Project #1

Motion Detection using 1-D Temporal Derivatives

EECE 5639

Instructor: Octavia Camps

Authors: Juan Pablo Bernal Medina, Shivam Sharma

Abstract

In this project we used temporal derivatives to highlight moving objects in a sequence of images. We varied the inputs by first blurring the set of images by applying either box or gaussian filters. We also used different derivative filters: a standard 0.5[-1 0 1] Sobel operator and 1-D derivative of a gaussian filter with various values of standard deviation. We compared the results of using various blurring filters, derivative filters, and threshold values.

Algorithm

The algorithm begins with reading in a set of images and converting them to grayscale. The image is then blurred if so desired with a box filter of a given size or a Gaussian filter of a given standard deviation. Then the derivative filter is defined. We either used a Sobel filter: 0.5[-1 0 1] or a 1D derivative of a Gaussian filter. This was calculated using this function:

$$\frac{\mathrm{d}G(x)}{\mathrm{d}x} = -Kxe^{\frac{-x^2}{2\sigma^2}}$$

K was computed to make the sum of mod(G`(x)) to be 1, thereby normalizing it. Like for every other gaussian filter the size is taken as 5 times the standard deviation, and made sure to be an odd number.

With the derivative filter, we then computed the derivative at each frame of the image set, and saved it as a new image. Then the derivative image is converted to a binary image by determining the suitable threshold value. This is done by measuring the average noise of pixels in the background and assigning a threshold slightly greater than the noise. The EST_NOISE function from the previous was used to look at the average noise of each pixel. The thresholding allowed the derivative to ignore the noise in the background but hopefully still capture any movements in the image.

The binary derivative image is then masked over a pure red image, which is then subsequently added back to the original RGB image with slightly lower intensity. This produced images where the high derivative pixels are red and the rest is slightly darker to make the red more visible. This allowed us to much more easily visualize our results, compared to just looking at the binary derivative mask.

The set of final images are then encoded into a .mp4v video file and saved. This process is done for each set of images: RedChair, Office, EnterExitCrossingPaths2cor.

Experiments

Modifying the temporal filter (edge/motion detection):

For this experiment, a temporal filter 0.5[-1 0 1] and a gaussian derivative filter with different sigma were applied. The different filters will determine how many frames before and after are going to be compared to detect motion. With the first filter we are only looking at one frame before and one frame after. First, the temporal filter was applied (Fig.1) and the result was not great. A lot of the motion of the man is not detected like the arm or the right foot. Because, if the motion is slow the motion won't be seen after a couple of frames after or before and this filter only looks at 1 after and 1 before. Then, the gaussian derivative was applied and the results were better. The gaussian derivatives look at 5*sigma frames before and after doing a better job at detecting motion. The first sigma used was 1 (Fig.2) the results are better than the temporal filter but still, but not very different there are still some parts missing. With sigma 2

(Fig.3) the result improves a lot. Now the filter looks at 10 frames after and before. Now the man is almost fully captured. Finally, sigma was increased to 10 and 20 (Fig4 and Fig4.1) making the result as good as it can be using the gaussian derivative. It was noticed that 10 and 20 are identical. On the side, we tried with sigma values of 50 100 and 200 and all of them gave the same output. It is because after a certain number of frames the image is constant because the object is no longer there. Consequently, the values of far away frames won't modify the output.

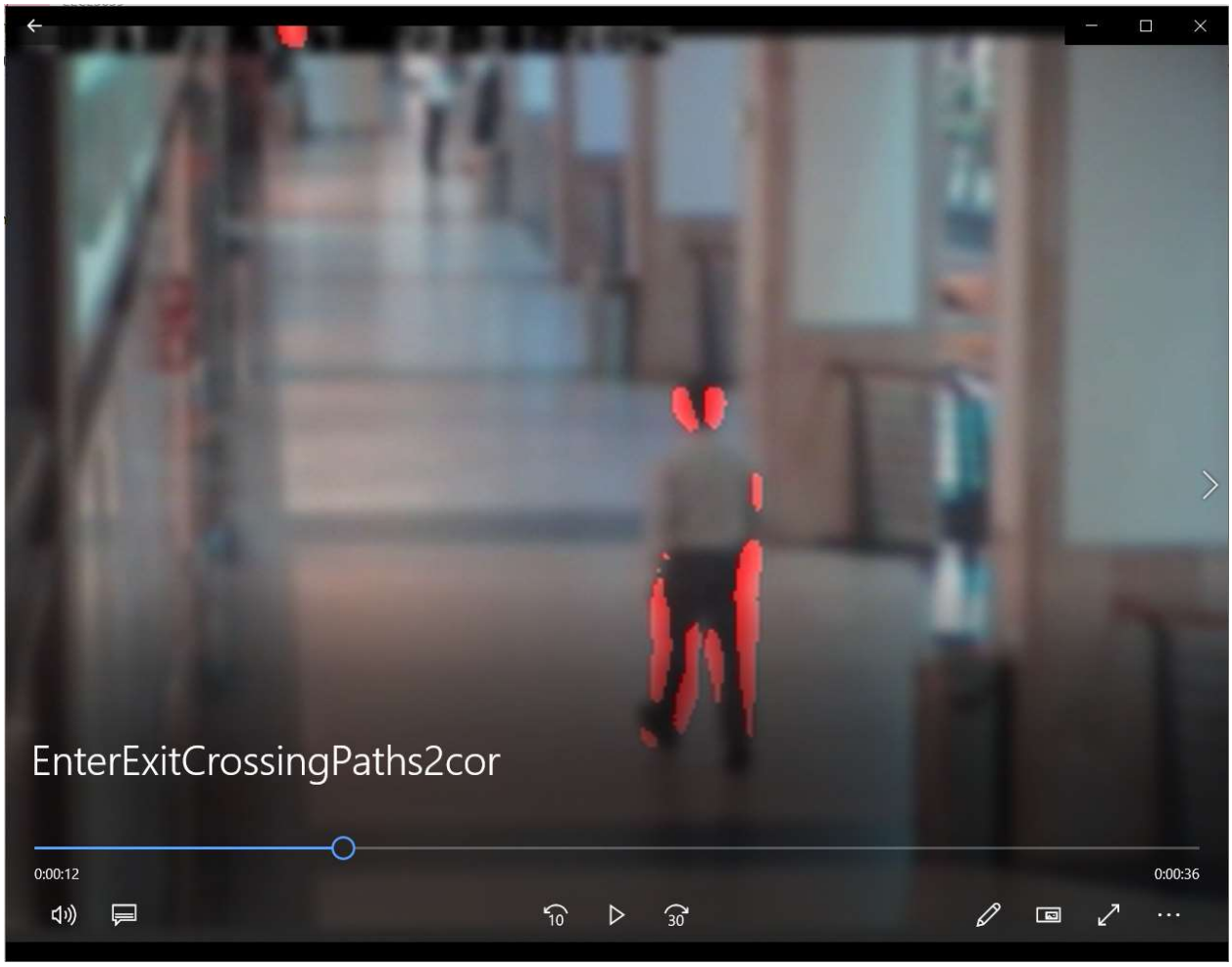

Fig.1 temporal filter 0.5[-1 0 1]

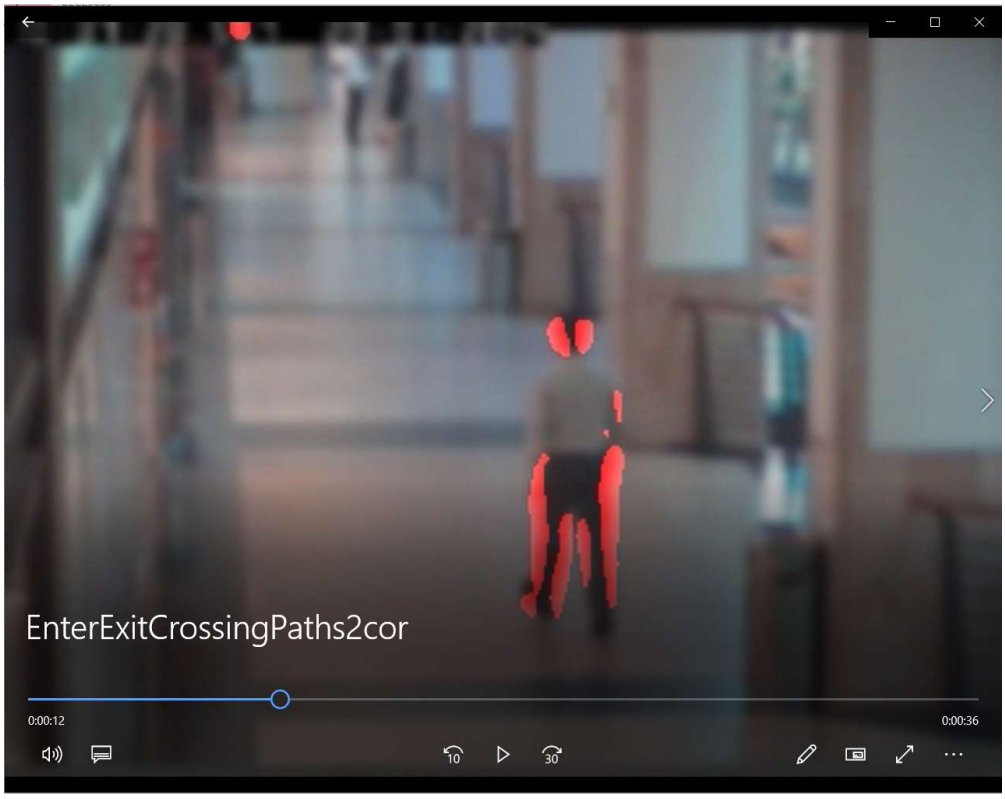

Fig.2 Gaussian derivative filter with sigma 1

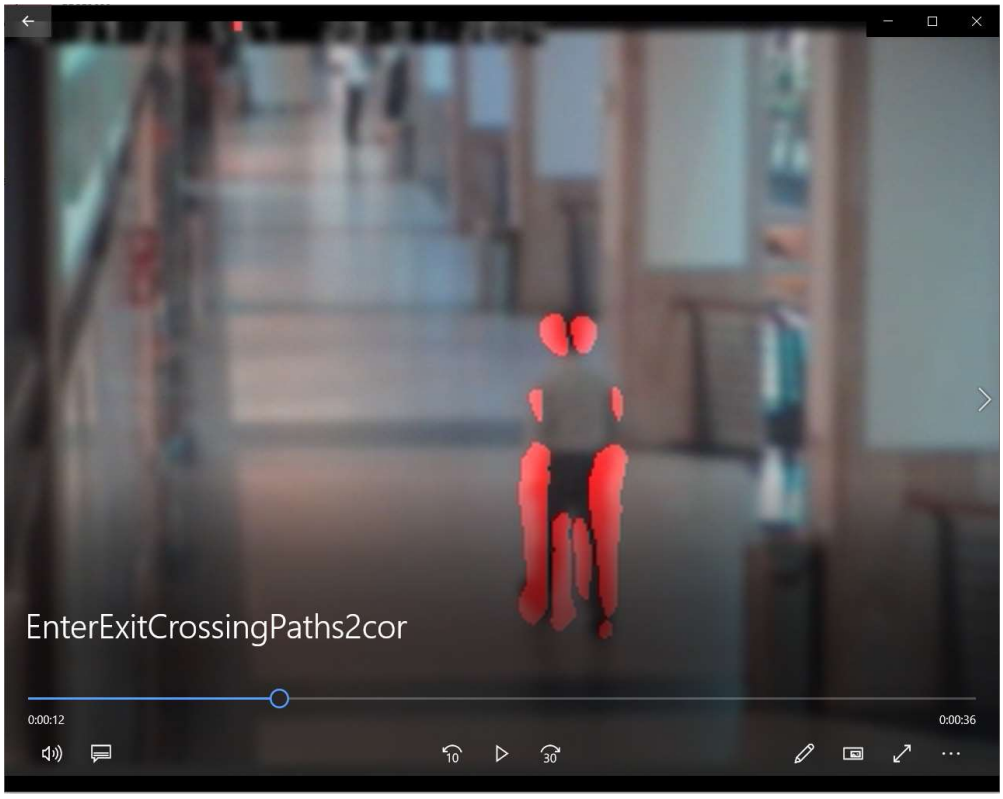

Fig.3 Gaussian derivative filter with sigma 2

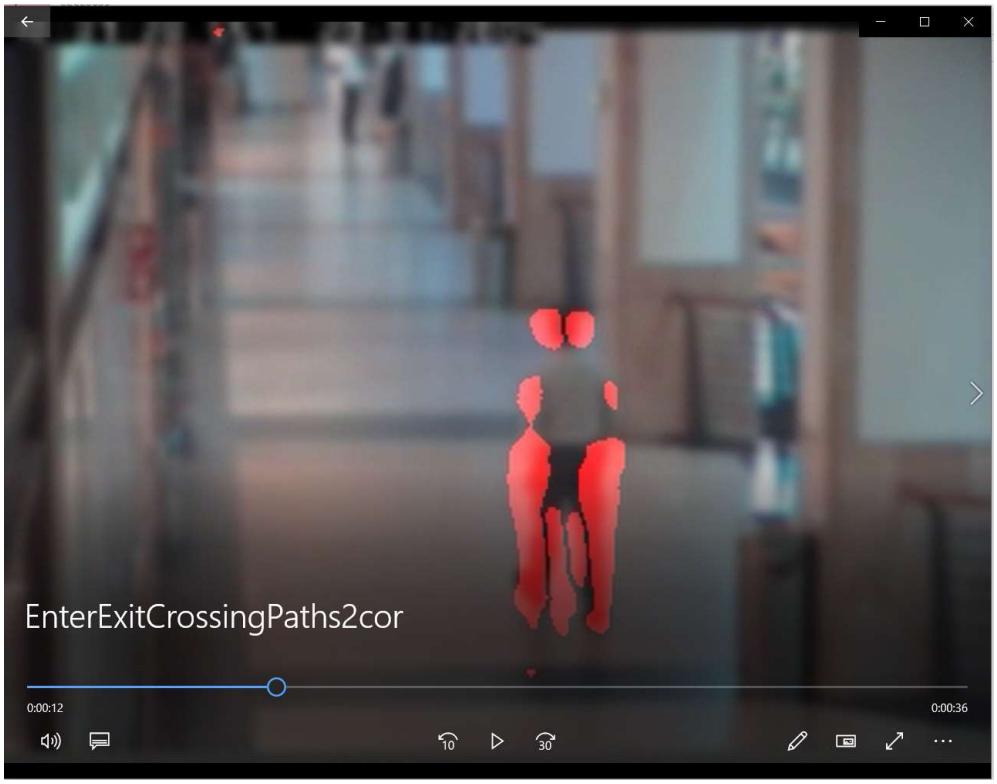

Fig.4 Gaussian derivative filter with sigma 10

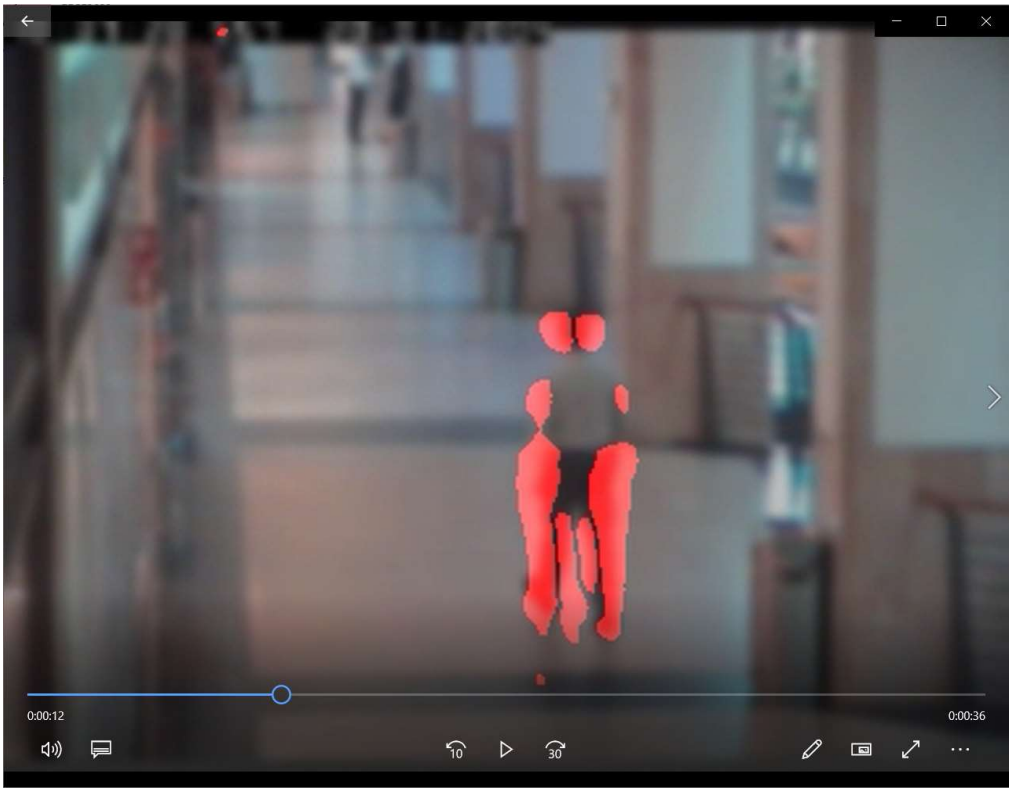

Fig.4.1 Gaussian derivative filter with sigma 20

Modifying the spatial filter (smoothing):

The second experiment was changing the amount smoothing done before applying the temporal derivative filter. First, a 3x3 box filter was applied (Fig.5), then a 5x5 box filter (Fig.6) and finally a 2D gaussian filter with values of sigma 1, 2 and, 4 (Fig.7 to 9). By applying these filters, the noise was reduced and thus, the variations of pixels where there aren't any objects moving are reduced. However, if there is too much blurring, the moving pixels will be merged with many other pixels and the motion won't be detected. For the box filters, it can be seen that the 5x5 generates more blurring in the image than the 3x3 filter. In fig.1 there was some motion detected on the top of the image, in places where there aren't any objects moving. However, in Fig.6 these false positives are reduced significantly but not entirely. Moreover, when the Gaussian filter with sigma 2 was applied (Fig.8) all the false positives were removed. More blurring, but not excessive, made it possible to detect motion more accurately. In addition, Gaussian filters with different values of sigma were applied to see the effect sigma has on the resulting images. By looking at Fig.7 that is the output of the gaussian filter with sigma 1, Fig.8 with sigma 2, Fig.9 with sigma 4. With sigma 1 we can see that we have a lot of false positives and with sigma 4 we lose some of the pixels that should be identified as an object in motion. Therefore, we can assume that there is an ideal value for sigma around 2 that will give us the most motion detection with the least false positives. If sigma goes below this value the false positives will increase and if it goes above we will start losing motion detection accuracy.

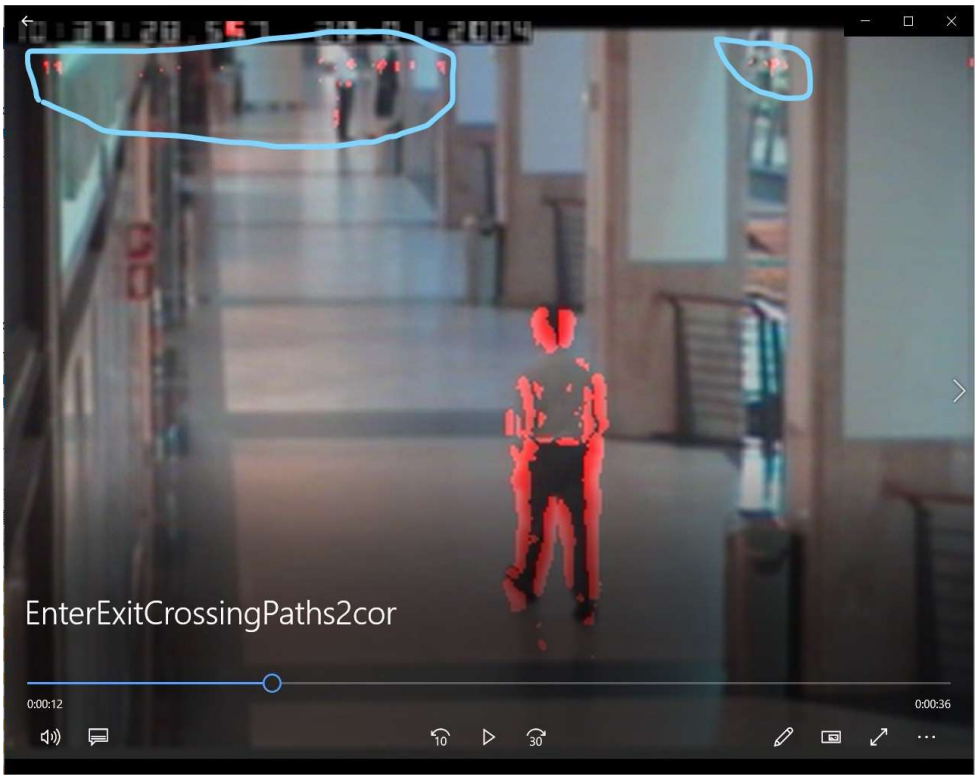

Fig.5 3x3 Filter

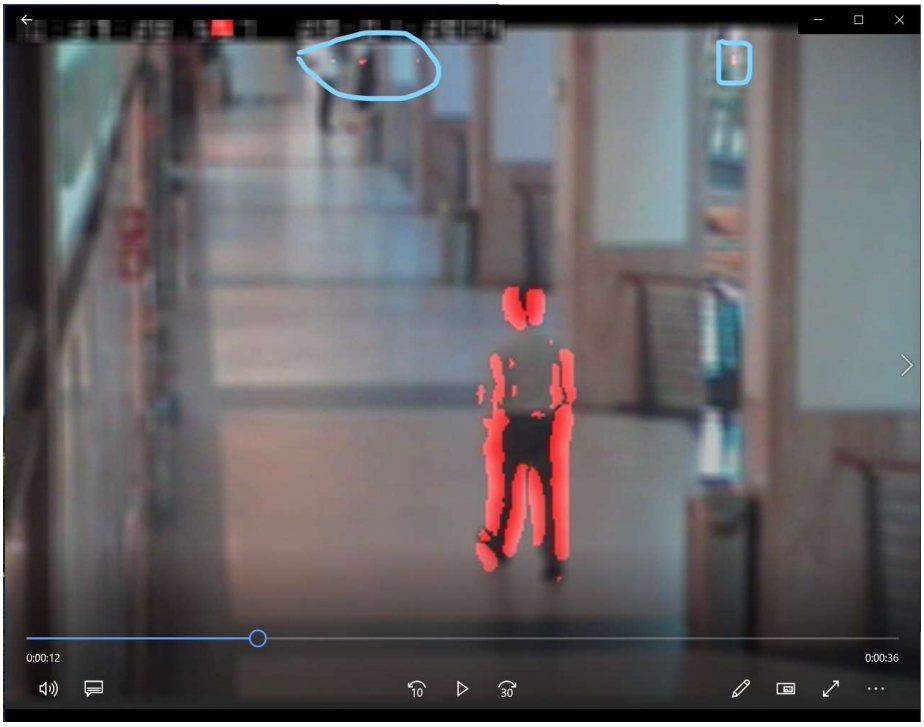

Fig.6 5x5 filter

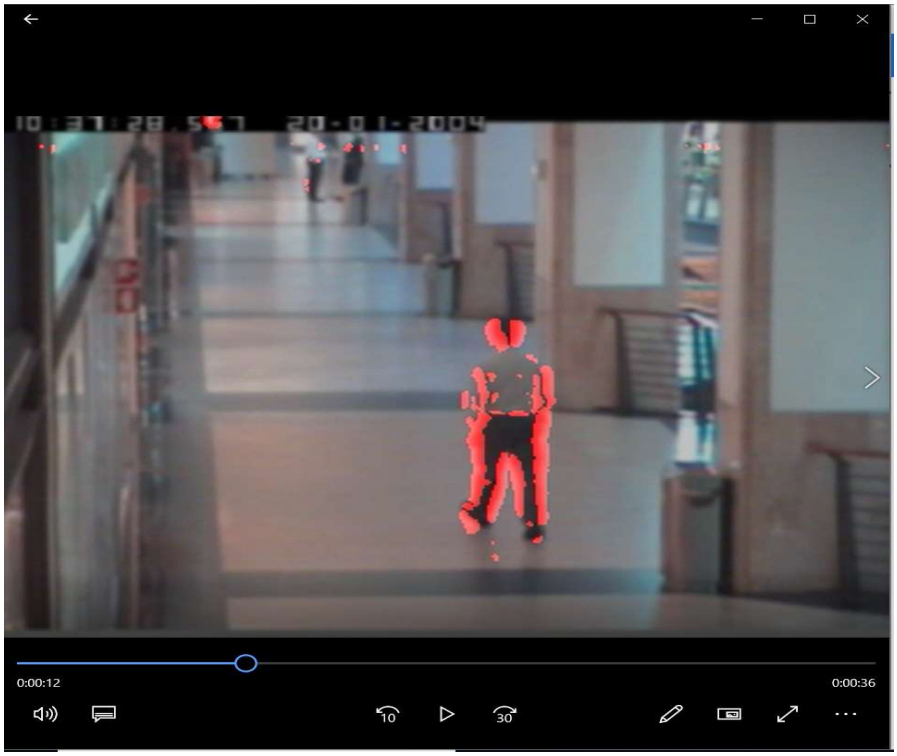

Fig.7 Gaussian with Sigma 1

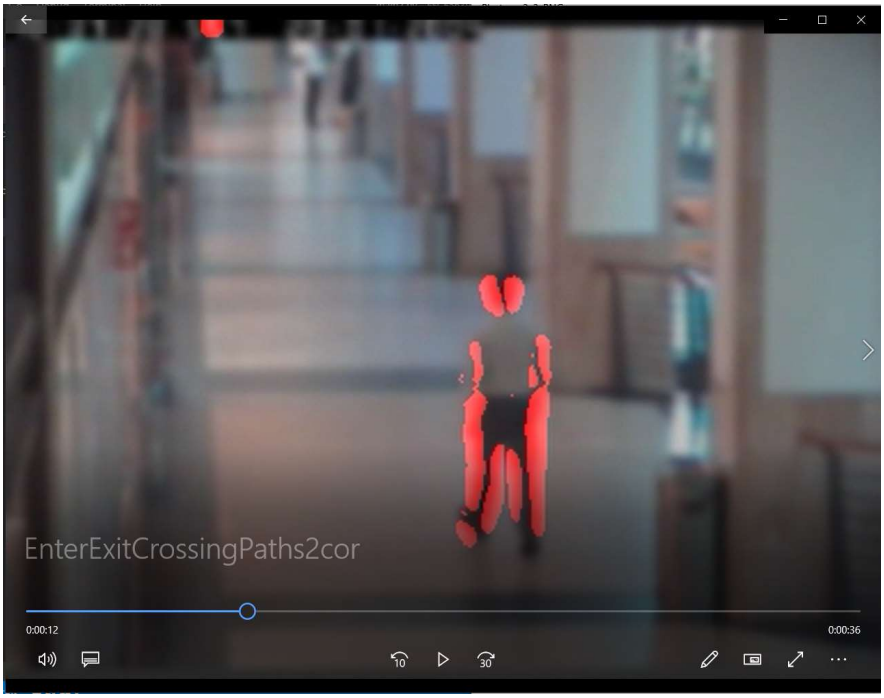

Fig.8 Gaussian with Sigma 2

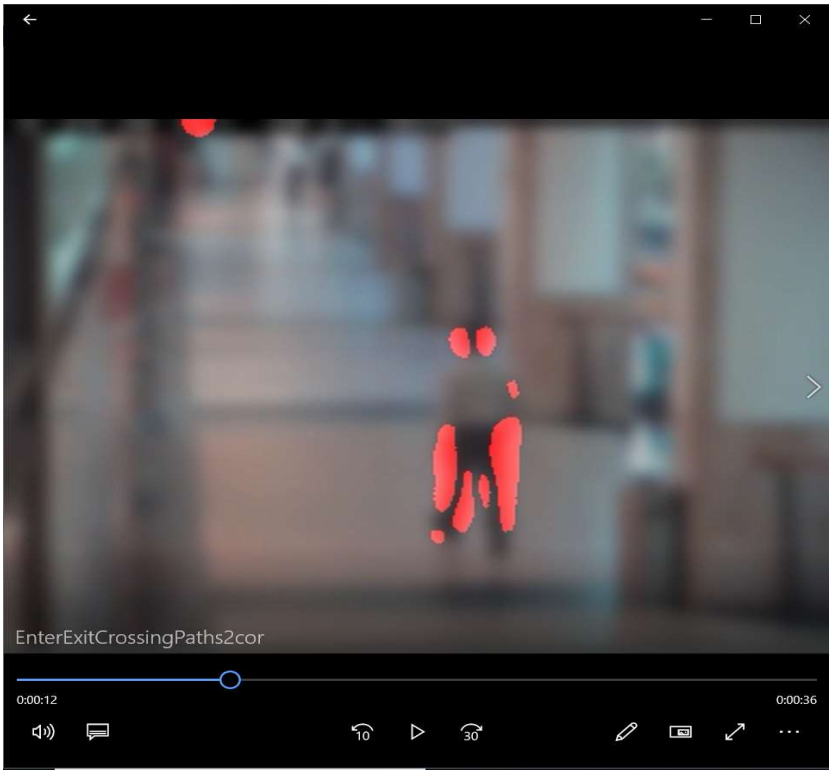

Fig.9 Gaussian with Sigma 4

Modifying the Threshold:

The third experiment was to modify the Threshold values to see the effect it has on motion detection. The threshold value is the parameter that determines where there is an edge after applying the temporal derivative filter. If the value is higher than this threshold, meaning that there is a big change in the pixel over time, the program will detect an edge or motion. If the threshold is too low we will have a lot of false positives since the program will detect that there is an edge everywhere there is noise or motion. To reduce the negative effect that the smoothing filter and the temporal derivative filter have on the motion detection, the Gaussian derivative filter and the 2D spatial Gaussian filter were applied. As seen in the previous experiments this were the best options. In Fig 13 where the threshold was 1, it can be seen that there is motion detected everywhere. Then the threshold was increased to 5 (Fig.12) and there was a lot of false positives reduction but there still were some on the top and surrounding the man outside of the moving object. When the threshold was set to 10 (Fig.11) we found that this value was very close to the ideal value since the motion is very centralized on the man, that is our moving object, and there were no false positives. Finally, the threshold was increased to 20 and a lot of motion detection was lost. In Fig.10 it can be seen that the torso and arms of the man are not being detected as moving objects.

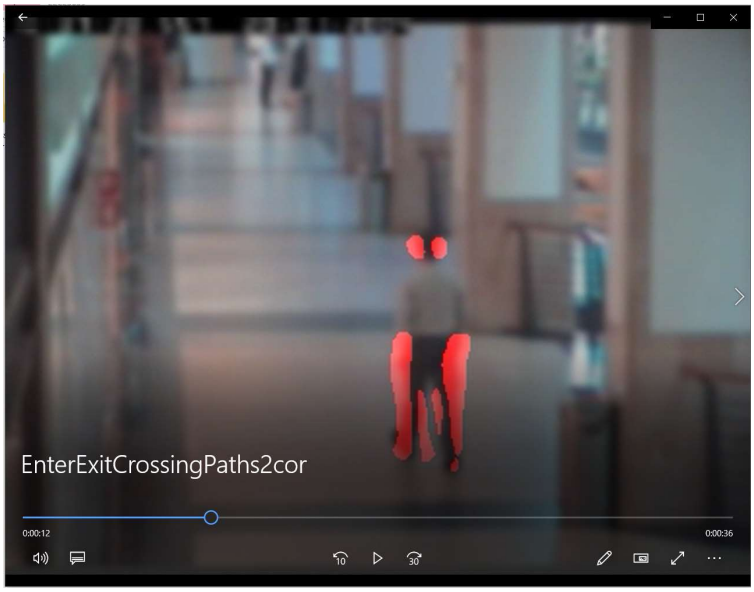

Fig.10 threshold of 20

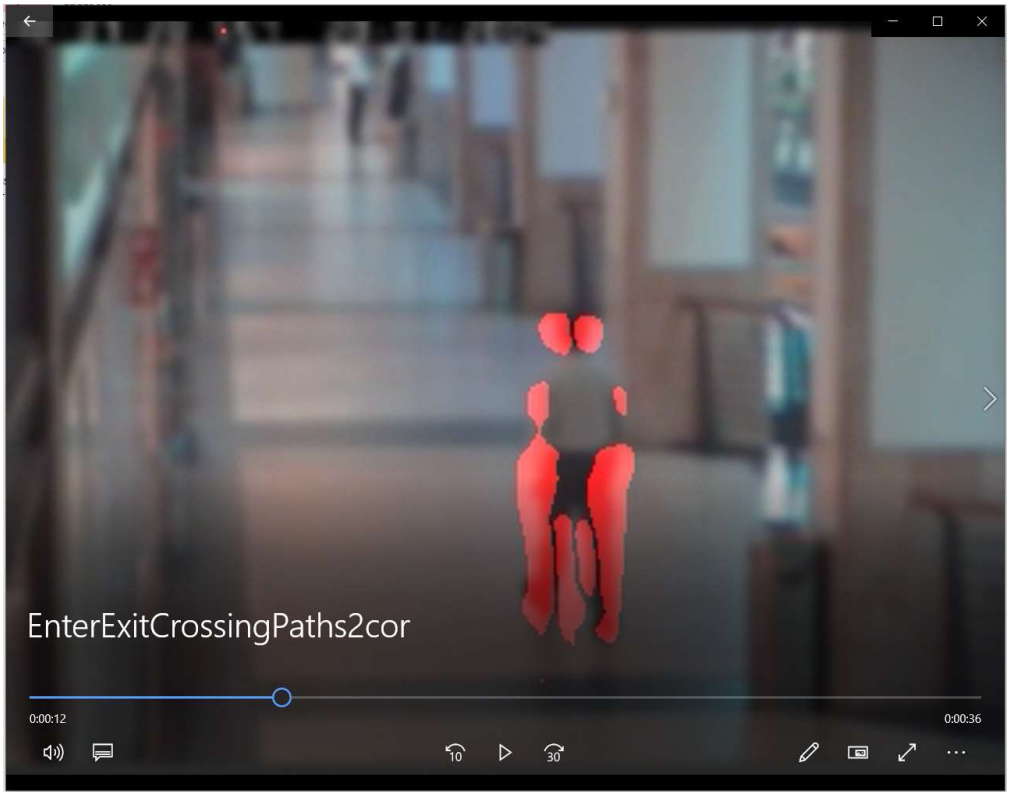

Fig.11 threshold of 10

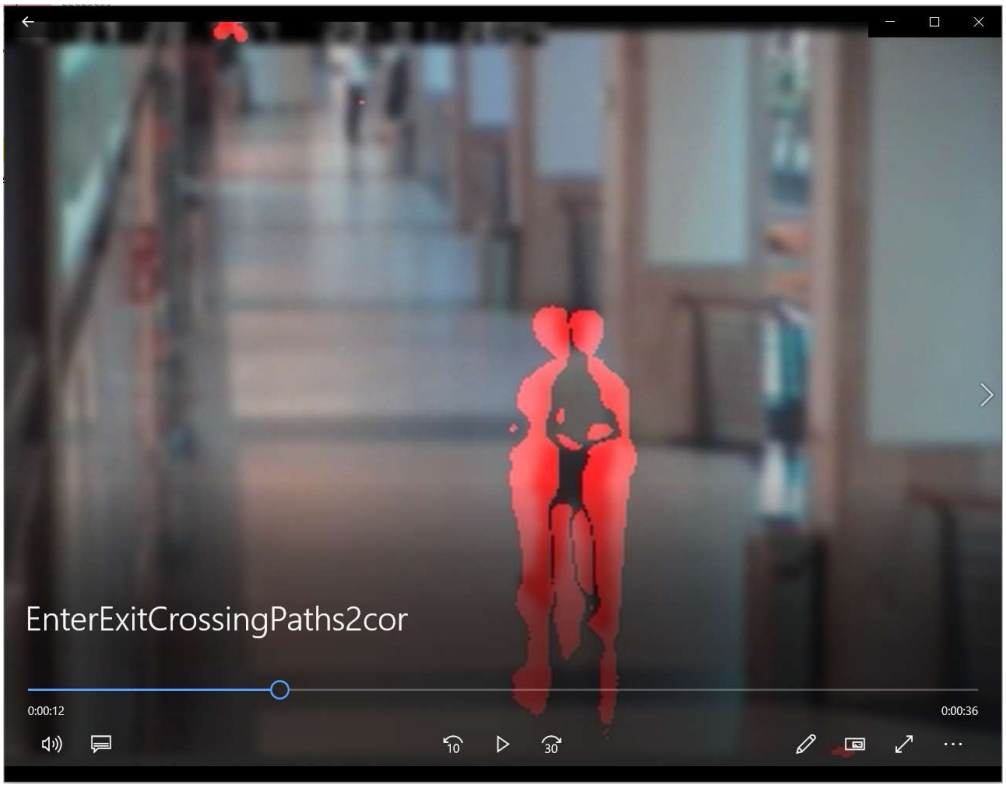

Fig.12 Threshold of 5

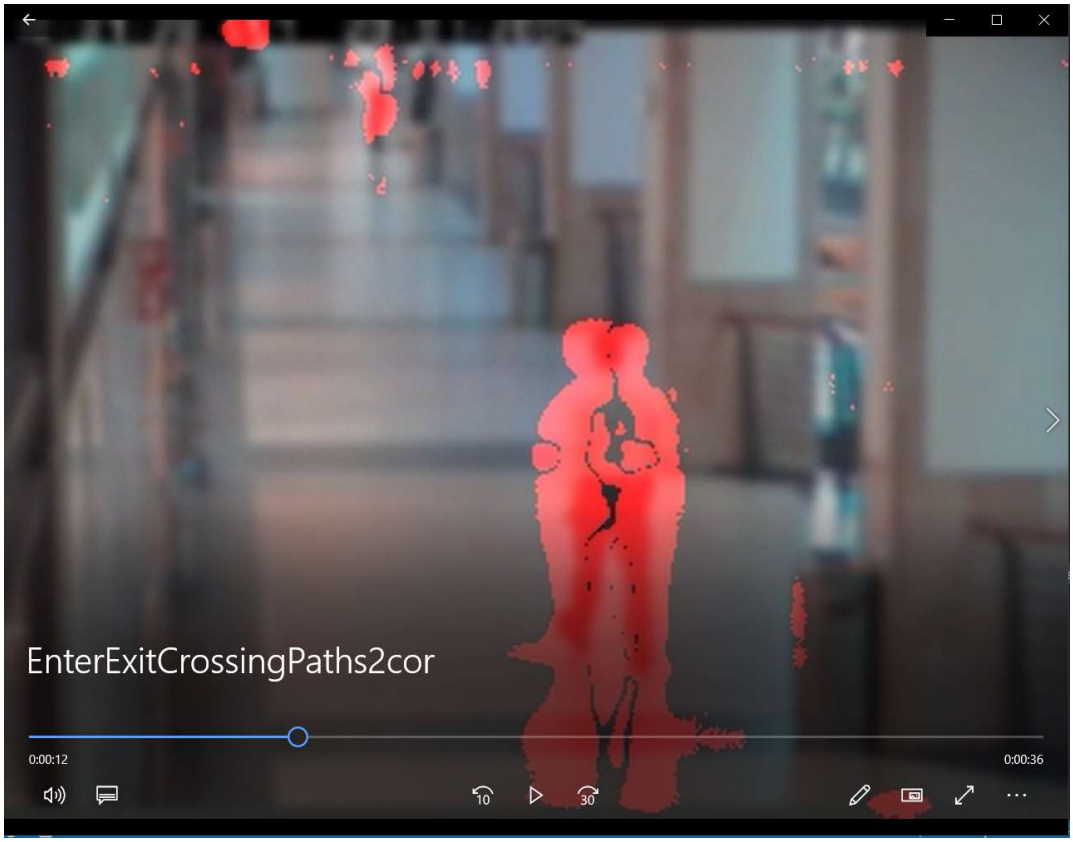

Fig.13 Threshold of 1

Values of parameters used

Sigma of Gaussian derivative: we chose a value of 10 to make sure that the motion is detected properly and the computational cost is not too high.

Sigma of 2D spatial Gaussian filter: we choose the value of 2 based on trial and error. We don't

Threshold: This value of sigma is the same or very close to the value of sigma from the Gaussian noise found in the derivative of the image. By using a noise estimation algorithm we found that for the 3 videos provided were:

Red chair: 5.0029

Office: 3.1

EnterExitCrossingPaths2cor:3.1181

Therefore, we chose for the red chair a value of 10, for the office a value of 8 and for the EnterExitCrossingPaths2cor a value of 8. A random frame of the final output of the 3 videos was chosen

to show the results using these parameters. Fig.14 the output from EnterExitCrossingPaths2cor is very accurate there aren't false positives and the man is detected as the only moving object. Fig.15 the output from Red chair is not very good but it was as good as we could make it. The problem is that the quality of the video is very low. There weren't enough frames per second and the man and the chair jumped from one place to another from frame to frame. Fig.16 is the output from the office. There is not much motion detected in one frame because the object is moving too slow. But overall in the video the motion can be seen.

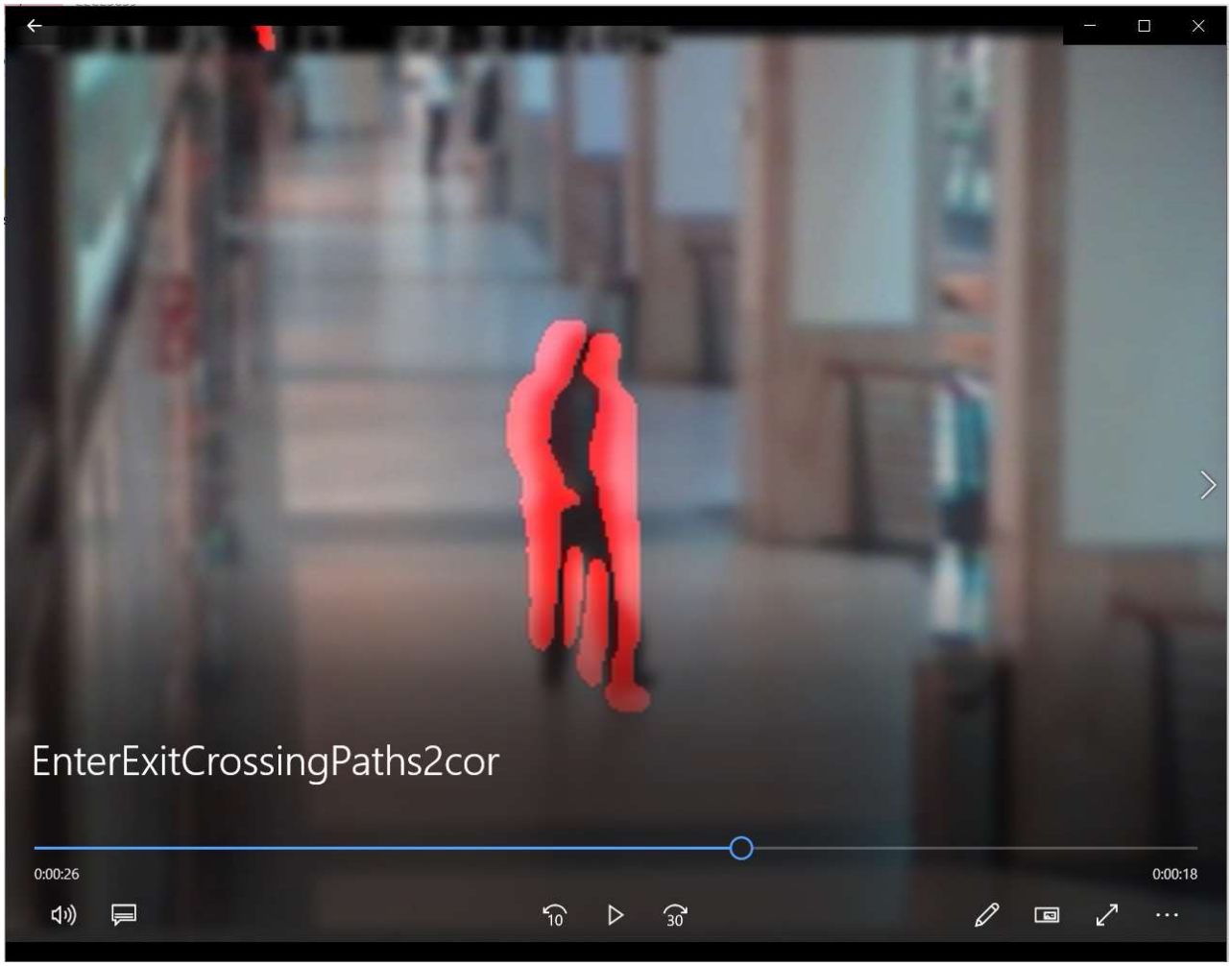

Fig.14 EnterExitCrossingPaths2cor

Fig.15 Red Chair

Fig.16 Office

Conclusions

In conclusion blurring the image before finding the derivative was very useful in removing noise and having a more clean and clear result. Estimating the noise was also very helpful in choosing a correct threshold.

Additionally using a 1D derivative of a gaussian filter instead of the Sobel filter helped to smooth out the result and make it more stable.

The simple operation of applying a temporal derivative filter was quite effective in highlight moving objects. However it only works when there is a static background as shown when the lights were switched on in the Office and RedChair videos. Even though nothing moved, the whole image became red because there was a sharp gradient at the time the light was turned on. So this method is easy to implement but is not very robust for real world non-specific applications.

Appendix

```
import numpy as np
from matplotlib import pyplot as plt
import math
import cv2
import os
import gc


# Find the noise estimate of a set of images
def estimate_noise(imset):
    avg = np.average(imset, axis=0)

    sig = np.zeros_like(imset[0], dtype=np.float64)

    for n in range(len(imset)):
        sig += (avg - imset[n].astype(dtype=np.float64)) ** 2

    sig = (sig / (len(imset) - 1)) ** (1/2)

    avg_sigma = np.average(sig)

    return sig, avg_sigma


# Computes the 1-D derivative of a gaussian filter given an input sigma
# Normalizes the filter so that the sum of the magnitude of each element equals 1
def get_gaussian_derivative(sdev):
    filter_size = math.ceil(sigma * 5)

    if filter_size % 2 == 0:
        filter_size += 1

    gaussian_filter = np.empty(filter_size)
    mid = math.floor(filter_size / 2)

    for i in range(filter_size):
        gaussian_filter[i] = -(i - mid) * math.exp((-(i - mid) ** 2) / (2 * sdev ** 2))

    gaussian_filter /= sum(np.fabs(gaussian_filter))
    return gaussian_filter


dataset_path = ["RedChair", "EnterExitCrossingPaths2cor", "Office"]
```

```
sigma = 2.0

# Calculate the kernel size based on sigma
kernel_size = math.ceil(5 * sigma)
if kernel_size % 2 == 0:
   kernel_size += 1

for path in dataset_path:
   print(path + '/' + path + '/')

   if path == "RedChair":
       threshold = 10
   elif path == "EnterExitCrossingPaths2cor":
       threshold = 8
   elif path == "Office":
       threshold = 8

   # Open each image in the specified folder and save them in imgset
   imgset = []
   for serial_number in os.listdir(path + '/' + path + '/'):
       imgset.append(cv2.imread(path + '/' + path + '/' + serial_number))

   w, h = imgset[0].shape[1], imgset[0].shape[0]

   # Blur all the images in imgset

   # imgset = [cv2.blur(frame, (3, 3)) for frame in imgset]
   # imgset = [cv2.blur(frame, (5, 5)) for frame in imgset]
   imgset = [cv2.GaussianBlur(frame, (kernel_size, kernel_size), sigma) for frame in imgset]

   # Convert all the images to grayscale
   grayscale = [cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) for frame in imgset]

   derivative_set = [np.zeros_like(frame, dtype=np.float32) for frame in grayscale]
   res = imgset.copy()

   # kernel = 0.5 * np.array([-1, 0, 1])   # Sobel Filter
   kernel = get_gaussian_derivative(5.0)   # 1-D Gaussian derivative filter

   kernel_mid = math.floor((len(kernel) / 2))
   for i in range(kernel_mid, len(imgset) - kernel_mid):
       # Calculate the derivative based on the filter
       for k in range(len(kernel)):
           derivative_set[i] += grayscale[i + k - kernel_mid].astype(np.float32) * kernel[k]
       derivative_set[i] = np.fabs(derivative_set[i])

       # Threshold the derivative
       _, thresh = cv2.threshold(derivative_set[i], threshold, 255, cv2.THRESH_BINARY)
       thresh = thresh.astype(np.uint8)
```

```
        # Create a red image to mask the derivative onto the original image
        redImg = np.zeros(imgset[0].shape, imgset[0].dtype)
        redImg[:, :] = (0, 0, 255)
        redMask = cv2.bitwise_and(redImg, redImg, mask=thresh)

        # Add the red mask with the original image at 75% intensity
        cv2.addWeighted(redMask, 1, res[i], 0.75, 0, res[i])

    # Deallocate memory
    del grayscale
    del imgset
    gc.collect()

    # Estimate the noise of the derivative
    # 5.0029 - RedChair
    # 3.1881 - EnterExit
    # 3.1 - Office
    # noise, noise_avg = estimate_noise(derivative_set)
    # print(noise_avg)

    # Save the set of images as a .mp4v video
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    video = cv2.VideoWriter(path + ".mp4v", fourcc, 10, (w, h), isColor=0)
    for img in res:
        video.write(img)
```