Dixon Jin and Emerson Boyd
EECE 5639
12/3/17

Computer Vision Project 3 - Dense Optical Flow

Abstract

The purpose of this project was to estimate the dense optic flow from a pair of images using the Lucas-Kanade method. The first step was to read in two images and convert them into grayscale. Then, the spatial intensity gradients were calculated for the second image. In order to do this, the image was first smoothed to get rid of some noise, and then a horizontal and vertical Prewitt filter was applied. After, the temporal gradient was calculated by subtracting the smoothed version of the first image from a smoothed version of the second image. Finally, the flow vector was calculated by creating a system of linear equations at each pixel using the spatial intensity gradients and temporal gradient. The flow vectors were then overlaid onto the image to show the flow field.

Description of Algorithms

The first algorithm used was the spacial 2D Gaussian filter which was applied onto the images to smooth them, but as a consequence, blur them as well. The 2D Gaussian filter took a normalized average of the box of pixels around the center based on the standard deviation provided. Changing the standard deviation in this equation would change the values that each pixel assume, and as such, different standard deviations were tested in order to determine the best output.

The next algorithm used was the Prewitt edge detector algorithm. Depending on whether or not the edge detector is vertical or horizontal, the filter would search for different edges, however, the algorithm behind them is the same. For both, the filter was a 3x3 matrix, and it calculated the value of the current pixel by taking the row or column of pixels after the current pixel and subtracted the row or column of pixels before the current pixel. This calculated the spatial intensity gradients and will be used in calculating the flow vector in the next algorithm.

After coming up with horizontal and vertical Prewitt values for each pixel, the C matrix was calculated. The C matrix was a 2x2 matrix consisting of the vertical Prewitt value squared, the horizontal Prewitt value squared, and the vertical and horizontal Prewitt values multiplied together. The C matrix is one side of a system of equations with the other side consisting of a 2x1 matrix consisting of the temporal gradient multiplied by the vertical Prewitt value or the horizontal Prewitt value. We called this matrix the T matrix. The flow vector at each pixel was then calculated by creating a system of equations with the T matrix and the C matrix. The

system of equations took the form Cx = -T, where x is the flow vector consisting of components u and v. We used MATLAB's "\" system of equations solver to calculate the flow vector at each pixel by performing the command "x = C \ (-1 * T)".

The full path of algorithms visualized can be seen in Appendix A1. The flow vectors can be seen in orange in the image.

Experiments, Values of Parameters Used, and Observations

These experiments were run for the first and second images in every image set. There were five image sets given in total.

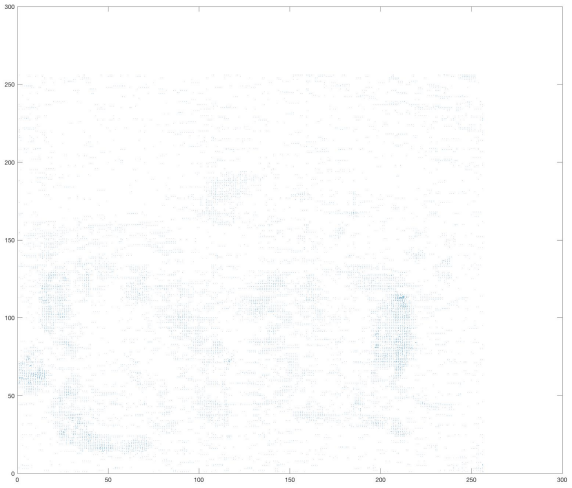
We varied certain parameters throughout our testing to determine the optimal values for certain algorithms. The two parameters that were varied were the standard deviation of our 2D Gaussian filter and the size of the Lucas-Kanade window.

We chose an initial C and T matrix window size of 3x3 because the Lucas-Kanade tutorial we followed suggested that we try that size [1]. We then varied the Gaussian smoothing filter standard deviation for that window size and chose the best result. After finding the optimal standard deviation, we varied the window size using the optimal standard deviation to find the optimal window size.

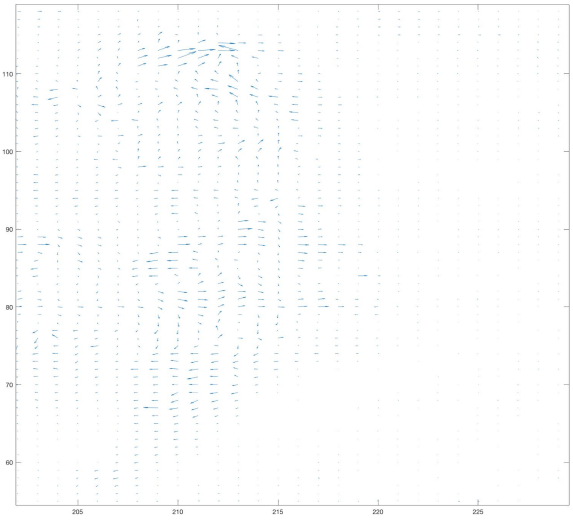
We used the first two images from the image set "toy1" in our experimentation with the two parameters. The closeup images of the flow vectors are of the elephant that moves to the right in the top right of the image set. Once the final optimal parameters were determined, we executed our algorithm on the other image sets using those parameters.

While the assignment requested we overlay the quiver plot on top of the images, we found it was easier to distinguish the flow when viewing the vectors separately from the image. For the flowchart in Appendix A1, we do overlay the flow vectors on top of the image. For demonstrating our experiments, however, we do not overlay the flow vectors on top of the images.

Here, a window size of 3x3 and a standard deviation of 0.5 was used.



The overall flow vectors in the image set.

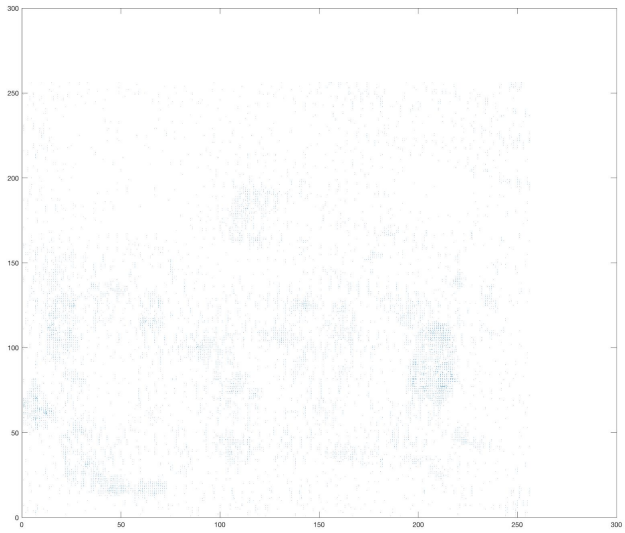


A closeup of the flow vectors for the toy elephant.

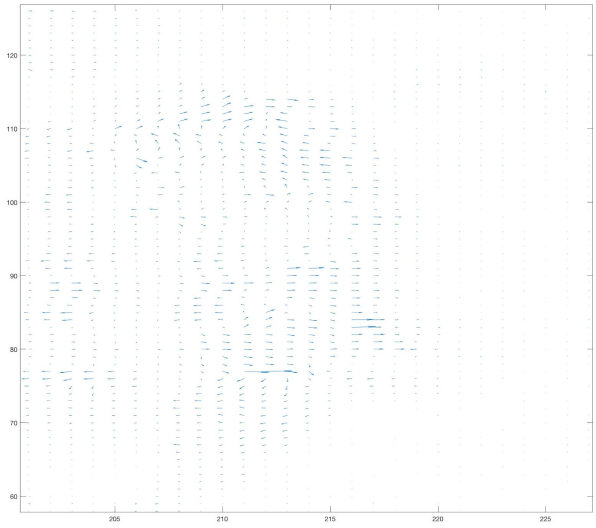
In the zoomed-out image, the concentrated parts of the flow vectors are where there is the most movement between image 1 and image 2 of the first set of toys. The most concentrated patch is where the elephant in the top right of the of the images moves from left to right. The reason that the concentrated patch appears toward the bottom right of the quiver plot is because quiver inverts the y axis when plotting the vectors.

It is very difficult to distinguish the direction of the vectors with the photo of the entire flow vector image. The zoomed-in image gives a better idea of how the vectors have been calculated.

Here, a window size of 3x3 and a standard deviation of 1.0 was used.



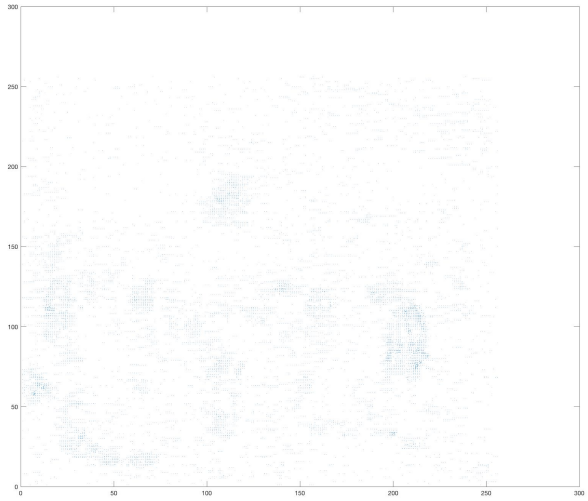
The overall flow vectors in the image set.



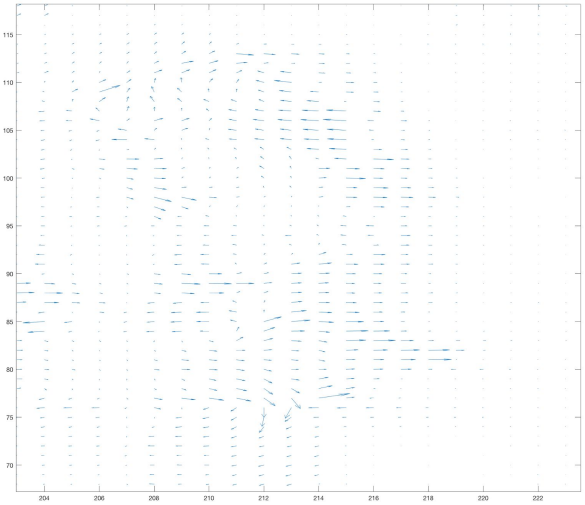
A closeup of the flow vectors for the toy elephant.

Here, we can analyze the differences between the flow vectors when changing the standard deviation of the Gaussian smoothing filter. Between a standard deviation of 0.5 and 1.0, we see that using a deviation of 1.0 leads to more pronounced flow vectors and a greater number of them. It is for these reasons that we prefer a standard deviation of 1.0.

Here, a window size of 3x3 and a standard deviation of 1.4 was used.



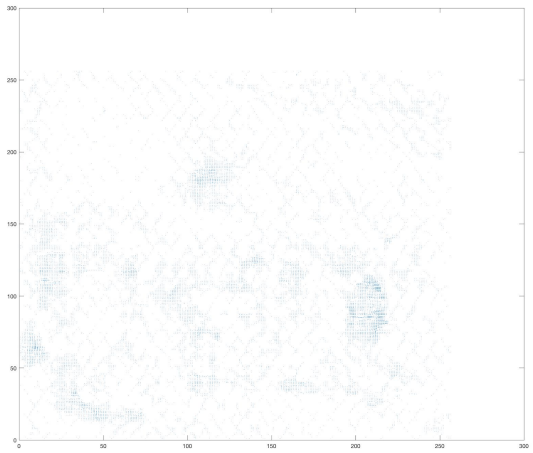
The overall flow vectors in the image set.



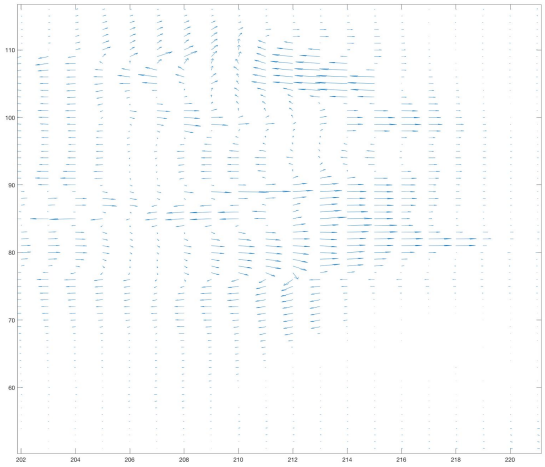
A closeup of the flow vectors for the toy elephant.

Here, we see a slight improvement in the closeup using a standard deviation of 1.4 over 1.0. There are a slightly greater number of flow vectors that point in the same direction (to the right) when using a standard deviation of 1.4.

Here, a window size of 5x5 and a standard deviation of 1.4 was used.



The overall flow vectors in the image set.

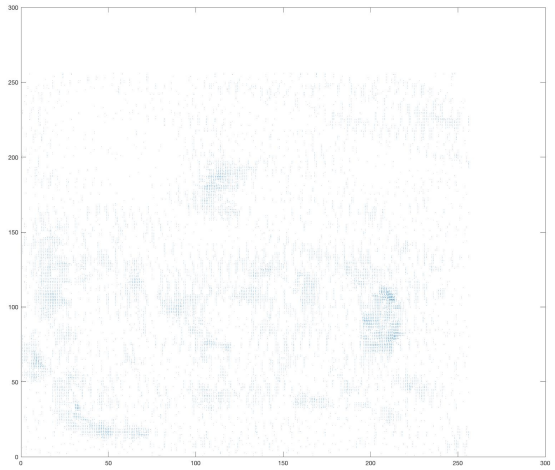


A closeup of the flow vectors for the toy elephant.

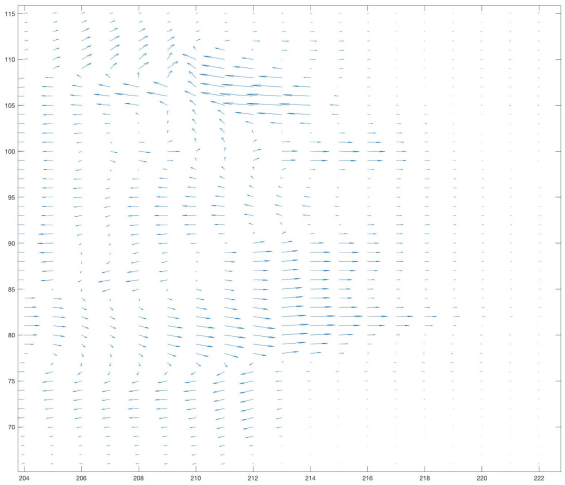
The patches of flow vectors when using a window size of 5x5 is essentially the same as the patches of flow vectors when using a window size of 3x3.

The flow vectors in the closeup of this trial are far more pronounced. However, there are many more vectors that point in the wrong direction (from right to left), so we stick to the notion that the 3x3 window size is best when calculating the C and T matrices.

Here, a window size of 7x7 and a standard deviation of 1.4 was used.



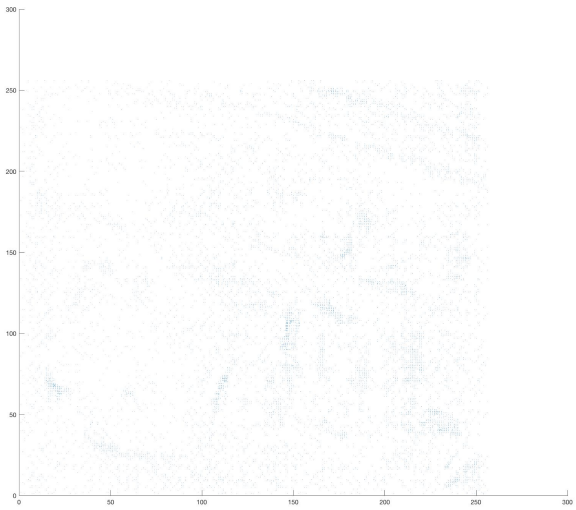
The overall flow vectors in the image set.



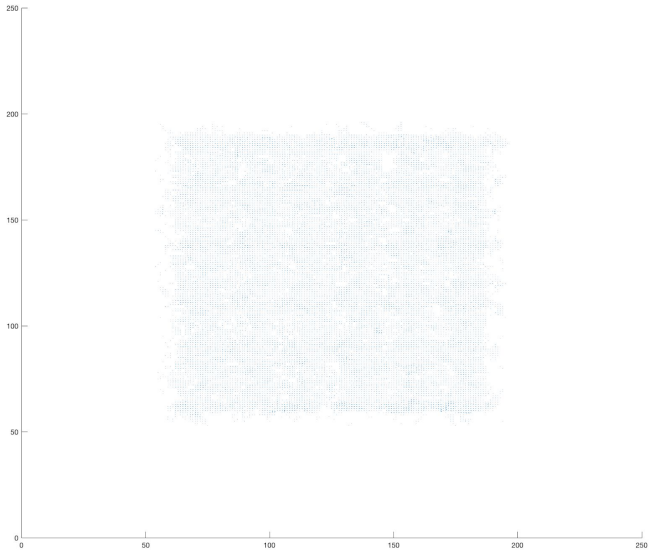
A closeup of the flow vectors for the toy elephant.

Here, a window size of 7x7 brings the flaws of using a window size of 5x5 to an even worse degree. Many of the flow vectors point in the wrong direction (some diagonal, some backwards). We ultimately determine that the ideal window size for calculating the C and T matrices is 3x3. The ideal standard deviation when smoothing the images is 1.4.

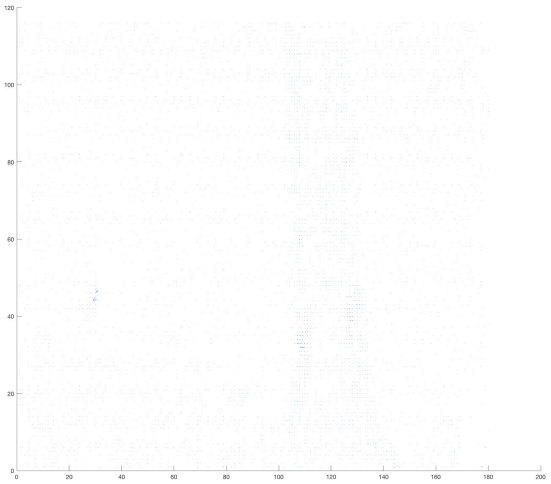
Below are more flow vector calculations for other image sets using a standard deviation of 1.4 and a window size of 3x3.



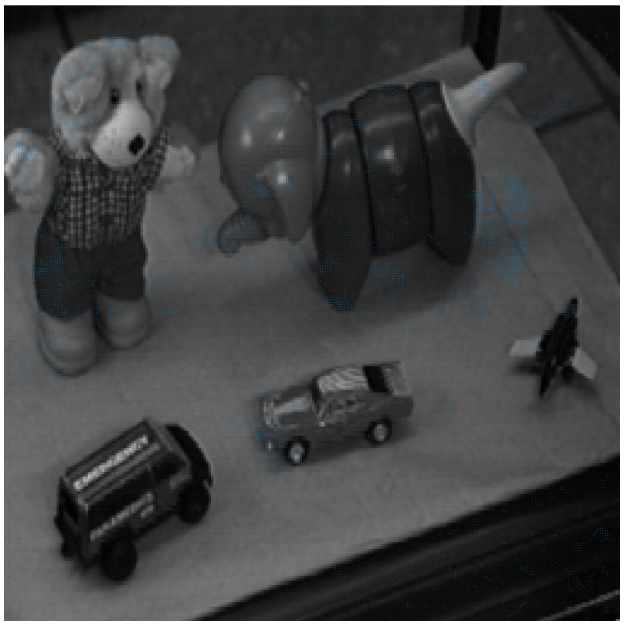
The flow vector between image 2 and image 3 of image set toy1



The flow vector between image 1 and image 2 of image set LkTest1

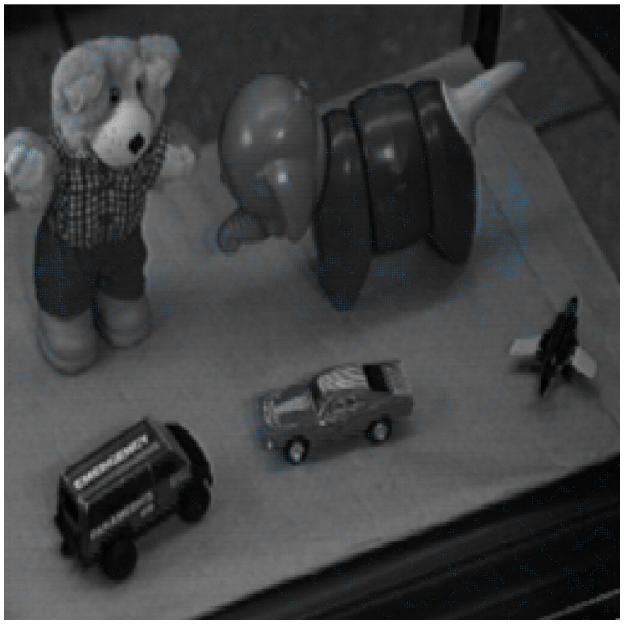


The flow vector between image 1 and image 2 of image set LkTest2



The flow vector between image 1 and image 2 overlaid on image 1 of image set toy1

For the flow vectors overlaid on image 1 of the image set “toy1”, we see the densest set of flow vectors are located on the left/right of the objects. This is because the objects move left to right between the images. There is also a small amount of flow detection in the background, likely due to slight movement in the camera or noise that failed to be eliminated from the Gaussian smoothing filter. Ultimately, the flow vector detection was successful for this image.



The flow vector between image 2 and image 3 overlaid on image 2 of image set toy1

Conclusion

One benefit of optical flow calculation is that it is rather quick relative to other algorithms we have implemented. For example, calculating homographies required several steps and took about a minute to accurately predict. This algorithm only has three simple steps and computes virtually instantaneously. This is one reason why the Lucas-Kanade method is so useful.

Optical flow can make for powerful algorithms if the findings are applied in greater depth. Using optical flow, along with other algorithms like object detection, one can make for several features in higher-level algorithms. These high-level applications could take many forms, such as object movement prediction and human expression tracking. As mentioned earlier, the Lucas-Kanade optical flow calculations are extremely quick and can be performed in real time. One potential future experiment that use the findings of this experiment could be to make a moving dart board. A player could throw a dart, and cameras could be used to track the dart's movement through the air. With fast computation and precise machinery, the trajectory of the dart could be predicted and the dart board could move to land a bullseye every throw. This type of application is possible, as demonstrated in a video referenced in the bibliography [2].

Overall, the use of motion detection in computer vision is very powerful. It is interesting and fun to experiment with algorithms that are used by computer vision scientists today.

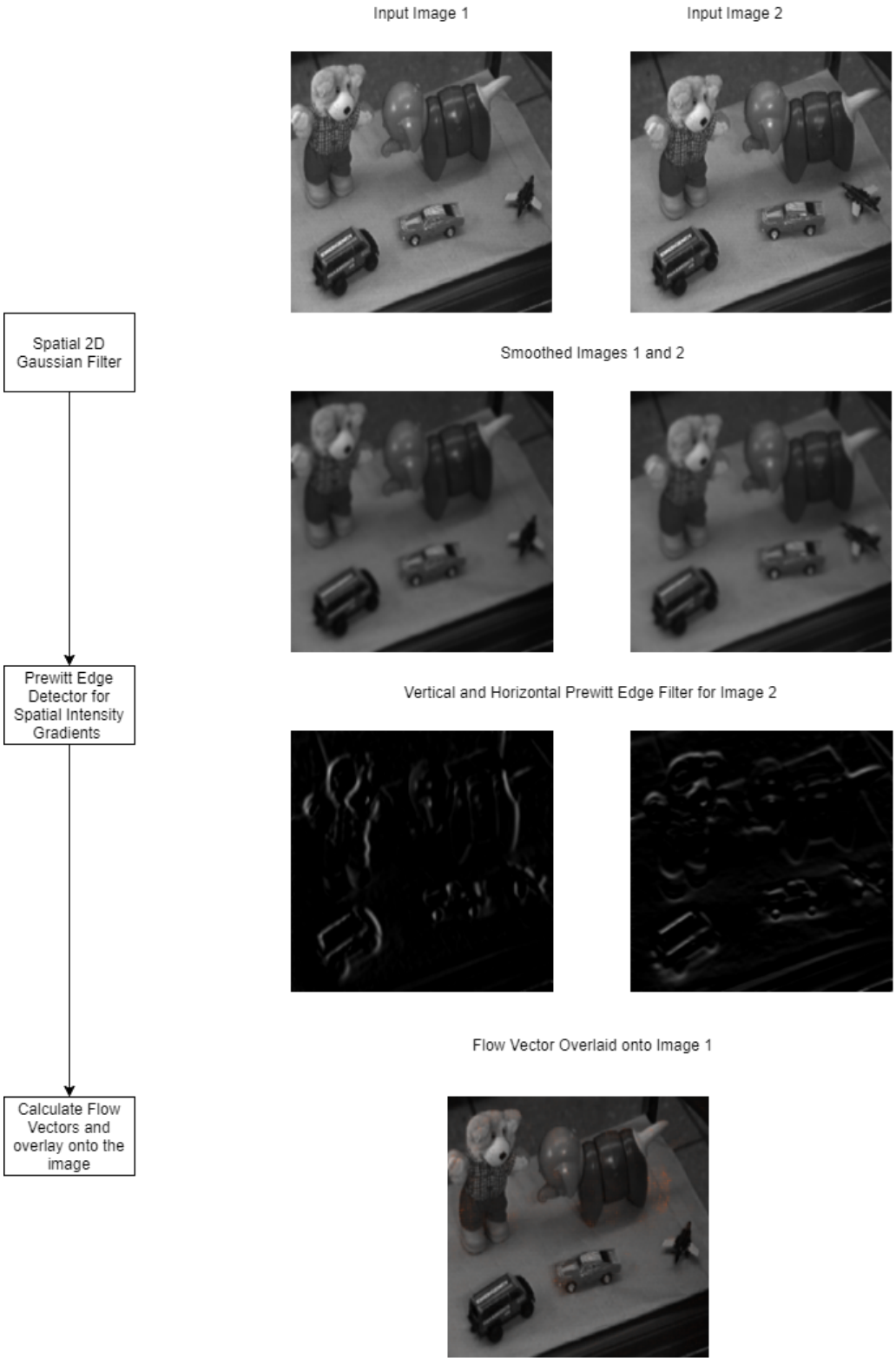
Bibliography

[1]
http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html

[2]
https://www.youtube.com/watch?v=MHTizZ_XcUM

Appendix

A1. Flow Chart



```
clc
clear all
close all

warning off; % warnings were spawning from the linsolve(A,B) calls

tic

imageType = 4;
gaussianSmoothingSigma = 1.4;
CMatrixWindowSize = 3;

% calculates the flow vector between imageNum and imageNum+1
% so, imageNum of 1 calculates the flow vector between images 1 and 2
imageNum = 1;

if (imageType == 1)
    folderName = 'Images/toy1/';
    fileNamePrefix = 'toys';
    fileNameSuffix = '.gif';
    fileNumberMin = 1;
    fileNumberMax = 3;
elseif (imageType == 2)
    folderName = 'Images/toy2/';
    fileNamePrefix = 'toys2';
    fileNameSuffix = '.gif';
    fileNumberMin = 1;
    fileNumberMax = 3;
elseif (imageType == 3)
    folderName = 'Images/LKTest1/';
    fileNamePrefix = 'LKTest1im';
    fileNameSuffix = '.pgm';
    fileNumberMin = 1;
    fileNumberMax = 2;
elseif (imageType == 4)
    folderName = 'Images/LKTest2/';
    fileNamePrefix = 'LKTest2im';
    fileNameSuffix = '.pgm';
    fileNumberMin = 1;
    fileNumberMax = 2;
elseif (imageType == 5)
    folderName = 'Images/LKTest3/';
    fileNamePrefix = 'LKTest3im';
    fileNameSuffix = '.pgm';
    fileNumberMin = 1;
    fileNumberMax = 2;
end

grayImgList = [];

numImages = 0;
for i = fileNumberMin:fileNumberMax
```

```
    fullFileName = strcat(folderName, fileNamePrefix, sprintf('%d',
 i), fileNameSuffix);
    img = imread(fullFileName);

    numImages = numImages + 1;
    grayImgList(:, :, numImages) = img;
end

gaussianFilterList = spacial2DGaussianFilter(grayImgList,
 gaussianSmoothingSigma);

[Iy, Ix] = prewittFilter(gaussianFilterList(:,:,imageNum+1));

It = gaussianFilterList(:,:,imageNum+1) -
 gaussianFilterList(:,:,imageNum);

[CMatrix, TMatrix] = CMatrix(Iy, Ix, It, CMatrixWindowSize);

[u, v] = flowVector(CMatrix, TMatrix);

% uncomment this line to overlay the flow vector on top of the first
 image
% imshow(grayImgList(:,:,imageNum)/255)

hold on
quiver(u,v)
hold off
```

Published with MATLAB® R2017a

```
function [gaussianFilterList] = spacial2DGaussianFilter(imgList,
 ssigma)
%UNTITLED5 Summary of this function goes here
%   Detailed explanation goes here
    Gx = @(x) 1 / sqrt(2 * pi * (ssigma^2)) * exp(-1 * x^2 / (2 *
 (ssigma^2)));
    Gy = @(y) 1 / sqrt(2 * pi * (ssigma^2)) * exp(-1 * y^2 / (2 *
 (ssigma^2)));
    imgListSize = size(imgList);
    xDir = imgListSize(2);
    yDir = imgListSize(1);
    zDir = imgListSize(3);

    boxSize = ceil(((ssigma * 5) + 1)/2)*2 - 1; % gets an odd box size

    startPixel = boxSize - floor(boxSize/2);
    endPixelX = xDir - floor(boxSize/2);
    endPixelY = yDir - floor(boxSize/2);
    gaussianFilterList = imgList;

    gaussianFilterHorizontal = zeros(1, boxSize);
    for i = 1:boxSize
        x = i - ceil(boxSize/2);
        gaussianFilterHorizontal(i) = Gx(x);
    end
    gaussianFilterVertical = transpose(gaussianFilterHorizontal);
    gaussianFilterSumX = sum(gaussianFilterHorizontal);
    gaussianFilterSumY = sum(gaussianFilterVertical);

    gaussianFilterSum = gaussianFilterSumX * gaussianFilterSumY;

    for k = 1:zDir
        gaussianFilterList(startPixel:endPixelY,
 startPixel:endPixelX, k) = (1/gaussianFilterSum) *
 (conv2(conv2(imgList(:,:,k), gaussianFilterHorizontal, 'valid'),
 gaussianFilterVertical, 'valid'));
    end
end
```

Published with MATLAB® R2017a

```
function [verticalPrewitt, horizontalPrewitt] = prewittFilter(image)
%UNTITLED4 Summary of this function goes here
%   Detailed explanation goes here
verticalPrewittFilter = [-1 0 1;
                         -1 0 1;
                         -1 0 1];

horizontalPrewittFilter = [-1 -1 -1;
                           0 0 0;
                           1 1 1];

verticalPrewitt = imfilter(image, verticalPrewittFilter, 'replicate');
horizontalPrewitt = imfilter(image,
 horizontalPrewittFilter, 'replicate');

end
```

Published with MATLAB® R2017a

```
function [C, T] = CMatrix(verticalPrewitt, horizontalPrewitt, It,
 boxFilterSize)

verticalHorizontalPrewitt = verticalPrewitt .* horizontalPrewitt;
verticalPrewittSquared = verticalPrewitt .* verticalPrewitt;
horizontalPrewittSquared = horizontalPrewitt .* horizontalPrewitt;

verticalTemporal = verticalPrewitt .* It;
horizontalTemporal = horizontalPrewitt .* It;

boxFilter = ones(boxFilterSize, boxFilterSize)./(boxFilterSize *
 boxFilterSize);

boxFilterVerticalHorizontalPrewitt =
 imfilter(verticalHorizontalPrewitt, boxFilter);
boxFilterVerticalPrewitt = imfilter(verticalPrewittSquared,
 boxFilter);
boxFilterHorizontalPrewitt = imfilter(horizontalPrewittSquared,
 boxFilter);
boxFilterVerticalTemporal = imfilter(verticalTemporal, boxFilter);
boxFilterHorizontalTemporal = imfilter(horizontalTemporal, boxFilter);

sizeMatrix = size(verticalHorizontalPrewitt);
sizeX = sizeMatrix(1);
sizeY = sizeMatrix(2);

C = zeros(sizeX, sizeY, 2, 2);
T = zeros(sizeX, sizeY, 2, 1);
for i = 1:sizeX
    for j = 1:sizeY
        C(i,j,:,:) = [boxFilterHorizontalPrewitt(i, j)
 boxFilterVerticalHorizontalPrewitt(i, j);
            boxFilterVerticalHorizontalPrewitt(i, j)
 boxFilterVerticalPrewitt(i, j)];
       T(i,j,:,:) = [boxFilterHorizontalTemporal(i, j);
 boxFilterVerticalTemporal(i, j)];
    end
end

end
```

Published with MATLAB® R2017a

```
function [u, v] = flowVector(CMatrix, TMatrix)

% invert the T matrix for the calculation of u and v
TMatrix = -1 * TMatrix;

sizeMatrix = size(CMatrix);
sizeX = sizeMatrix(1);
sizeY = sizeMatrix(2);
u = zeros(sizeX, sizeY);
v = zeros(sizeX, sizeY);

for i = 1:sizeX
    for j = 1:sizeY
        %x = CMatrix(i,j,:,:)\TMatrix(i,j,:);
        C(:,:) = CMatrix(i,j,:,:);
        T(:,:) = TMatrix(i,j,:);
        %T = transpose(T);
        x = C\T;
%         x2 = inv(C'*C)*C'*T;
        u(i,j) = x(1);
        v(i,j) = x(2);
    end
end

end
```

Published with MATLAB® R2017a