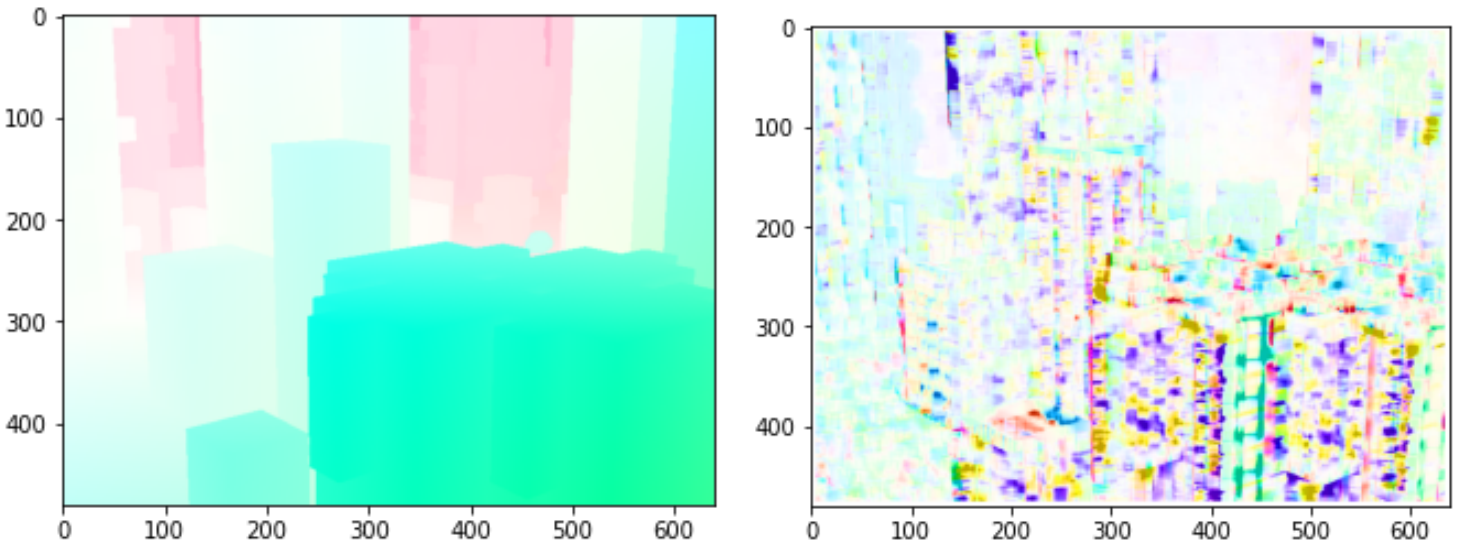
Ground Truth VS Obtained Flow for default params (Avg end point error=1.2 pixels)

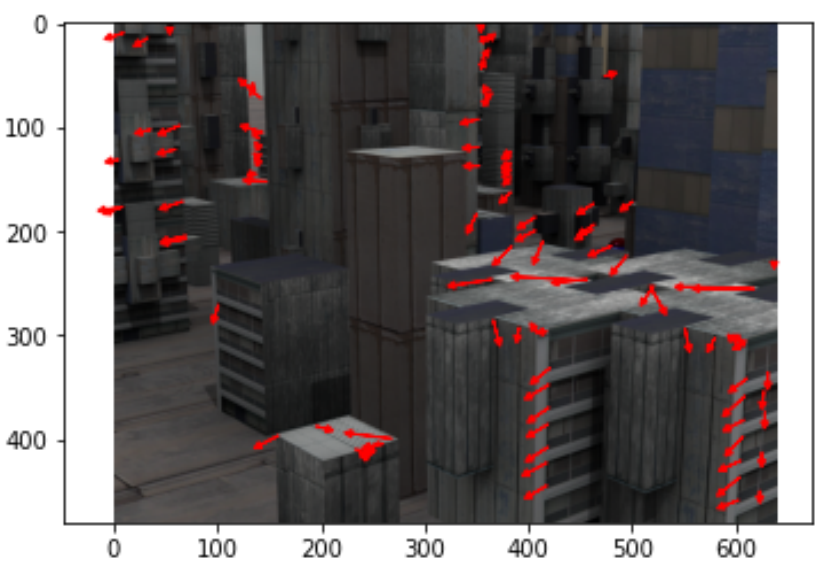


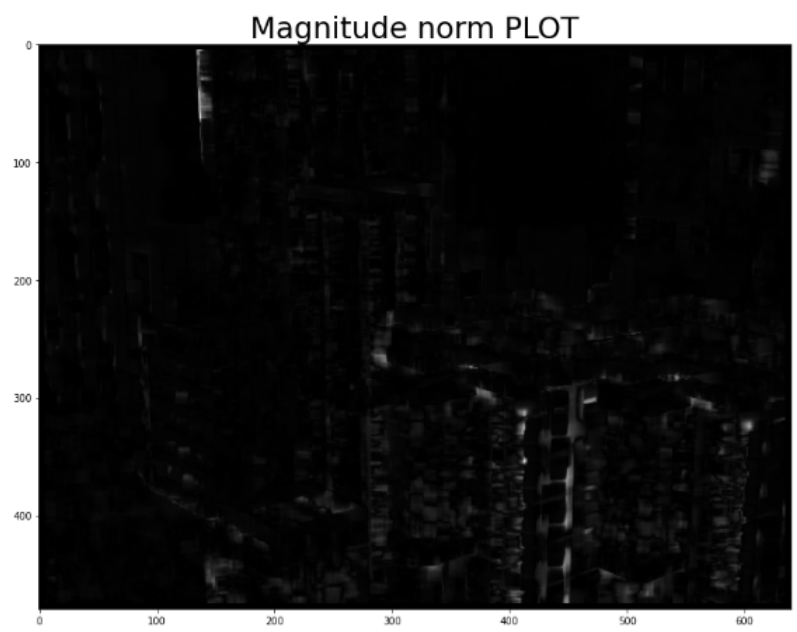




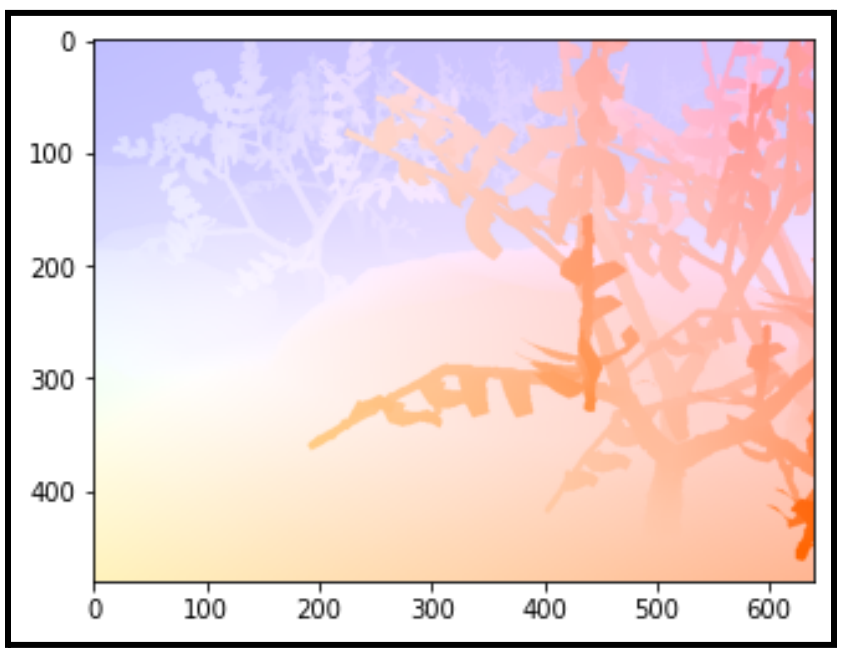
Ground Truth VS Obtained Flow for default params

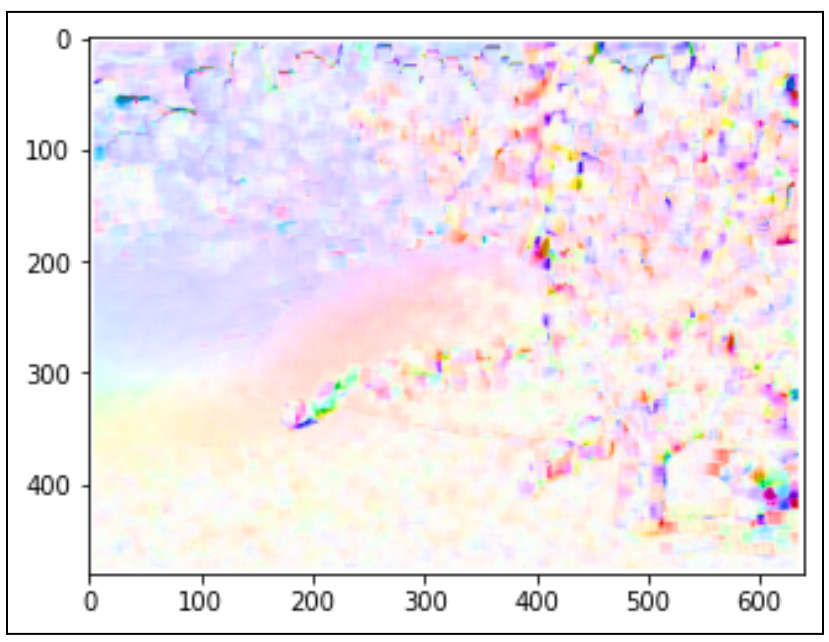


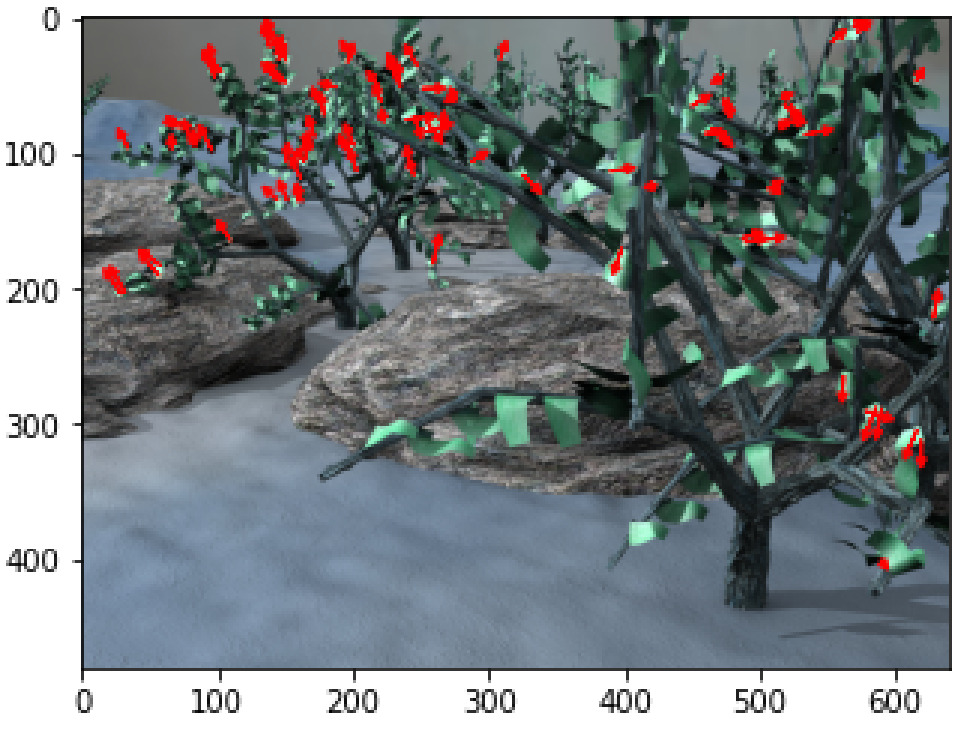




Ground Truth VS Obtained Flow for default params (Avg endpoint error=3.89 pixels)



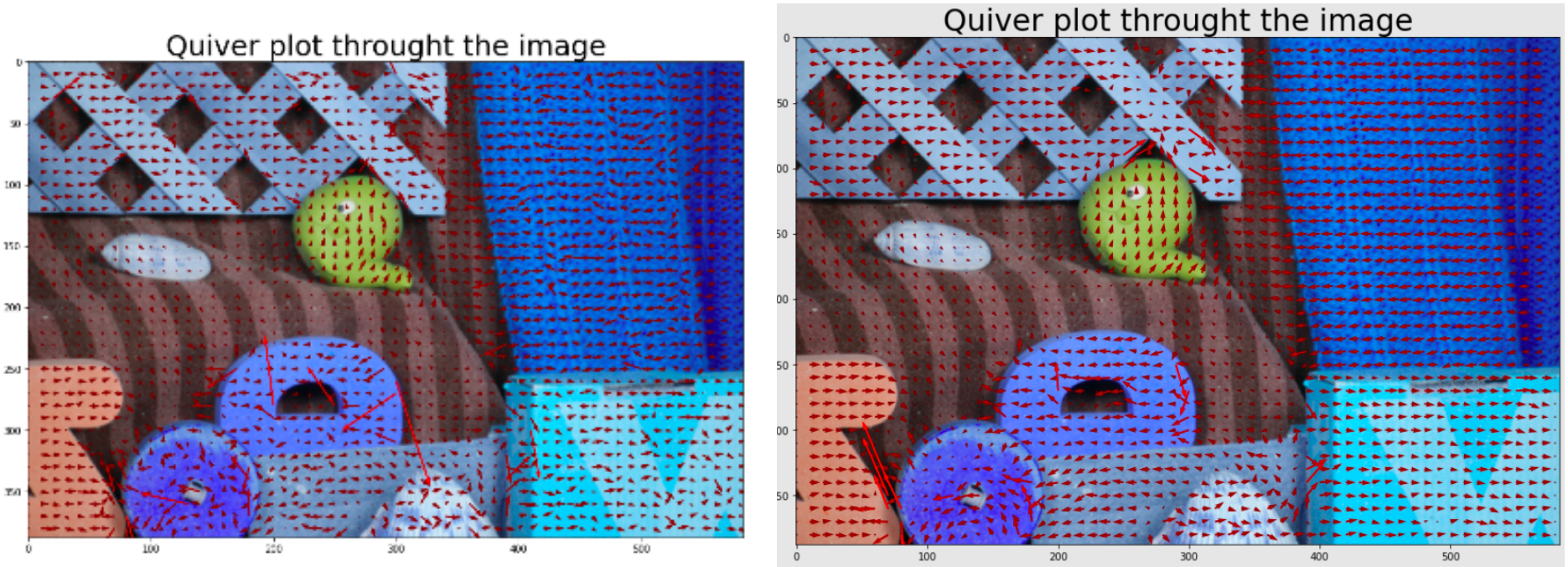




RubberWhale Scenes

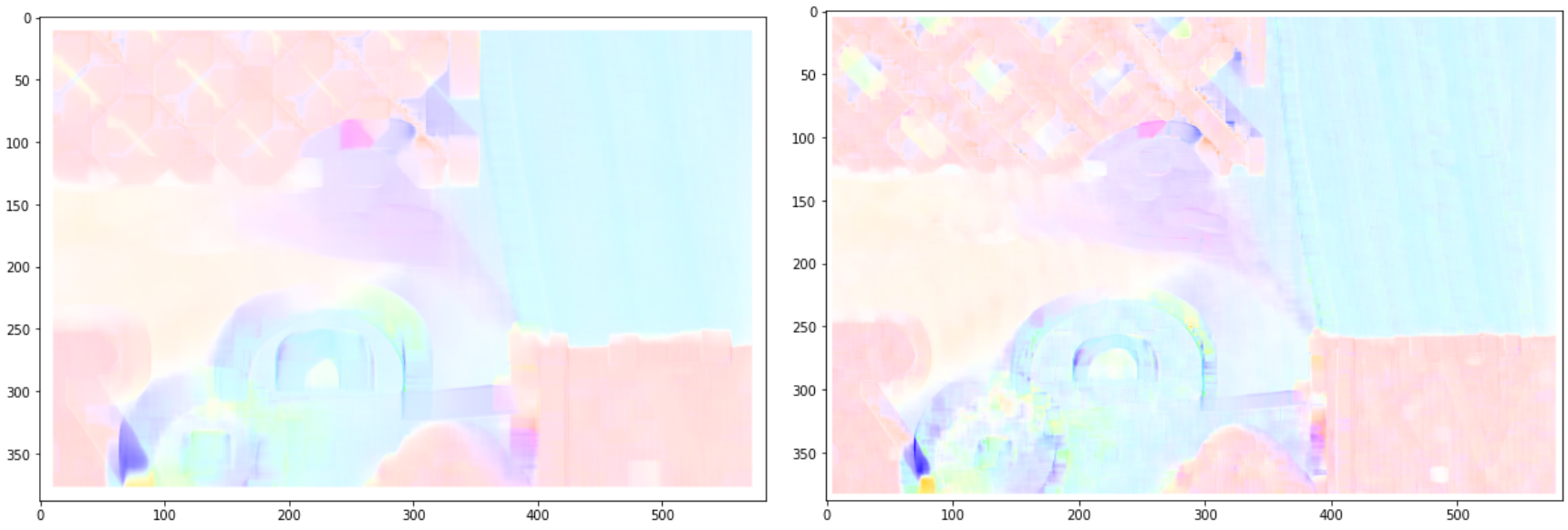
In general, the algorithm works well for this image. The image has a lot of corner points and the motion is also NOT very large. Also, since the objects are large (unlike the leaves in GROVE3), the ASSUMPTION THAT “neighboring pixels have similar optical flow” seems to be justified in this scene.

- Left (window size=3) VS Right (window size=11)

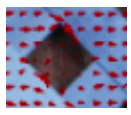


Obs: On the right, we see that optical flow vectors in the neighborhood of pixels point more or less in the same direction (the larger window size with averaged contributions seems to balance out the movement in that specific pixel). Also, on the left, we can see 1-2 large red arrows (potential outliers) but the arrows at the same location to the right are normal (and correct). This shows that keeping a larger window size helps overcome effects of unusual changes in brightness, intensity etc).

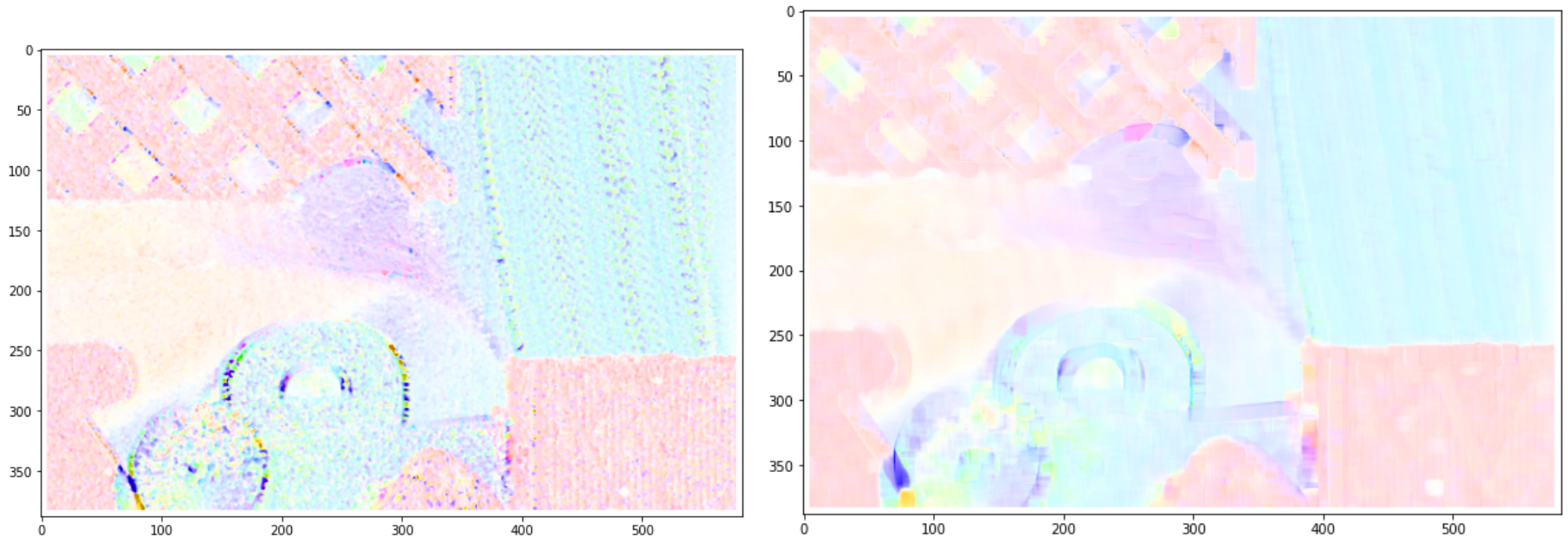
- Left (window size=23) VS Right (window size=11)



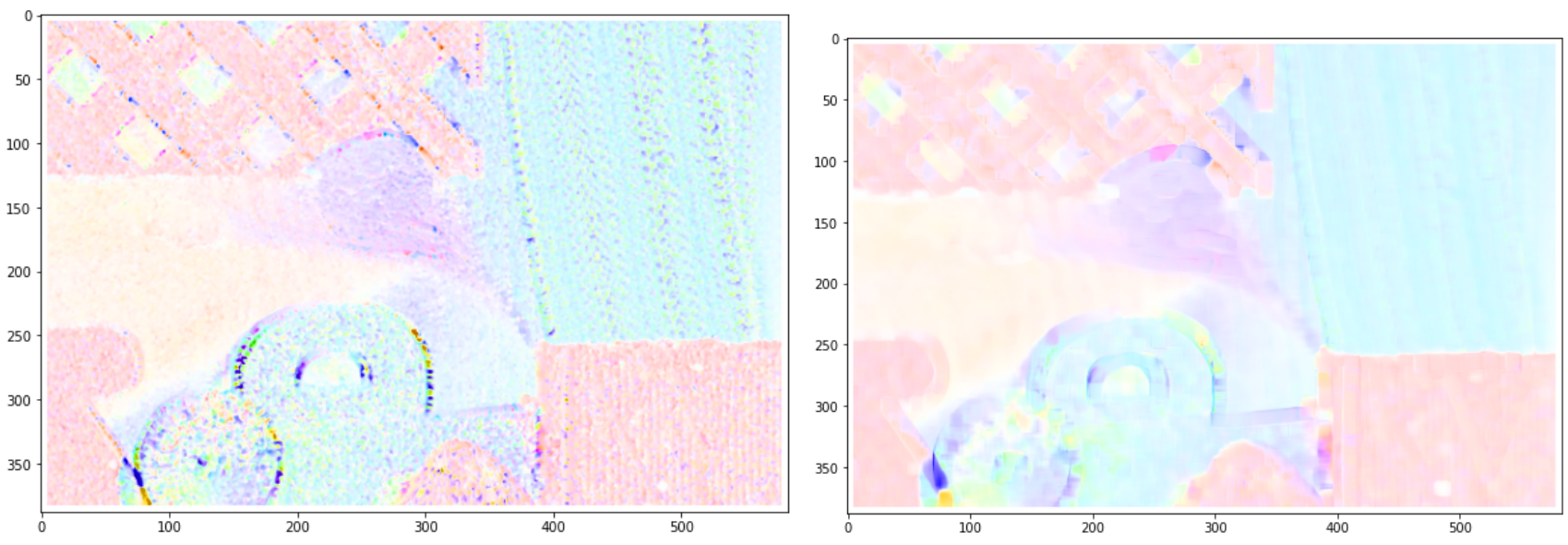
Obs: On the left, we can see that almost all the points on the curtain are blue whereas there are some purple points on the right. This again shows that points whose score may not be reliable can cross a reliable threshold after increasing the window size.

However, we see that on the top left, the diamond pattern  is visible on the right whereas the white stationary areas are covered by pink on the left. This clearly shows that increasing the window size has decreased accuracy of these points.

- Left (gaussian smoothing) VS Right (Average smoothing)



- Left (gaussian smoothing with sigma=1) vs Right (Gaussian smoothing with sigma=5)

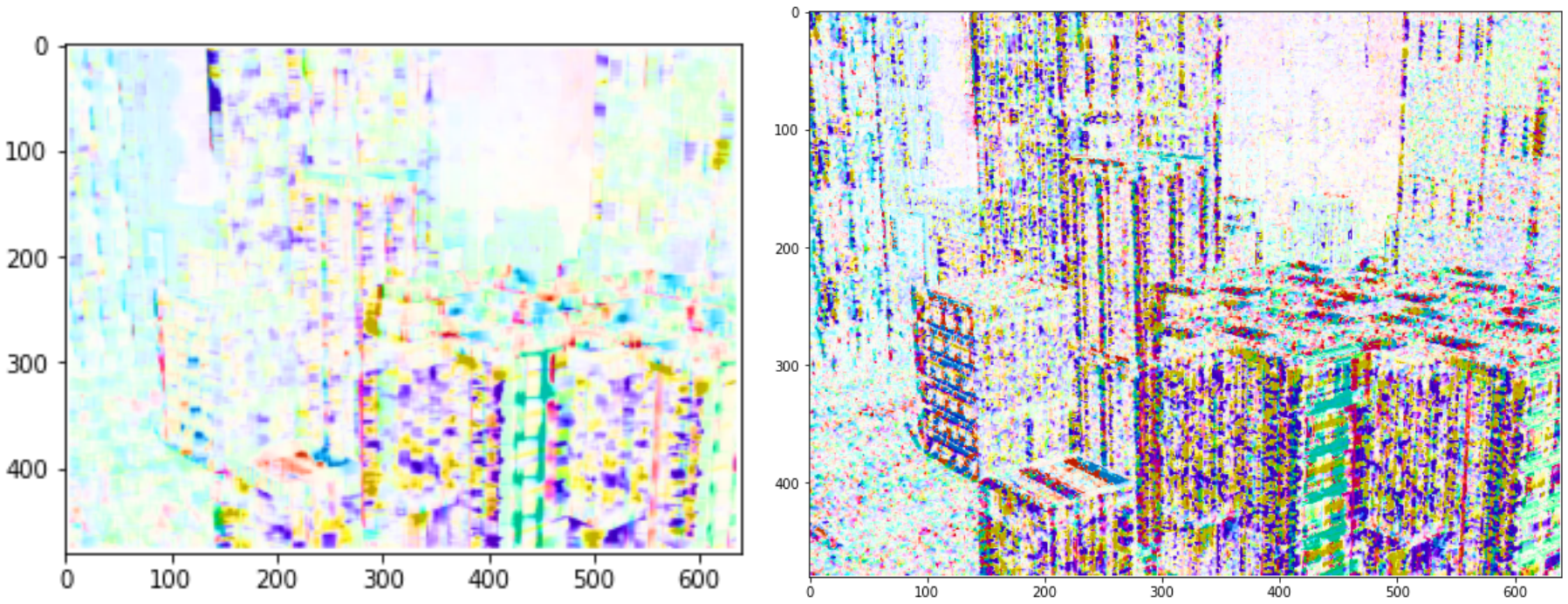


Keeping a larger sigma should reduce the influence of the neighboring points. Right seems to be better (on the rim of the circular object) as increasing SIGMA means that the circular rim (edge region) is NO longer influenced by a different moving body (the bed in this case).

On Urban2

In general, the algorithm DOES NOT WORK WELL for this image. The motion is large and due to neighboring points not being part of the same rigid body, the assumption that “neighbouring pixels have similar optical flow” is NOT very justified in this type of scene.

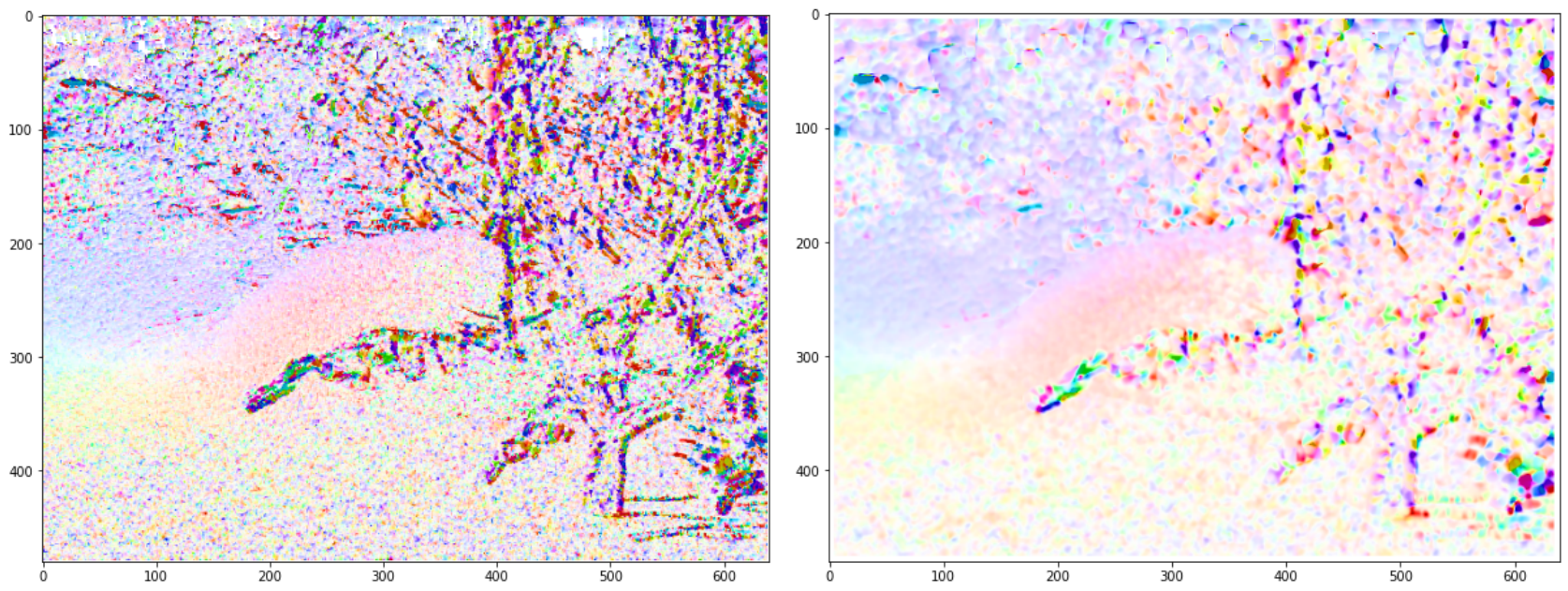
- Left (window size=11) VS Right (Window size=3)



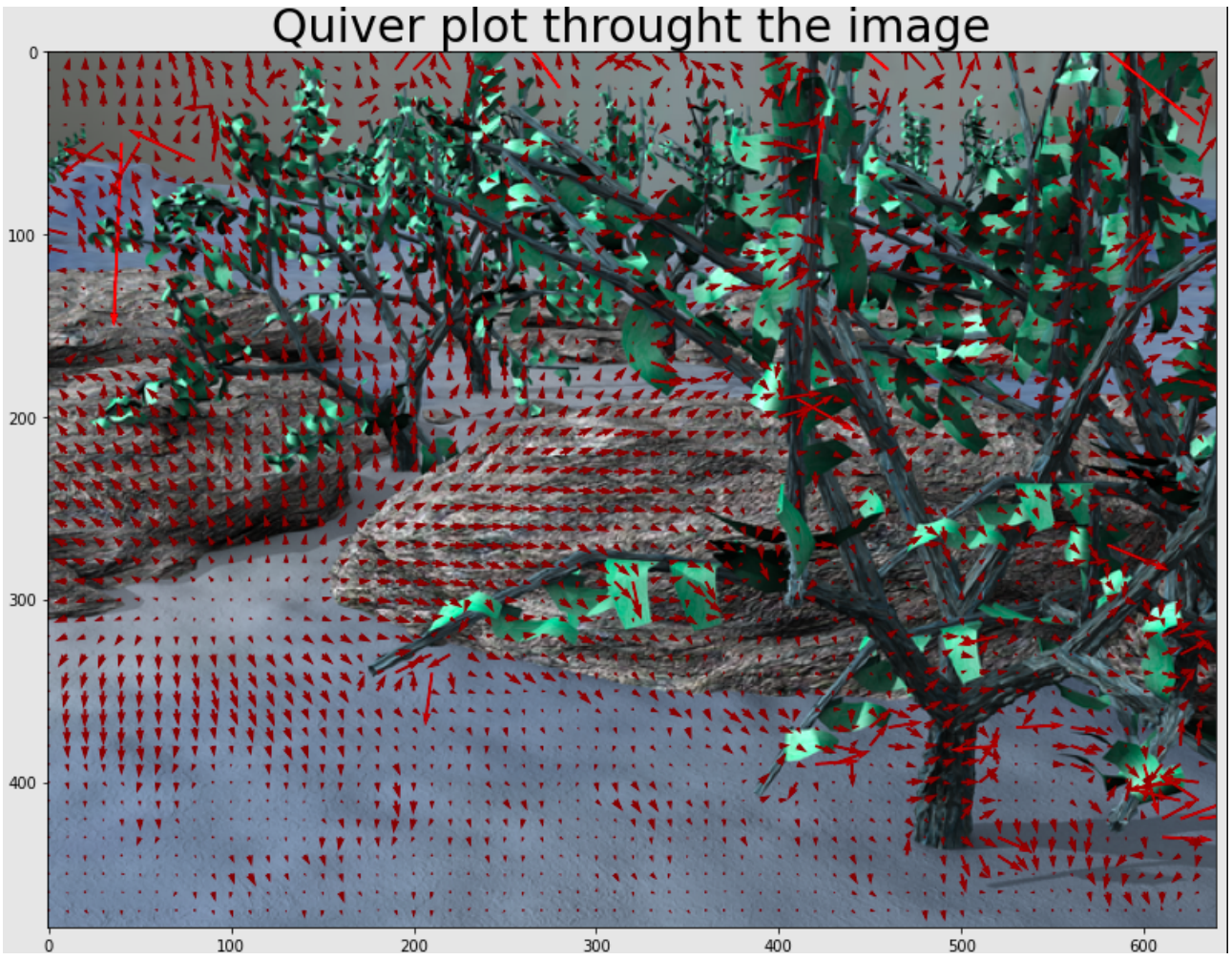
Obs: On left, due to larger window size, we can see the flow vector diluted and merging into the neighboring pixels. On the right, we can clearly see that the corners are accurately portrayed (LIGHT BLUE) whereas the other planar points are NOT (since their score is unreliable and due to small window size, the corner reliable points are not able to contribute and neutralize the scores of the planar points).

Results on Grove 3

- Avg smoothing on WIndow size =3 (left) VS Gaussian smoothing with window size of 11



- Here, the best plot was obtained by using a large window size of 23. However, the result ITSELF is not very good (since due to the presence of so many corners with similar appearance), error in matching one corner to it’s correct correspondence may be misleading the LEAST SQUARES SOLN.



My main experiments with Sobel operators were with the 3x3 one and the naive 3x1 one. In my view, both gave similar performance and I would not have been able to discriminate between them had I not known which is which.

N instance where Pyramidal method performs better

- Firstly, I would like to comment on the paper and the differences in my implementation:
  - The paper downsampled the image using the following formula:

$$
\begin{aligned}
I^L(x,y) \quad = \quad & \frac{1}{4} I^{L-1}(2x, 2y) + \\
& \frac{1}{8}\left(I^{L-1}(2x-1, 2y) + I^{L-1}(2x+1, 2y) + I^{L-1}(2x, 2y-1) + I^{L-1}(2x, 2y+1)\right) + \\
& \frac{1}{16}\left(I^{L-1}(2x-1, 2y-1) + I^{L-1}(2x+1, 2y+1) + I^{L-1}(2x-1, 2y+1) + I^{L-1}(2x+1, 2y+1)\right).
\end{aligned}
$$

    -
    - OpenCV uses a slight variation of this and this is the one I implement as well:
      - Correlate the matrix with the following kernel

$$
\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}
$$

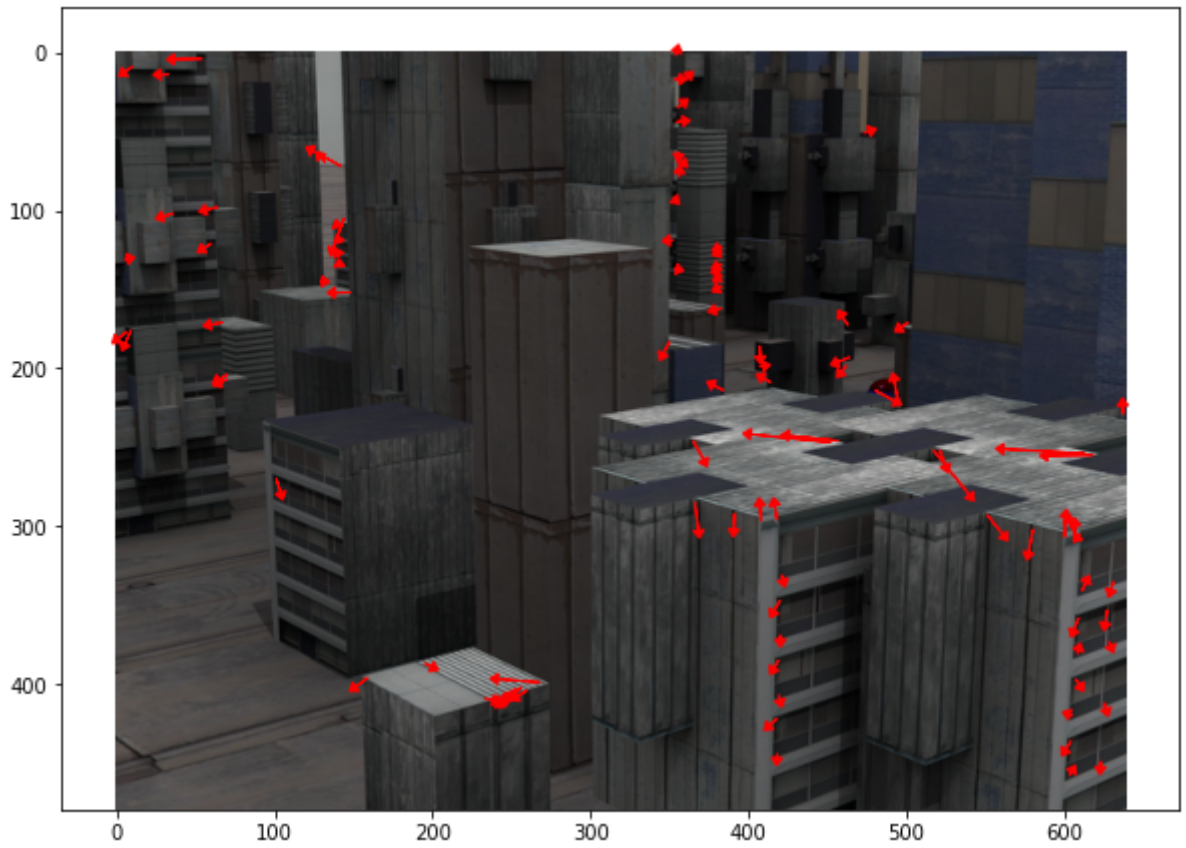
      -
      - Then, ignore every alternate row and column
  - For upsampling, the paper suggests simply multiplying the gradient vector by 2. However, the odd-numbered points like 23 would map to row 11.5 which would need to be rounded to one of the integers. To avoid this, I do the following:
    - Injecting even zero rows and columns
    - Convolve the result with the same kernel (mult by 4)

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

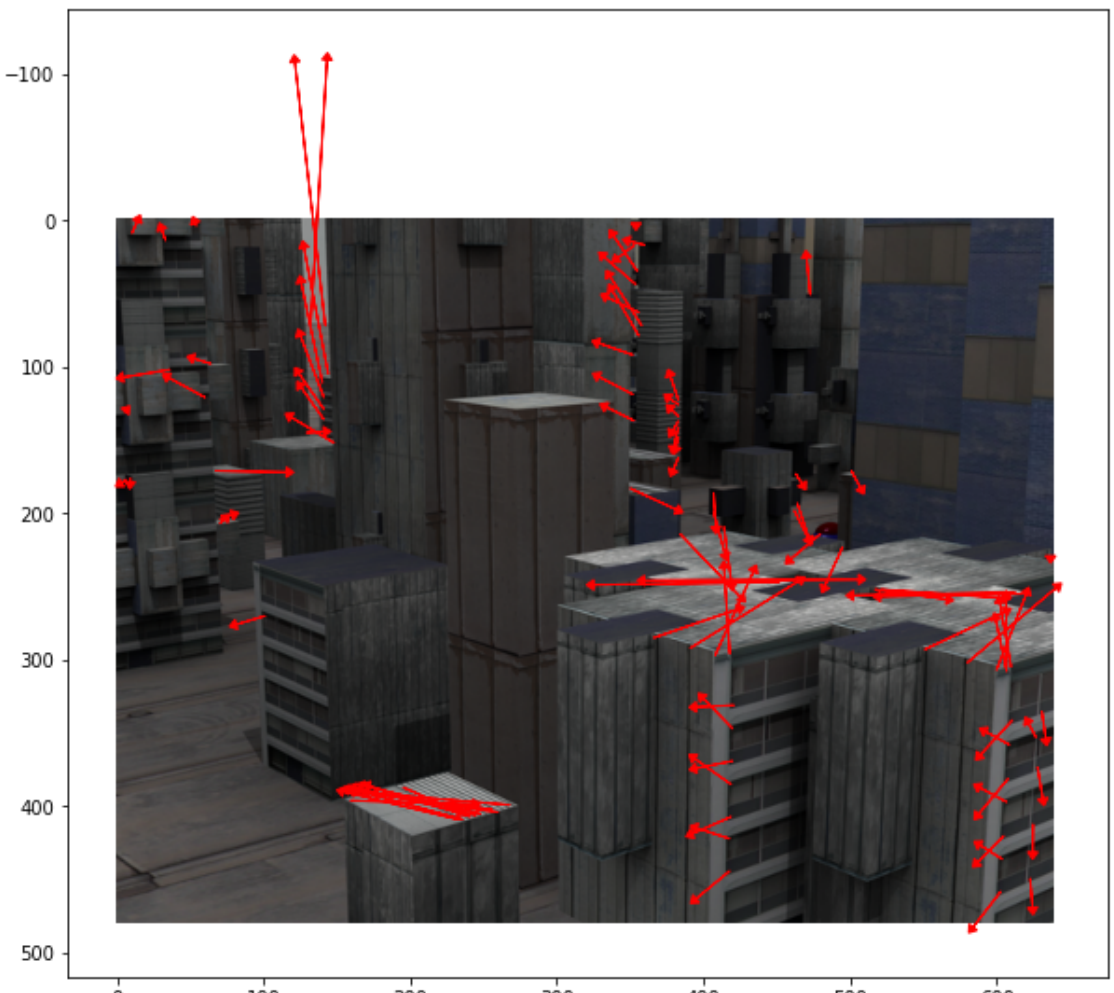
-
    - Although the paper tries to improve the estimate “k” times, the algorithm in the ASSIGNMENT PDF did it just once (k=1) and here, I have obeyed the assignment pdf

Slightly Better results when tested on Urban2 with consecutive frames being frame07.jpg and frame14.jpg

Naive implementation



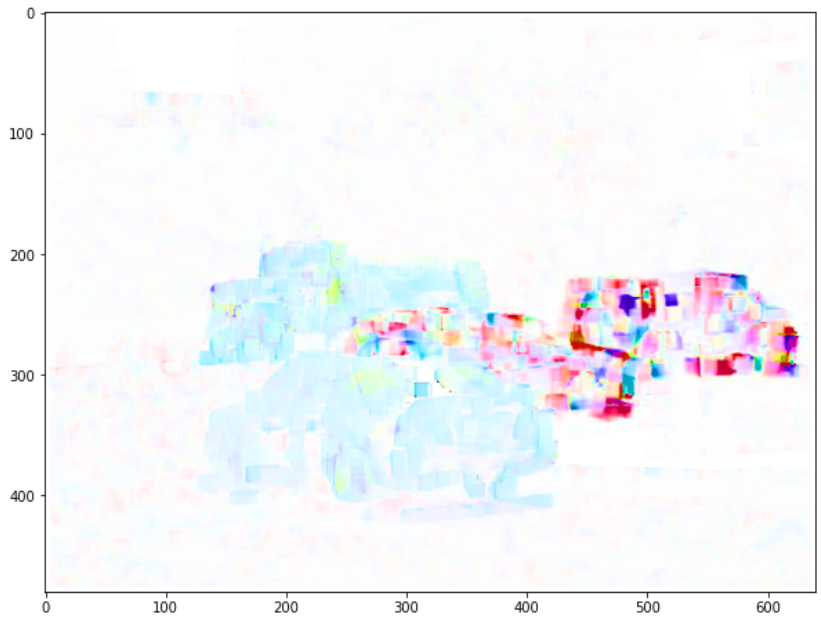
Pyramidal implementation (window size=9, levels=4)

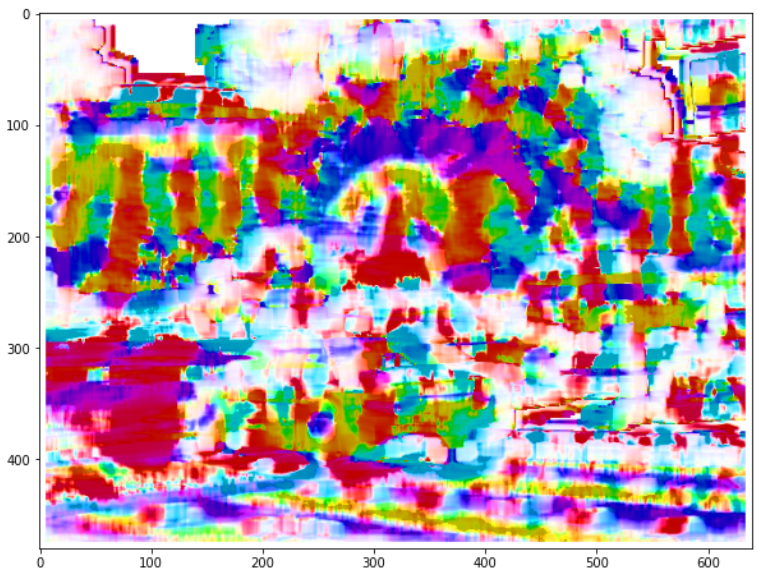


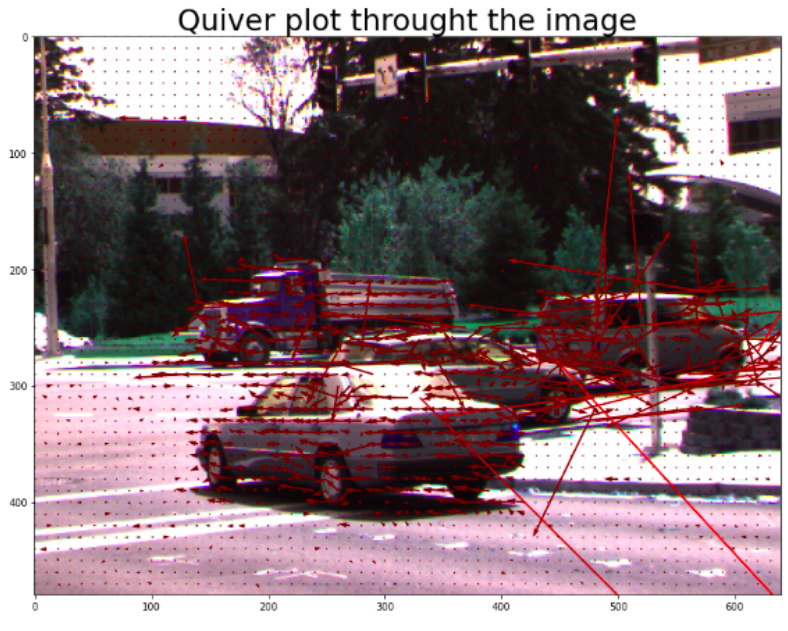
Even though the pyramidal version overestimates some of the vector's magnitude wise, it does a better job of guessing direction and magnitude in general.
The scene objects do not just shift to the bottom right but there is a rotation component (possibly of the camera) as well. This means that most of the vectors actually should point to the top left (slightly) and we see this is the case in image 2, whereas image 1 misses it

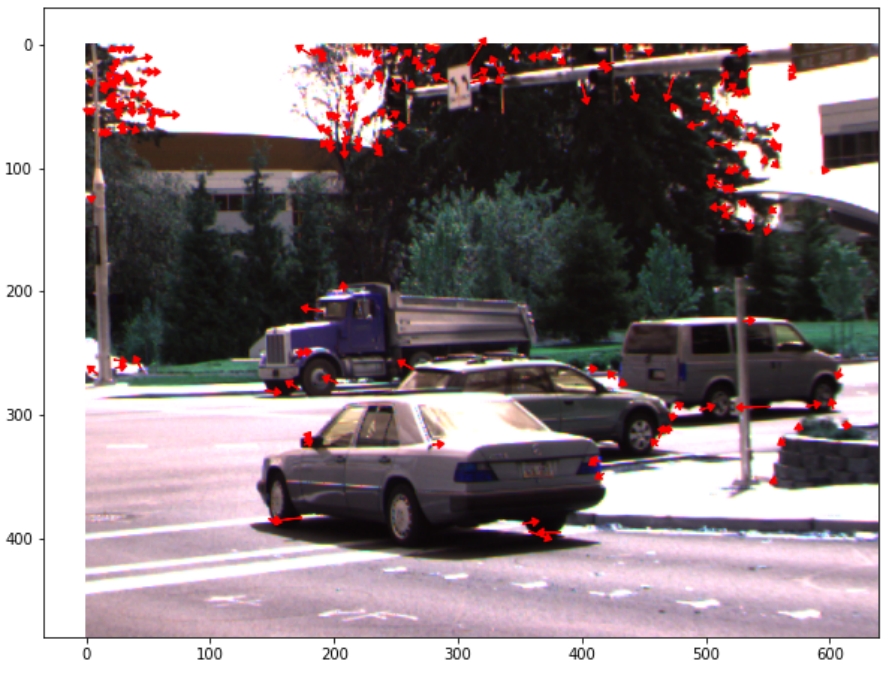
I wanted to get access to a scene where the pixels move quickly, so as to check if the pyramidal version works better than the naive version. Hence, I am using the “dumptruck” scene from the dataset.
Surprisingly Slightly Worse results when tested on DumpTruck with consecutive frames being frame07.jpg and frame14.jpg

- Flow obtained by Naive (LEFT) VS Flow obtained by pyramidal method (RIGHT)
    - Pyramidal seems worse, both in terms of accuracy and in terms of robustness.






Quiver plot throught the image
0
100
200
300
400
0
100
200
300
400
500
600


0
100
200
300
400
0
100
200
300
400
500
600