EECE 5639 - Project 1

Motion Detection Using Simple Image Filtering

Kevin Russell Eugen Feng

2023-02-19

Abstract

In this project, we explored a simple technique for motion detection by evaluating captured image sequences from a stationary camera and background with relatively small moving objects. For our method, we evaluated large gradients in the temporal evolution of pixel values and observed intensity values at a pixel over time. We implemented various image processing algorithms such as box filtering, temporal filtering, Gaussian filtering 1D, Gaussian filtering 2D, and simple absolute difference of the previous image in the sequence. Our report describes implementation of our program, experiments we conducted, the tuning parameters we used, observations, and conclusions.

Description

Our project explored converting the images in gray scale, apply a 1D differential operator at each normalized pixel between the current image and the previous image, and computed the temporal derivative $|x-y|$ between the two sequences of images. We applied a threshold to the derivatives where we created a zero and one mask that outlines the intensity values of the derivatives. The mask is blended with the original image with 0.3 alpha value of opacity to outline clusters of points in a green hue.

Temporal Derivative

For temporal derivative, we attempted 100 different trials to two consecutive frames 59 and 60 for both office and red chair that adjusted the threshold between 0.001 to 0.1 by an increment of 0.001. For the office frames, we observed that the images got really noisy between 0.001 to 0.01 threshold, and tapers quickly at 0.05 threshold. The sweet spot is 0.03 threshold, which you can see in Figure 1. As for red chair, images were really noisy between 0.001 to 0.03 threshold, but retained majority of motion even to the 0.1 threshold.

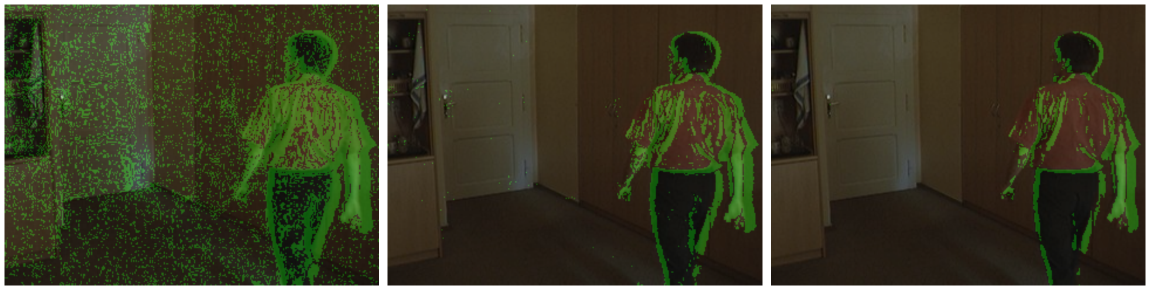

Figure 1: *[Office Photo Sequence] A man walking in a room with a green hue that represents motion. (left photo) derivatives was threshold at 0.01, which made the image very noisy. (center photo) derivatives was threshold at 0.03, which had some noise. (right photo) derivatives was threshold at 0.05, which lost motion detail*

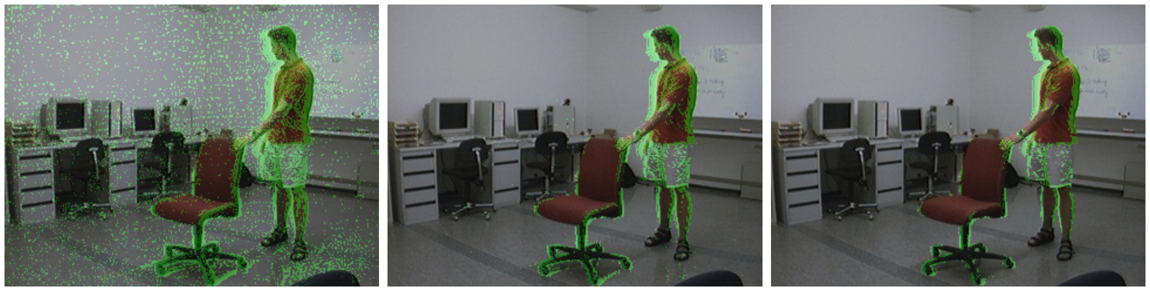

Figure 2: *[Red Chair Photo Sequence] A man pushing a chair with a green hue that represents motion. (left photo) derivatives was threshold at 0.03, which made the image very noisy. (center photo) derivatives was threshold at 0.06, which had some noise. (right photo) derivatives was threshold at 0.1, which retained a good amount of detail*

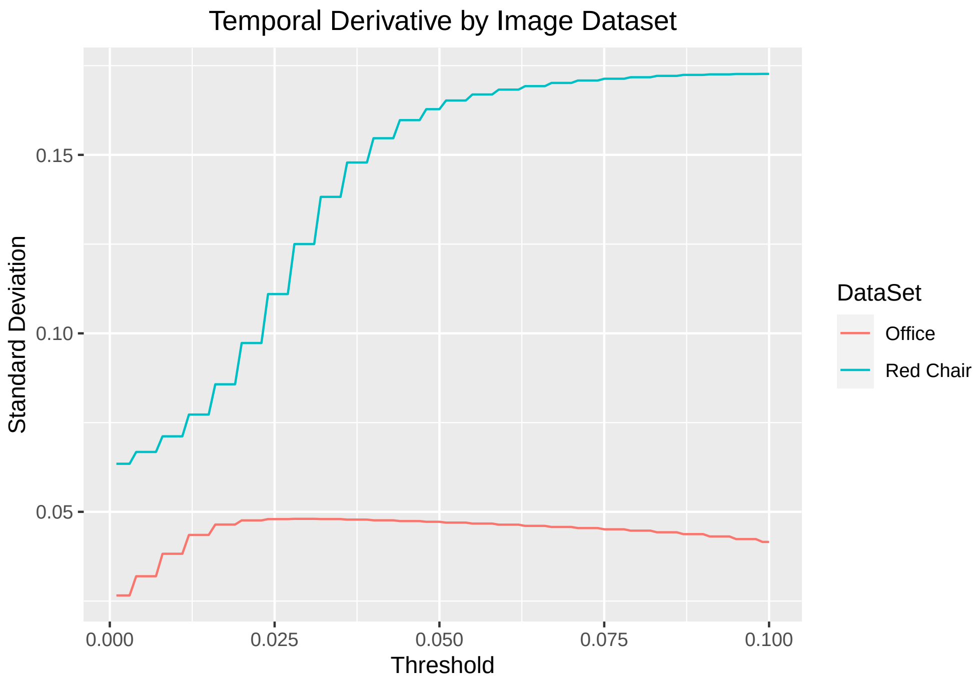

Figure 3: *A comparison of noise (standard deviation) between the Office and Red Chair data sets among threshold 0.001 to 0.1*

Simple Filter with Temporal Derivative

Following the same formula as the temporal derivative, we applied a simple $(-1, 0, +1)$ filter by leveraging the convolve function from signal before the temporal derivative. A temporal derivative filter is a type of filter that is used to calculate the rate of change of a signal over time. It is commonly used in signal processing and computer vision applications, where it is often used to detect edges and other features in an image or video. The basic idea behind a temporal derivative filter is to calculate the difference between two adjacent frames of a video sequence, or two adjacent samples in a time-series signal. This difference is then divided by the time interval between the two frames or samples, giving the rate of change of the signal at that point in time. The convolve function leverages the $g(x,y) = \sum_{s=0}^{a} \sum_{t=0}^{b} w(s,t) * I(x+s, y+t)$ where $a = \frac{m-1}{2}; b = \frac{n-1}{2}$. Again, we attempted 100 different trials to two consecutive frames 59 and 60 for both office and red chair image sequences. We applied a threshold of 0.001 to 0.1 to the intensity derivative values, and blended the mask with the original at a 0.3 alpha. We found that the office had moderate amount of noisy between 0.001 to 0.007 threshold, had a clean highlight of motion at 0.01 threshold, and had very little motion detected at 0.02 threshold. As for the red chair data set, images were extremely noisy between 0.001 to 0.01 and lost a lot of detail at 0.05 threshold, which the sweet spot is 0.03 threshold.

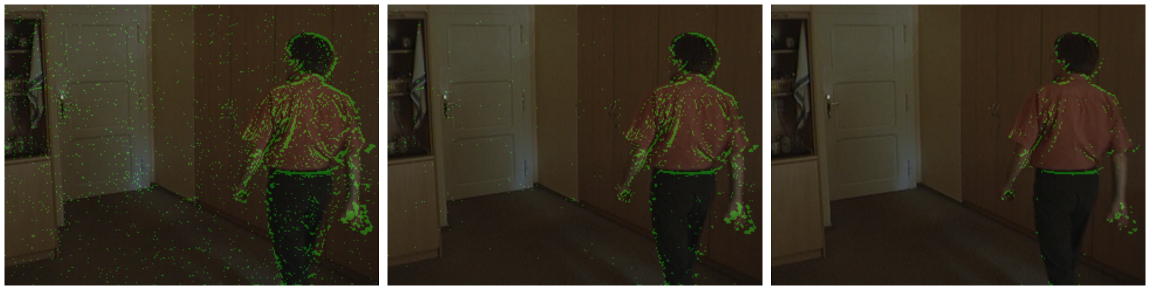

Figure 4: *[Office Photo Sequence] A man walking in a room with a green hue that represents motion. (left photo) derivatives was threshold at 0.007, which made the image very noisy. (center photo) derivatives was threshold at 0.01, which had some noise. (right photo) derivatives was threshold at 0.02, which lost alot of detail*

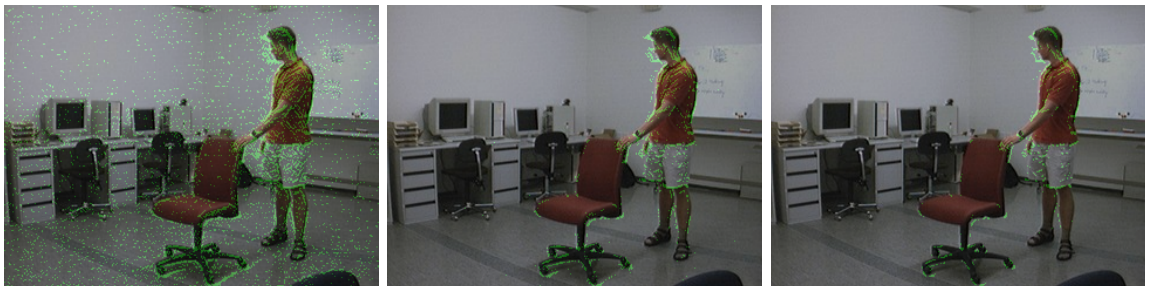

Figure 5: *[Red Chair Photo Sequence] A man pushing a chair with a green hue that represents motion. (left photo) derivatives was threshold at 0.01, which made the image very noisy. (center photo) derivatives was threshold at 0.03, which had some noise. (right photo) derivatives was threshold at 0.05, which lost alot of detail*

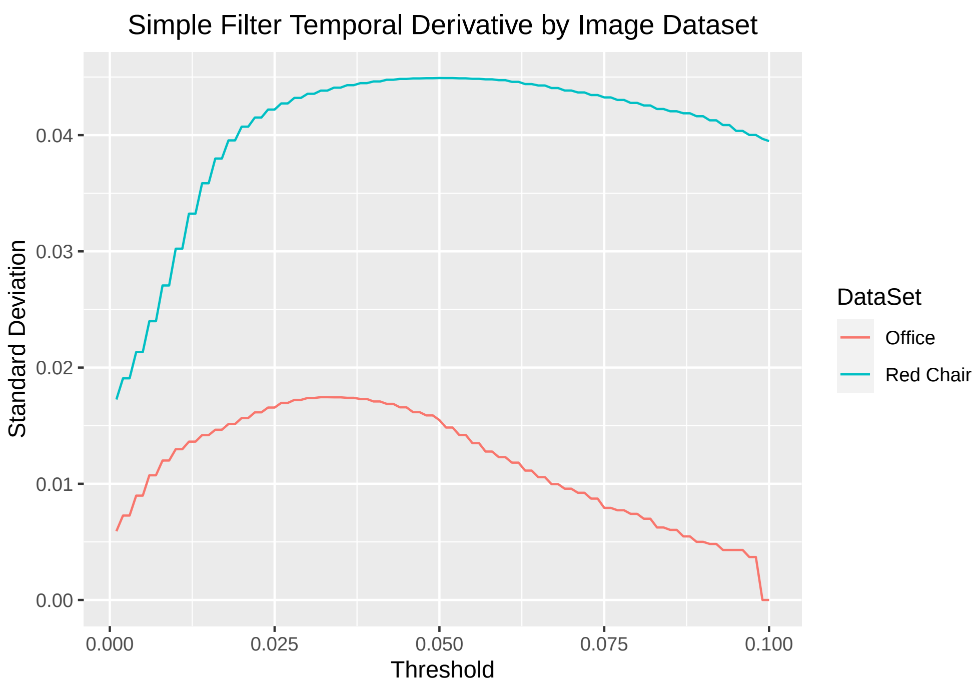

Figure 6: *A comparison of noise (standard deviation) between the Office and Red Chair data sets among threshold 0.001 to 0.1*

Gaussian 1D Filter with Temporal Derivative

Like the temporal derivative method, we applied a Gaussian 1D filter before the temporal derivative. In a 1D Gaussian filter, the filter coefficients are generated by computing a 1D Gaussian function centered at the origin. The filter kernel is then constructed by discretizing this function and normalizing the coefficients so that they sum to 1. The resulting kernel can be convolved with the signal to produce a smoothed output. The amount of smoothing that a Gaussian filter provides is controlled by a parameter called the standard deviation indicated by sigma, which determines the width of the bell-shaped curve. A larger standard deviation will result in a wider, smoother curve, while a smaller standard deviation will result in a narrower, sharper curve. We leveraged the following equation for 1D gaussian first derivative:

$$g(x) = e^{\frac{-x^2}{2\sigma^2}}$$
$$g'(x) = (\frac{-2x}{2\sigma^2})(e^{\frac{-x^2}{2\sigma^2}})$$
$$g'(x) = \frac{-x}{\sigma^2} e^{\frac{-x^2}{2\sigma^2}}$$

We experimented with 2000 experiments with sigma between 1 to 20 and the threshold between 0.001 to 0.1. We found that the odd sigma's yielded very noisy images whereas odd sigmas yielded cleaner pictures with the threshold being very high at 0.1 for the office sequence images. As for red chair, the images were extremely noisy with odd sigmas with cleaner pictures at higher thresholds for even sigmas at 0.1. Lastly, sigma played a bigger role than threshold

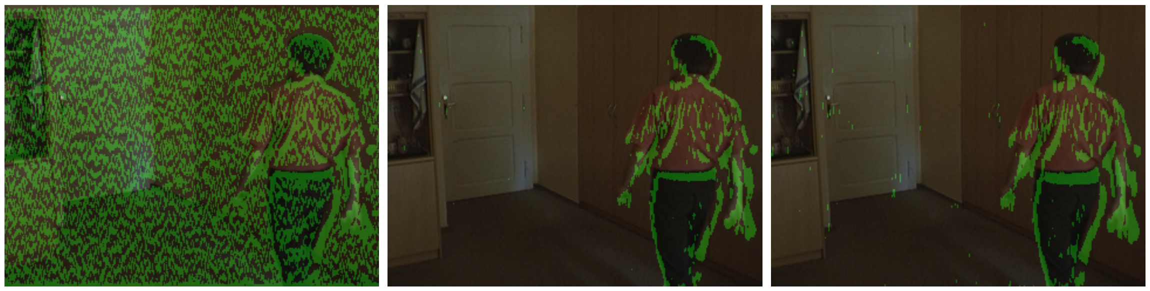

Figure 7: *[Office Photo Sequence] A man walking in a room with a green hue that represents motion. (left photo) Sigma at 3 and derivatives was threshold at 0.1, which made the image very noisy. (center photo) Sigma at 4 and derivatives was threshold at 0.1, which had some noise. (right photo) Sigma at 6 and derivatives was threshold at 0.1, which had some noise*

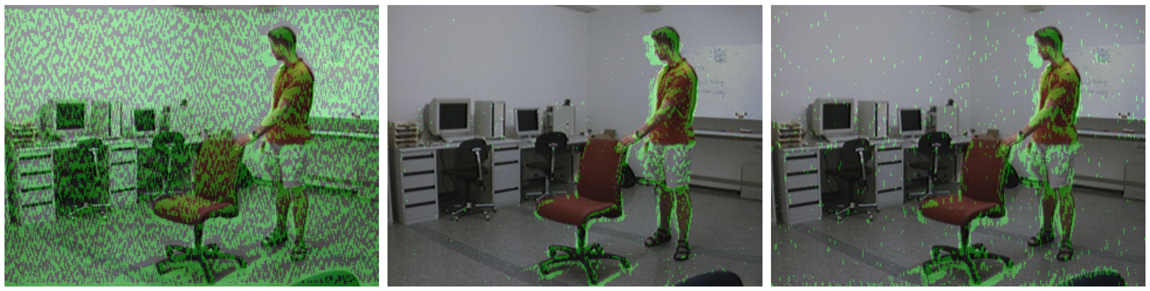

Figure 8: *[Red Chair Photo Sequence] A man pushing a chair with a green hue that represents motion. (left photo) Sigma at 3 and derivatives was threshold at 0.1, which made the image very noisy. (center photo) Sigma at 2 and derivatives was threshold at 0.1, which had some noise. (right photo) Sigma at 4 and derivatives was threshold at 0.1, which had alot of noise*

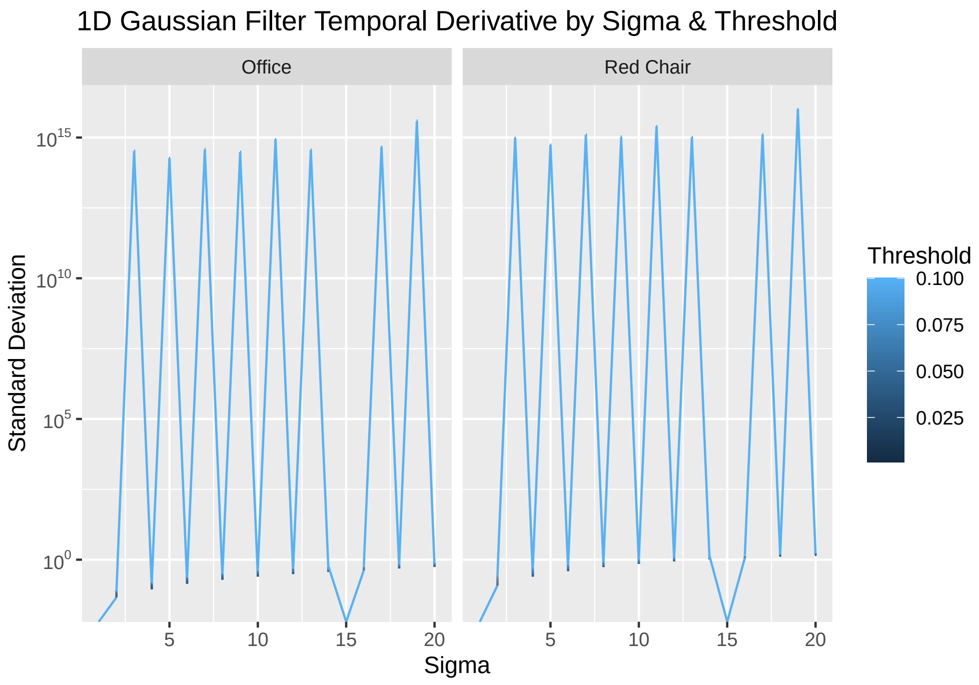

Figure 9: *A comparison of noise (standard deviation) between the Office and Red Chair data sets among sigmas 1 to 20*

Box Filter with Temporal Derivative

Like the Simple Filter on temporal derivative, we applied a box filter instead of a simple filter before the temporal derivative by leveraging the convolve function. The box filter is a linear filter that replaces each pixel in the image with a weighted average of its neighboring pixels, where the weights are equal to 1 over the size of the kernel. To apply the box filter to an image, we slide the kernel over the image, pixel by pixel, and compute the weighted average of the neighboring pixels within the kernel. The resulting value is then assigned to the center pixel in the kernel, which represents the smoothed version of the original pixel. The size of the kernel determines the degree of smoothing applied to the image. A larger kernel size results in a stronger smoothing effect, while a smaller kernel size produces a milder effect. We ran 1000 trials with odd filter sizes between 1 to 19 and threshold between 0.001 to 0.1 at 0.001 increments. For box filter, the motion detection is pretty smooth with little noise. Same can be said of the red chair images.

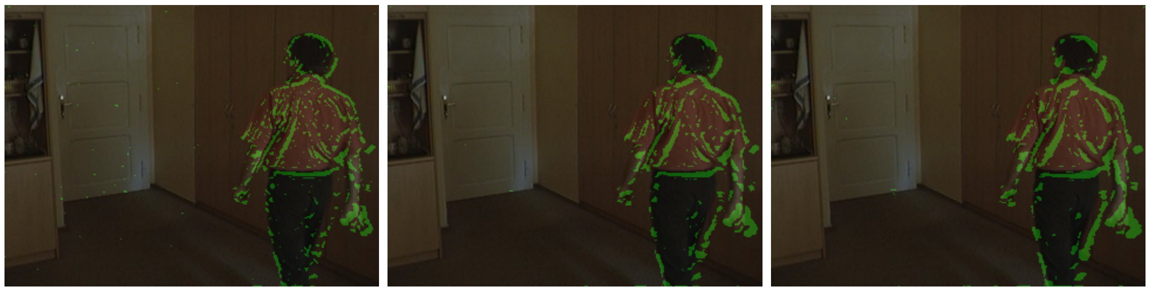

Figure 10: *[Office Photo Sequence] A man walking in a room with a green hue that represents motion. (left photo) Filter is 3x3 and derivatives threshold at 0.005. (center photo) Filter is 5x5 and derivatives threshold at 0.003. (right photo) Filter is 7x7 and derivatives threshold at 0.002*

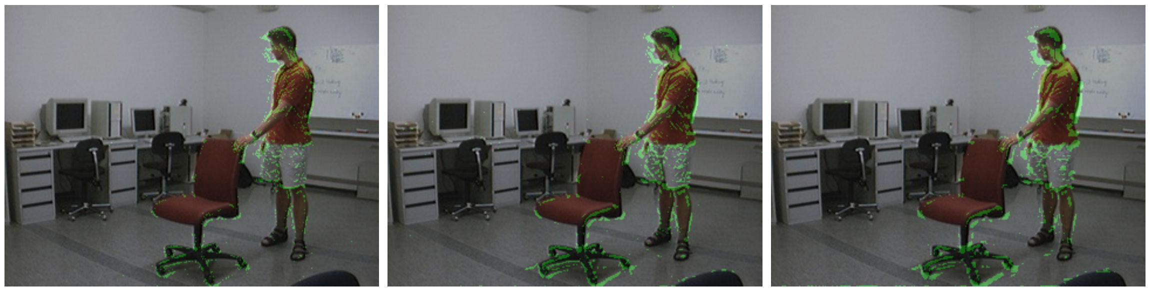

Figure 11: *[Red Chair Photo Sequence] A man pushing a chair with a green hue that represents motion. (left photo) Filter is 1x1 and derivatives threshold at 0.03. (center photo) Filter is 3x3 and derivatives threshold at 0.01. (right photo) Filter is 5x5 and derivatives threshold at 0.006*

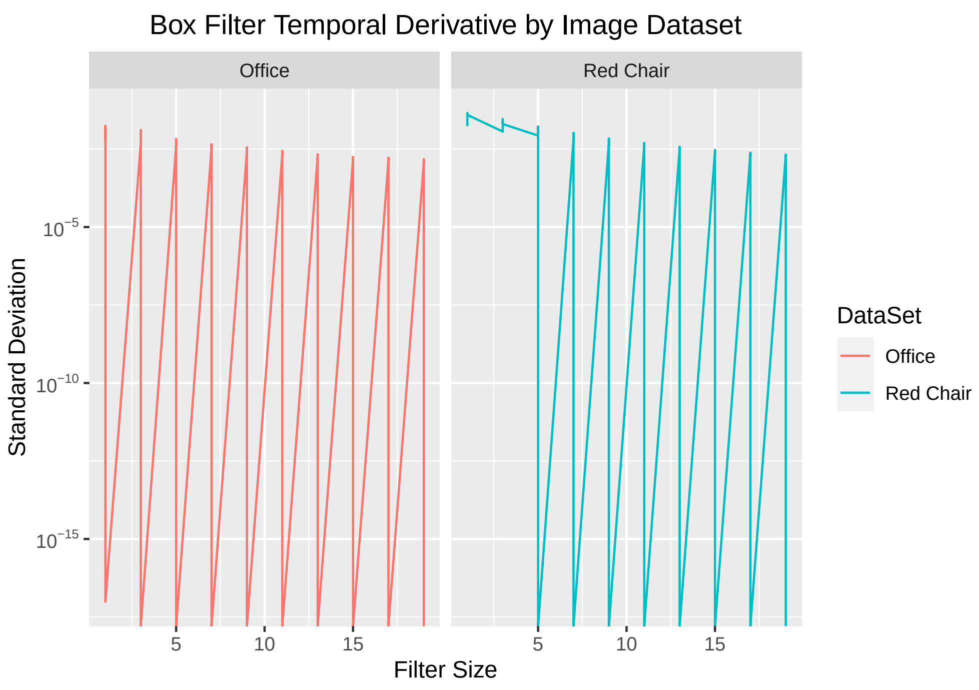

Figure 12: *A comparison of noise (standard deviation) between the Office and Red Chair data sets among odd filters 3 to 19*

Gaussian 2D Filter with Temporal Derivative

Like the Gaussian 1D method, we applied a Gaussian 2D filter before the temporal derivative. A 2D Gaussian filter is used to smooth or blur images by convolving the image with a Gaussian kernel. To apply the filter to an image, we first convolve the image with a kernel that is created by multiplying two 1D Gaussian functions in the horizontal and vertical directions. The resulting 2D kernel is then used to smooth the image by taking a weighted average of the neighboring pixels in the image. The size of the kernel and the value of sigma determine the amount of blurring or smoothing that is applied to the image. We used the following Gaussian formula for our algorithm:

$$g(x,y) = (e^{\frac{-x^2}{2\sigma^2}})(e^{\frac{-y^2}{2\sigma^2}})$$
$$g(x,y) = e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$
$$\frac{\partial g(x,y)}{\partial xy} = \frac{xy}{\sigma^4}e^{\frac{-x^2}{2\sigma^2}}$$

For our trials, we ran 20000 experiments with odd box filters between 1 to 19, sigma between 1 to 20, and the threshold between 0.001 to 0.1. We found that the 2D gaussian filter cleaned a lot noise in the images except for filter sizes 11x11, 13x13, 15x15, and 19x19. We found that pictures either had extremely heavy noise or no noise at all. One common theme was that we found lesser noisy images with the office sequence data set compared to the red chair sequence data set. Unfortunately, we saw no images with motion detected, but rather an all or nothing for noise.

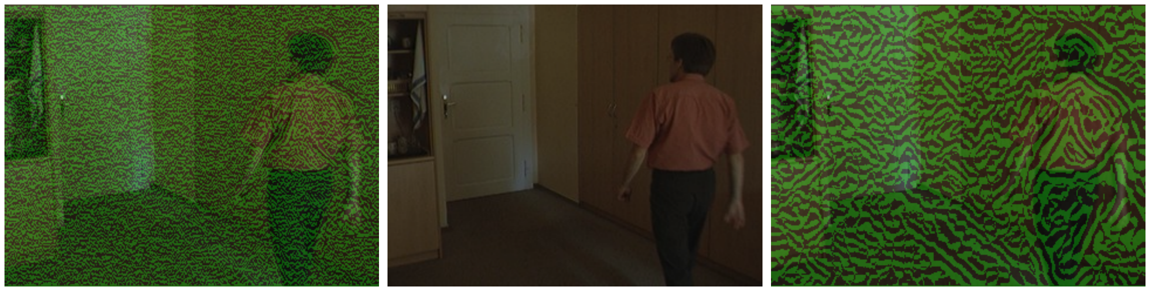

Figure 13: *[Office Photo Sequence] A man walking in a room with a green hue that represents motion. (left photo) Filter is 11x11, sigma is 1, and derivatives threshold at 0.001. (center photo) Filter is 3x3, sigma is 3, and derivatives threshold at 0.001. (right photo) Filter is 17x17, sigma is 4, and derivatives threshold at 0.029*

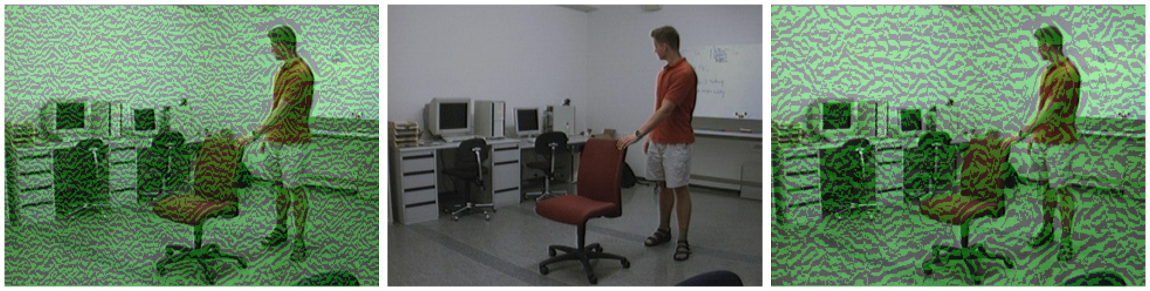

Figure 14: *[Red Chair Photo Sequence] A man pushing a chair with a green hue that represents motion. (left photo) Filter is 11x11, sigma is 2, and derivatives threshold at 0.001. (center photo) Filter is 3x3, sigma is 3, and derivatives threshold at 0.001. (right photo) Filter is 15x15, sigma is 4, and derivatives threshold at 0.001*

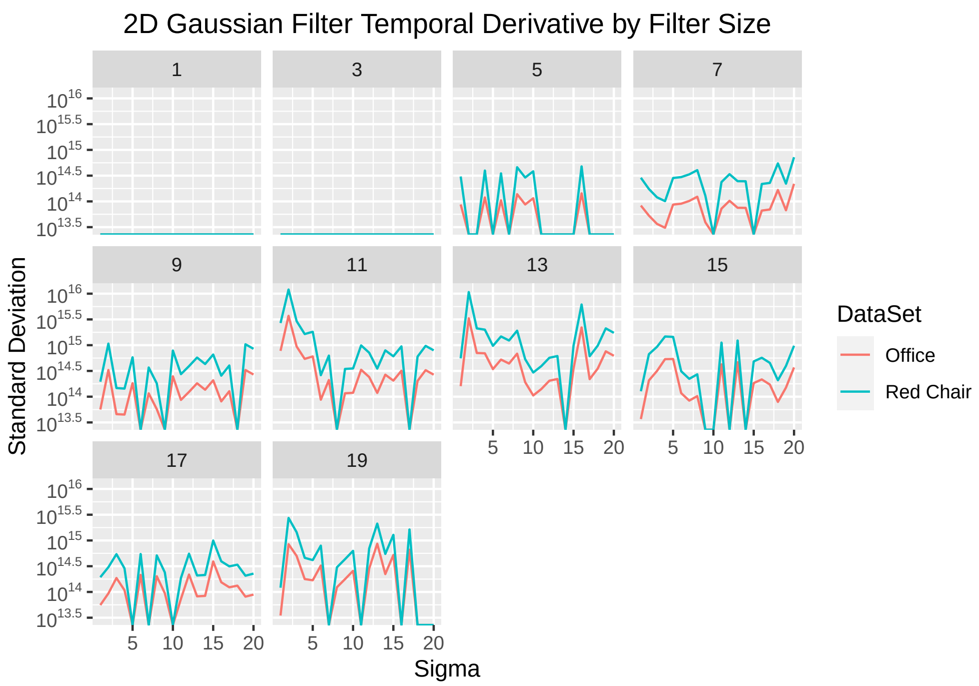

Figure 15: *A comparison of noise (standard deviation) between the Office and Red Chair data sets among sigmas 1 to 20 and odd filters 3 to 19*

Observations

In our experiments, we got to understand temporal derivatives as a means to detecting motion. We found that the temporal derivative method did detect motion, but there was a lot of noise in the red chair images. The more noisier images for red chair could be due to the quicker movements from the previous image to the current image that allowed more noise to be captured. This makes sense considering that quicker movements should yield bigger derivatives from the last frame's position to the current frame's position.

The simple filter with derivative was examined, thereafter, that showed a more refined handling of noise in motion detection. The biggest difference between the temporal derivative with and without a simple filter is the more granular noise patterns found in motion when using a simple filter. We ended up plotting the standard deviation for both methods and found that the temporal derivative with filter had a lot less noise in comparison to a temporal derivative without a filter.

Next, we examined the 1D Gaussian filter applied with the temporal derivative, which we were most fascinated about. We were curious whether noise would be filtered out just enough for great motion detection. After applying the filter, we found that the Gaussian filter tend to have an extreme amount of noise for odd sigmas or just the right amount of noise for motion detection. However, sigma did play a factor in retaining noise as the sigma increases.

Box Filtering, on the other hand, had very little noise for most filter sizes. We saw an increase of noise for size 1 and 3 for filter size, but size 5 in filter had the least amount of noise in the images. By examining the images, filter size 1 and 3 had less details of motion and more noise in comparison to size 5 filter. Additionally, the threshold added an extra layer of refining the noise from the motion in the image.

Lastly, the Gaussian 2D filter with temporal derivative was the most disappointing among all filters. We found that images were either extremely noisy or no noise at all. It is as if the Gaussian 2D eliminated noise all together or left really high derivatives of noise in the images. One interesting observation is that the noise gets wavy as the filter size and sigma increases.

Conclusion

Overall, the project provided an exciting hands-on experience on basic motion detection, and allowed us to understand how different filters react towards handling noise. We were also able to understand the tuning parameters better and understand how motion works for both slow steady movements and quick movements in image sequences. We believe that each method could be tuned by plotting the standard deviation by tuning parameter, and determine the optimal threshold for each parameter.

In Figure 3, the Temporal Derivative method shows an optimum threshold for the office data set at 0.024 and 0.050 for the red chair data set. In Figure 6, the Simple Filter Temporal Derivative method most likely has an optimum threshold for the office data set at 0.009 and 0.025 for the red chair data set. In Figure 9, the 1D Gaussian Filter Temporal Derivative method is most likely tuned at sigma 3 and Threshold 0.025 for both Office and Red Chair data sets. In Figure 12, Office and Red Chair data sets can be tuned at 5 filter size. In Figure 15, there is really no tuning needed since the picture can be all noise or no noise at all.

If we had to pick a method for basic motion detection, we would ultimately choose the box filtering with temporal derivative method. We believe that this method controls noise to illuminate the motion between image sequences a lot better than the other methods. Although Gaussian 1D has similar results, we felt that the images were a little more noisy than the box filter samples. Additionally, the detail of motion is more easier to control with box filtering, which we believe is an added feature. However, the 1D Gaussian should not be underestimated and does deserve more testing with different data sets.

Appendix

```
from PIL import Image
from PIL.Image import fromarray
from os.path import join, exists
from os import listdir, makedirs
import numpy as np
from numpy import zeros, abs, uint8, array, linalg
from scipy import signal
import time  # To see time
import pandas as pd

# Globals

image_dir = [r'.\Office', '.\RedChair']  # Input Folder of images
num_of_frames = [100, 352]
alpha = 0.3
noise_output = {1: list(), 2: list(), 3: list(), 4: list(), 5: list()}


out_dir = r'.\color_output'
temporal_deriv_out = r'.\temporal_deriv_out'
gaussian1d_out = r'.\1d_gaussian_out'  # Output directory for 1D Gaussian
gaussian2d_out = r'.\2d_gaussian_out'  # Output directory for 2D Gaussian
box_out = r'.\boxFilter_out'  # Output directory for Box Filter
tune_out = r'.\tune_out'


def norm(pic):
    return pic/255


def get_next_image(i):
    i = i + 1
    images = listdir(image_dir[folder])
    img = Image.open(join(image_dir[folder], images[i]))
    return i, img


""" #Manually convolve but it is too slow to implement, opted to use signal.convolve
def imfilter(img, filter):
    result = np.zeros_like(img)
    for i in range(1, img.shape[0] - 1):
        for j in range(img.shape[1] - len(filter) + 1):
            window = img[i-1:i+2, j:j+len(filter)]
            result[i, j + 1] = np.sum(window * filter)  #Iterates the operator window over every pixel
    return result
"""


def motion_detect(i, method, threshold, sigma=None, boxSize=None):
    i, prev_frame = get_next_image(i)
    prev_frame = np.array(prev_frame.convert("L"))
```

```python
    for frame in range(1, num_of_frames[folder]):
        i, curr_frame = get_next_image(i)  # Updating the current frame to the next
        gray_frame = np.array(curr_frame.convert("L"))  # Convert current frame into grayscale

        if method == 1:
            deriv = abs(norm(gray_frame) - norm(prev_frame))
            noise_output[method].append([threshold,
                                         deriv[deriv > threshold].std()])
            output = out_dir  # Output directory for Differential Operator

        elif method == 2:
            deriv = temporal_derivative(gray_frame, prev_frame)
            noise_output[method].append([threshold,
                                         deriv[deriv > threshold].std()])
            output = temporal_deriv_out  # Output directory for Temporal Derive 0.5 * [-1 0 1]

        elif method == 3:
            dimension = 1
            deriv = gaussianfilter(gray_frame,
                                   prev_frame,
                                   dimension,
                                   sigma)
            noise_output[method].append([sigma,
                                         threshold,
                                         deriv[deriv > threshold].std()])
            output = gaussian1d_out  # Output directory for 1D Gaussian Filter

        elif method == 4:
            dimension = 2
            deriv = gaussianfilter(gray_frame,
                                   prev_frame,
                                   dimension,
                                   sigma,
                                   boxSize)
            noise_output[method].append([boxSize,
                                         sigma,
                                         threshold,
                                         deriv[deriv > threshold].std()])
            output = gaussian2d_out  # Output directory for 2D Gaussian Filter

        elif method == 5:
            deriv = boxFilter(gray_frame, prev_frame, boxSize)
            noise_output[method].append([boxSize,
                                         threshold,
                                         deriv[deriv > threshold].std()])
            output = box_out  # Output directory for Box Filter

        else:  # Catch case
            print('Please select a valid method (1-5)')
            return

        mask = zeros(deriv.shape, dtype=uint8)  # Initializing the mask
        mask[deriv > threshold] = 1  # If the delta is significant (threshold) change that pixel value
        color_mask = zeros((curr_frame.height,
                            curr_frame.width, 3),
                           dtype=uint8)  # Initializing a color mask
        color_mask[:, :, 1] = mask[0:color_mask.shape[0],
                              0:color_mask.shape[1]] * 255  # Changing the R G B value to 255 to highli

        comp_image = Image.blend(curr_frame,
                                 fromarray(color_mask),
                                 alpha=alpha)  # Concatenate the mask with the original image

        if folder == 0:
            folder_dir = join(tune_out, 'office')
        else:
            folder_dir = join(tune_out, 'redchair')

        method_dir = join(folder_dir, '%s_method' % method)
        boxsize_dir = join(method_dir, '%s_box_size' % boxSize)
        sigma_dir = join(boxsize_dir, '%s_sigma' % sigma)

        if not exists(method_dir):
            makedirs(method_dir)

        if not exists(boxsize_dir):
            makedirs(boxsize_dir)

        if not exists(sigma_dir):
            makedirs(sigma_dir)

        comp_image.save(
            join(sigma_dir,
                 '{}_threshold (out01_{}).png'.format(threshold,
                                                      str(i).zfill(4))))  # Save and output image
        prev_frame = gray_frame
        return
    return


def temporal_derivative(gray_frame, prev_frame):
    T_deriv = np.array([-0.5,0,0.5]).reshape(-1,1)  # Temporal Derivative 0.5* [-1 0 1] filter
    deriv = signal.convolve(abs(norm(gray_frame)- norm(prev_frame)),
                            T_deriv, mode='same')
    return deriv


def gaussianfilter(gray_frame, prev_frame, dimension, sigma, boxSize = None):
    size = 3 * sigma
    if dimension == 1:
        x = np.arange(-size//2+1, size//2 + 1)  # Create a 1D mask
        gaussian1d = np.exp(-(x ** 2/(2 * (sigma ** 2))))  # 2D Gaussian Equation from the slides
        gaussian1d = gaussian1d * (-x / sigma ** 2)
        gaussian1d /= np.sum(gaussian1d)
        deriv = signal.convolve(abs(norm(gray_frame) - norm(prev_frame)),
                                gaussian1d.reshape(-1,1), mode='same')
    elif dimension == 2:
        center = boxSize // 2
        X,Y = np.mgrid[-center:center+1, -center:center+1]  # Create 2-D Gaussian Mask to smooth
        gaussian2d = np.exp(-(X ** 2 + Y ** 2) / (2 * sigma ** 2))  # 2D Gaussian Equation from the sli
        gaussian2d = gaussian2d * ((X * Y) / sigma ** 4)
        gaussian2d /= np.sum(gaussian2d)

        padded_image= np.pad(
            gray_frame,
            boxSize//2,
            mode ='edge')  # Padding of the gray_frame to fit the gaussian smoothing mask of boxSize x
        padded_previmage= np.pad(prev_frame,
                                 boxSize//2,
                                 mode ='edge')  # Padding of the prev_frame
        smoothed_image = signal.convolve2d(padded_image,
                                           gaussian2d,
                                           mode='valid')  # Create a new smoothed gray_frame as smoothe
        smoothed_prev_frame = signal.convolve2d(padded_previmage,
                                                gaussian2d,
                                                mode='valid')  # Create a new smoothed_prev_frame as sm

        return temporal_derivative(smoothed_image,smoothed_prev_frame)

    else:
        return
    return deriv


def boxFilter(gray_frame, prev_frame, boxSize):
    boxFilt = (1/(boxSize * boxSize)) * np.ones((boxSize,boxSize))  # Zero bias Mean Box filter
    #print(boxFilt)

    padded_image= np.pad(
        gray_frame,
        boxSize//2,
        mode ='edge')  # Padding of the gray_frame to fit the gaussian smoothing mask of boxSize x boxS
    padded_previmage= np.pad(
        prev_frame,
        boxSize//2,
        mode ='edge')  # Padding of the prev_frame
    smoothed_image = signal.convolve2d(
        padded_image,
        boxFilt,
        mode='valid')  # Create a new smoothed gray_frame as smoothed_image
    smoothed_prev_frame = signal.convolve2d(
        padded_previmage,
        boxFilt,
        mode='valid')  # Create a new smoothed_prev_frame as smoothed_image

    return temporal_derivative(smoothed_image,
                               smoothed_prev_frame)  # Send the smoothed images to find the deltas

def switchFolder():
    global folder

    folder += 1
    return


def record_results(filename: str, results: dict, columns: list):
    with pd.ExcelWriter(filename) as writer:
        for i, tab in enumerate(list(results.keys())):
            df = pd.DataFrame(results[tab], columns=columns[i])
            df.to_excel(writer, sheet_name=str(tab), index=False)


if __name__ == '__main__':
    filenames = ['office.xlsx', 'red_chair.xlsx']
    column_names = [['Threshold', 'STD'],
                    ['Threshold', 'STD'],
                    ['Sigma', 'Threshold', 'STD'],
                    ['Box_Size', 'Sigma', 'Threshold', 'STD'],
                    ['Box_Size', 'Threshold', 'STD']]
    folder = 0
    i = 58
    start_time = time.time()

    for f in range(0, 2):
        for m in range(1, 6):
            if m != 4:
                k_max = 101
            else:
                k_max = 31
            if m in (3, 4):
                l_max = 21
            else:
                l_max = 2

            if m in (4, 5):
                j_max = 21
            else:
                j_max = 2

            for j in range(1, j_max):
                for l in range(1, l_max):
                    for k in range(1, k_max):
                        if j % 2 == 1 or m not in (4, 5):
                            print('folder:', f,
                                  ', method:', m,
                                  ', threshold:', k / 1000,
                                  ", sigma:", l,
                                  ", box size:", j)
                            motion_detect(i, method=m,
                                          threshold=k / 1000,
                                          sigma=l,
                                          boxSize=j)
        record_results(filenames[f], noise_output, column_names)
        switchFolder()
        noise_output = {1: list(), 2: list(), 3: list(), 4: list(), 5: list()}

    print('Program finished in %s seconds' % (time.time() - start_time))
```