Homework Assignment 5
Computer Vision for HCI
Prof. Jim Davis
TA: Sayan Mandal

Anirudh Ganesh
CSE5524 (Au '18)
Score: ___/11
Due Date: 09/25/18

0.0.1 Imports

```
In [1]: from skimage.io import imread
        from skimage.filters import gaussian
        import numpy as np
        from matplotlib import pyplot as plt
        from skimage import img_as_float
        import math
```

1 Find best image differencing parameters.

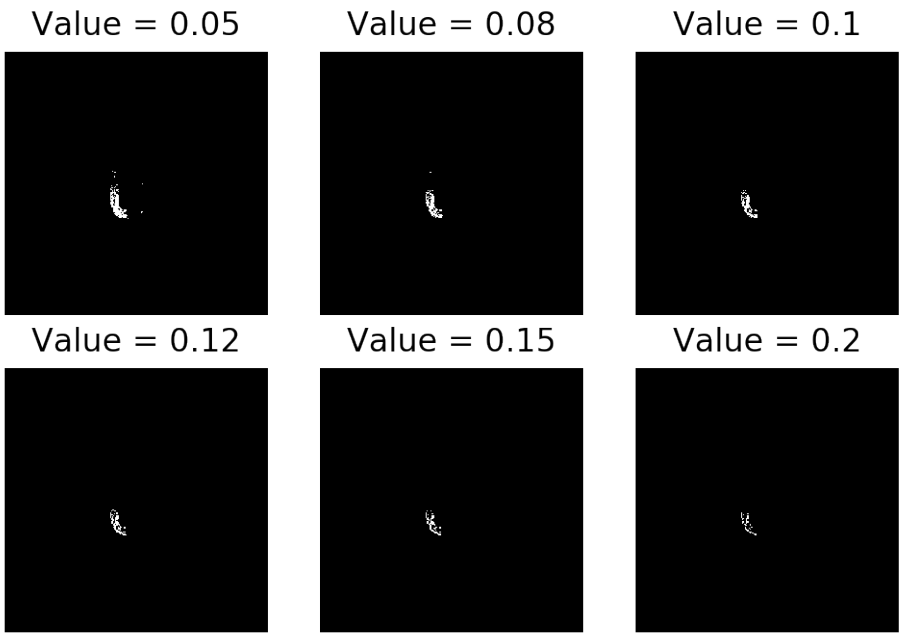
```
In [2]: im1 = imread('./data/aerobic-001.bmp')
        im1 = img_as_float(im1)
        im2 = imread('./data/aerobic-002.bmp')
        im2 = img_as_float(im2)
```

1.1 Simple Image Difference

For sake of simplicity of implementation, we shall use the naive differencing function, as given as below instead of using Weber's difference.

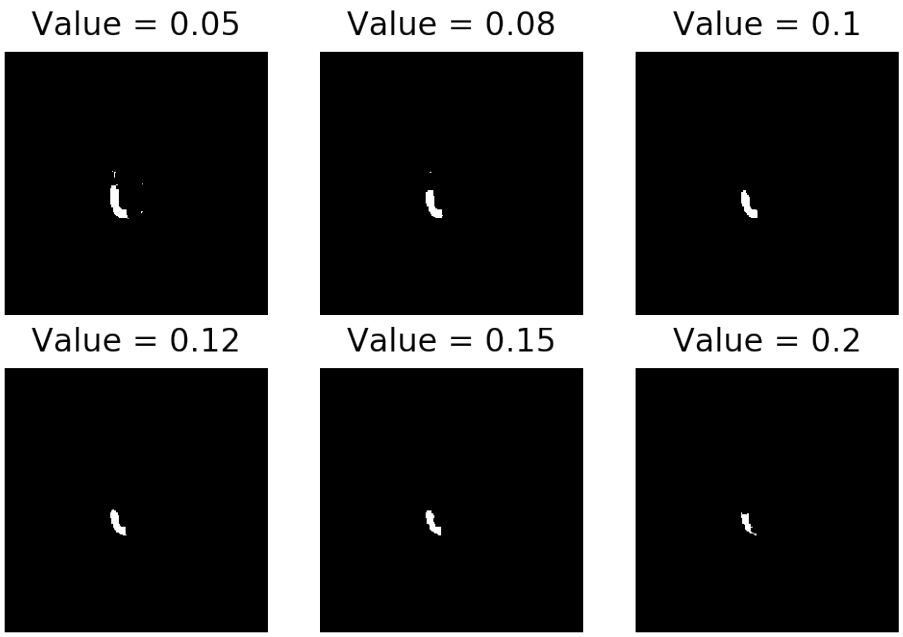
$\Delta I = 1$ if $|I_t - I_{t-1}| > \tau$

```
In [3]: threshLevels = [0.05, 0.08, 0.10, 0.12, 0.15, 0.2]
        f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', dpi=200)
        for thresh in threshLevels:
            magIm = np.abs(im1 - im2)
            tImg = magIm > thresh
            idx = threshLevels.index(thresh)
            #print((int(idx/3), idx%3))
            axarr[int(idx/3), idx%3].axis('off')
            axarr[int(idx/3), idx%3].set_title(f'Value = {thresh}')
            axarr[int(idx/3), idx%3].imshow(tImg, cmap = 'gray', aspect='auto')
```



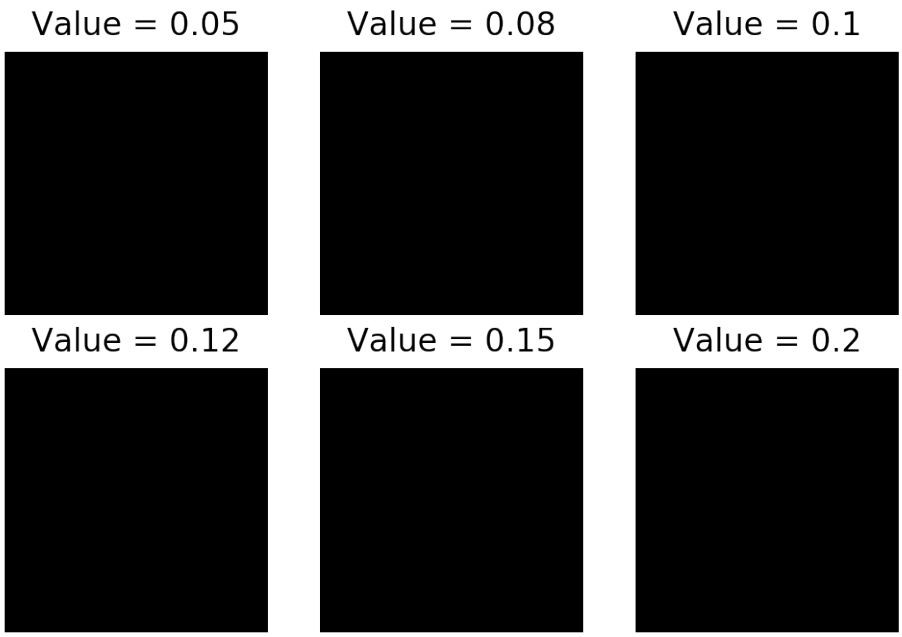
Notice that by default, the thresholding generates a fractured image, which is not ideal for motion analysis. This can be fixed by using morphological operators as discussed below.

```
In [4]: from skimage.morphology import closing
        from skimage.morphology import square

        threshLevels = [0.05, 0.08, 0.10, 0.12, 0.15, 0.2]
        f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', dpi=200)
        for thresh in threshLevels:
            magIm = np.abs(im1 - im2)
            magIm = closing(magIm, square(5))
            tImg = magIm > thresh
            idx = threshLevels.index(thresh)
            #print((int(idx/3), idx%3))
            axarr[int(idx/3), idx%3].axis('off')
            axarr[int(idx/3), idx%3].set_title(f'Value = {thresh}')
            axarr[int(idx/3), idx%3].imshow(tImg, cmap = 'gray', aspect='auto')
```



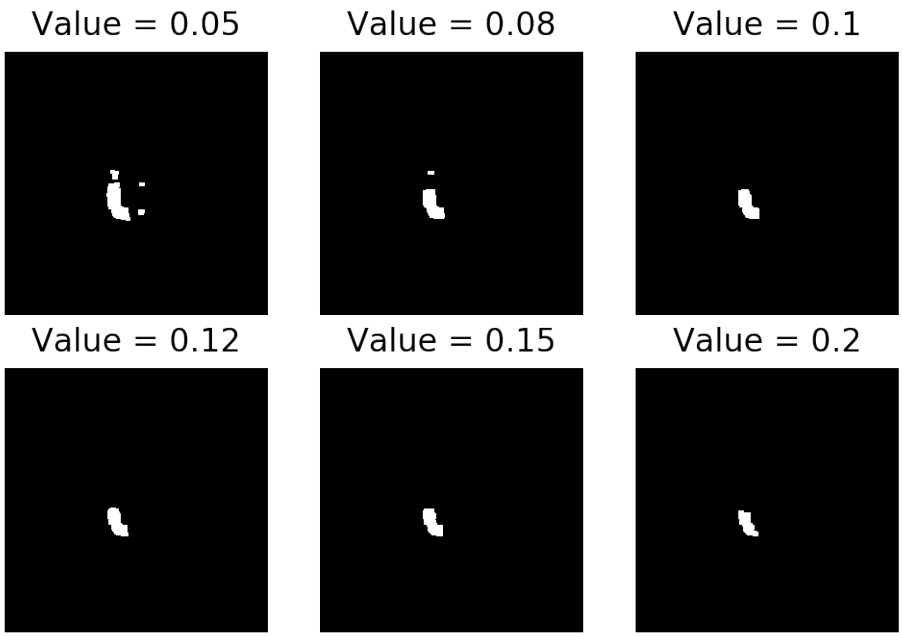
The results upon using closing, I found generates the best result. But I have also included other operations for sake of comparision.

```
In [5]: from skimage.morphology import opening
        threshLevels = [0.05, 0.08, 0.10, 0.12, 0.15, 0.2]
        f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', dpi=200)
        for thresh in threshLevels:
            magIm = np.abs(im1 - im2)
            magIm = opening(magIm, square(5))
            tImg = magIm > thresh
            idx = threshLevels.index(thresh)
            #print((int(idx/3), idx%3))
            axarr[int(idx/3), idx%3].axis('off')
            axarr[int(idx/3), idx%3].set_title(f'Value = {thresh}')
            axarr[int(idx/3), idx%3].imshow(tImg, cmap = 'gray', aspect='auto')
```



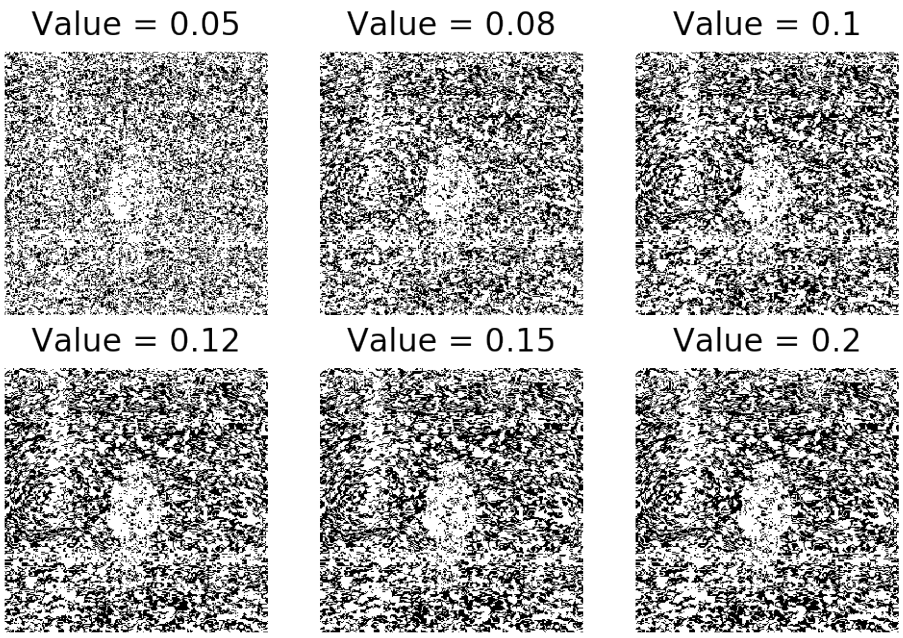
Opening fails because it erodes first causing our fractured image to vanish.

```
In [6]: from skimage.morphology import dilation
        threshLevels = [0.05, 0.08, 0.10, 0.12, 0.15, 0.2]
        f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', dpi=200)
        for thresh in threshLevels:
            magIm = np.abs(im1 - im2)
            magIm = dilation(magIm, square(5))
            tImg = magIm > thresh
            idx = threshLevels.index(thresh)
            #print((int(idx/3), idx%3))
            axarr[int(idx/3), idx%3].axis('off')
            axarr[int(idx/3), idx%3].set_title(f'Value = {thresh}')
            axarr[int(idx/3), idx%3].imshow(tImg, cmap = 'gray', aspect='auto')
```



While dilation generates a good result, the chunkyness could lead to false positives for motion detectors

```
In [7]: from skimage.filters.rank import median

        threshLevels = [0.05, 0.08, 0.10, 0.12, 0.15, 0.2]
        f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', dpi=200)
        for thresh in threshLevels:
            im1 = median(im1)
            im2 = median(im2)
            magIm = np.abs(im1 - im2)
            tImg = magIm > thresh
            idx = threshLevels.index(thresh)
            #print((int(idx/3), idx%3))
            axarr[int(idx/3), idx%3].axis('off')
            axarr[int(idx/3), idx%3].set_title(f'Value = {thresh}')
            axarr[int(idx/3), idx%3].imshow(tImg, cmap = 'gray', aspect='auto')

/usr/lib/python3.7/site-packages/skimage/util/dtype.py:130: UserWarning: Possible precision loss
  .format(dtypeobj_in, dtypeobj_out))
```

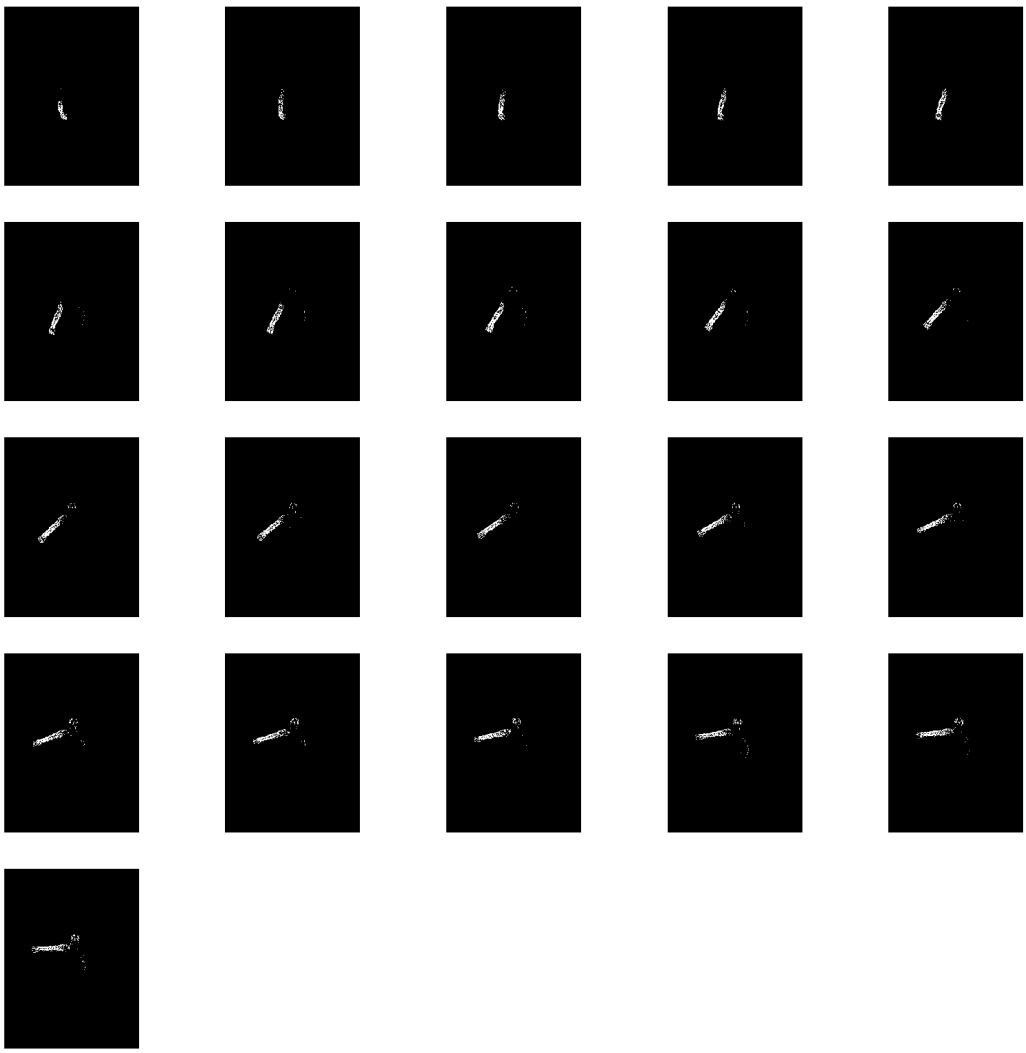


I'm not sure if this is an implementation fault, or the result of sharpening an image with a lot of information, whatever be the case, the image generated cannot be used for upcoming exercise.

2 Perform Motion History Imaging and Motion Energy Imaging using the best differencing technique

```
In [8]: aer_cube = []

        for i in range(1,23):
            tmp = imread('./data/aerobic-{:03}.bmp'.format(i))
            tmp = img_as_float(tmp)
            aer_cube.append(tmp)

        aer_cube=np.array(aer_cube)
```

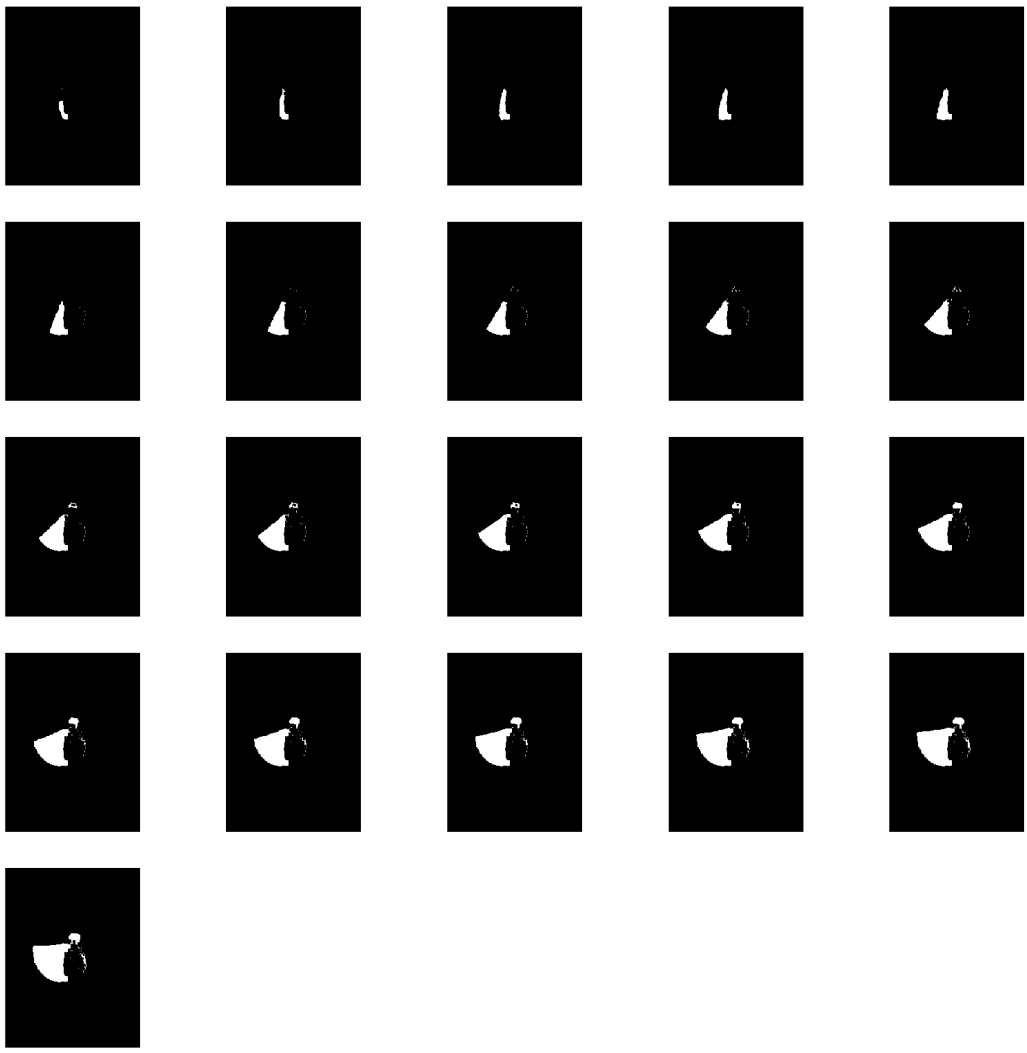
```
In [9]: f, axarr = plt.subplots(5, 5, sharex='col', sharey='row', dpi=200, figsize=(10,10))
        for idx in range(0, 21):
            axarr[int(idx/5), idx%5].axis('off')
            axarr[int(idx/5), idx%5].imshow(np.abs(aer_cube[idx]-aer_cube[idx+1]) > 0.08, cmap =

        for idx in range(21, 25):
            axarr[idx//5, idx%5].axis('off')
```



Just a plot of consecutive image differences.

```
In [10]: aer_cube.shape

         aer_cube[0].max()

         T = aer_cube.shape[0]

In [11]: aer_diff = np.abs(aer_cube[1:,:,:] - aer_cube[:-1,:,:])

         for i in range(aer_diff.shape[0]):
             aer_diff[i] = closing(aer_diff[i], square(5)) > 0.08

In [12]: MEI = [aer_diff[:i, :, :].max(axis=0) for i in range(1, T)]
```

```
In [13]: f, axarr = plt.subplots(5, 5, sharex='col', sharey='row', dpi=200, figsize=(10,10))
         for idx in range(0, 21):
             axarr[int(idx/5), idx%5].axis('off')
             axarr[int(idx/5), idx%5].imshow(MEI[idx], cmap = 'gray')

         for idx in range(21, 25):
             axarr[idx//5, idx%5].axis('off')
```

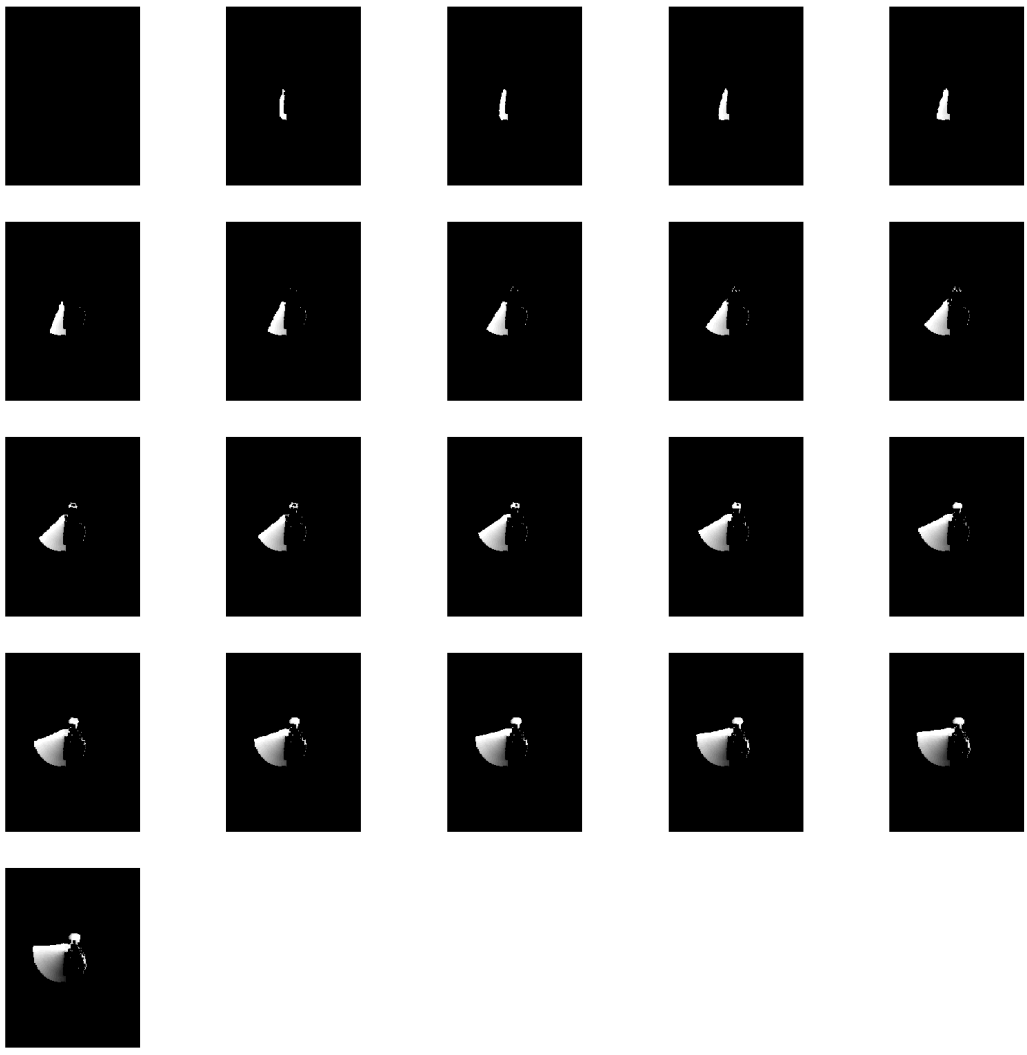


We can see that MEI imaging produces a chunky representation of the motion

```
In [14]: MHI = np.zeros(aer_cube.shape)

         for idx in range(1, T-1):
             for i in range(0, aer_cube.shape[1]):
                 for j in range(0, aer_cube.shape[2]):
```

```
                    if aer_diff[idx][i][j] == 1:
                        MHI[idx][i][j] = T
                    else:
                        MHI[idx][i][j] = max(0, MHI[idx-1][i][j] - 1)
```

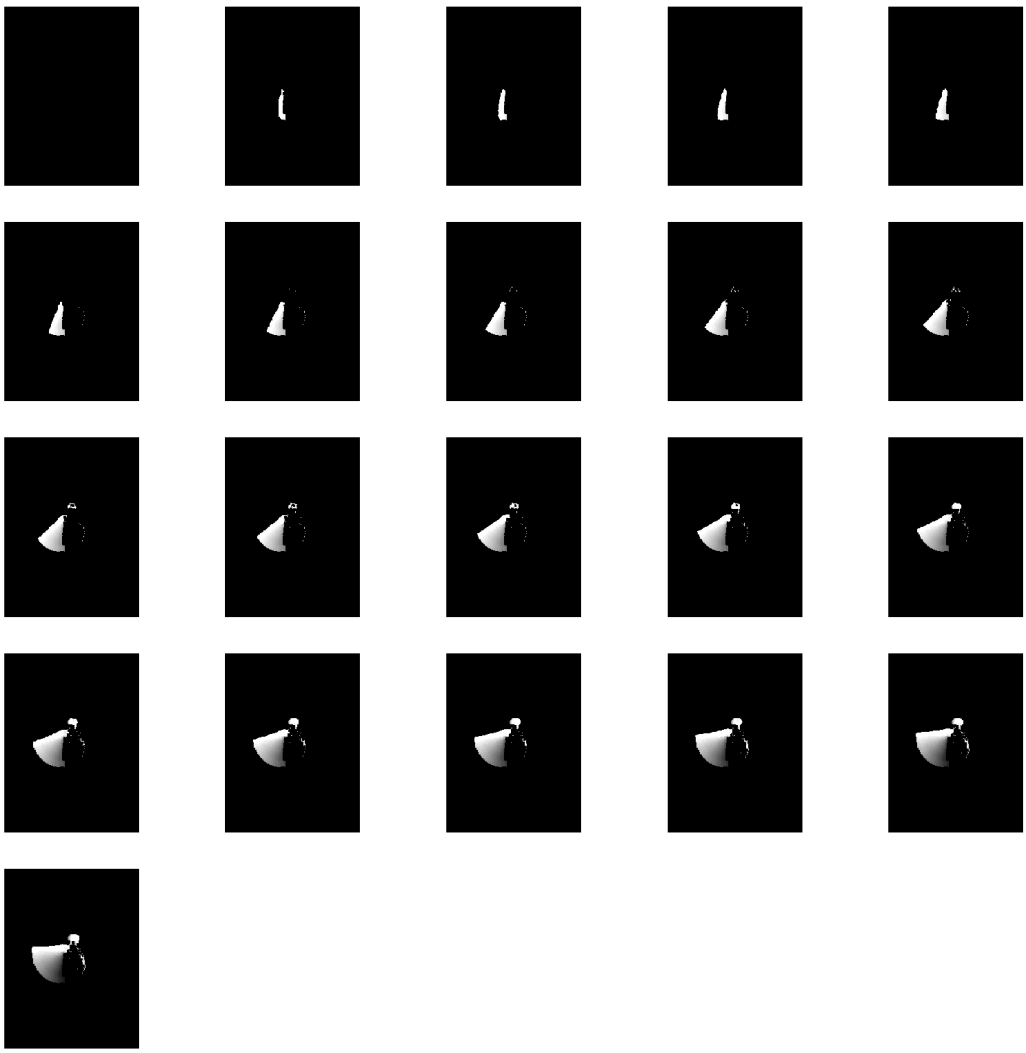
```
In [15]: f, axarr = plt.subplots(5, 5, sharex='col', sharey='row', dpi=200, figsize=(10,10))
         for idx in range(0, 21):
             axarr[int(idx/5), idx%5].axis('off')
             axarr[int(idx/5), idx%5].imshow(MHI[idx], cmap = 'gray')

         for idx in range(21, 25):
             axarr[idx//5, idx%5].axis('off')
```



The MHI imaging we can see produces a smoother texture which better captures the direction of the movement. MEI would result in the same image if the hand was being raised or lowered

while MHI would generate opposite textures, thus making it more useful.

```
In [16]: delta_T = (T-15)/(8)

         MHI_deltaT = np.zeros(MHI.shape)

         for idx in range(1, T-1):
             for i in range(0, aer_cube.shape[1]):
                 for j in range(0, aer_cube.shape[2]):
                     if MHI[idx][i][j] - delta_T > 0:
                         MHI_deltaT[idx][i][j] = MHI[idx][i][j] - delta_T

         f, axarr = plt.subplots(5, 5, sharex='col', sharey='row', dpi=200, figsize=(10,10))
         for idx in range(0, 21):
             axarr[int(idx/5), idx%5].axis('off')
             axarr[int(idx/5), idx%5].imshow(MHI_deltaT[idx], cmap = 'gray')

         for idx in range(21, 25):
             axarr[idx//5, idx%5].axis('off')
```



This is the implementation of MHI but with $\tau - \Delta\tau$

2.1 Similtude Moments for the last MHI and MEI

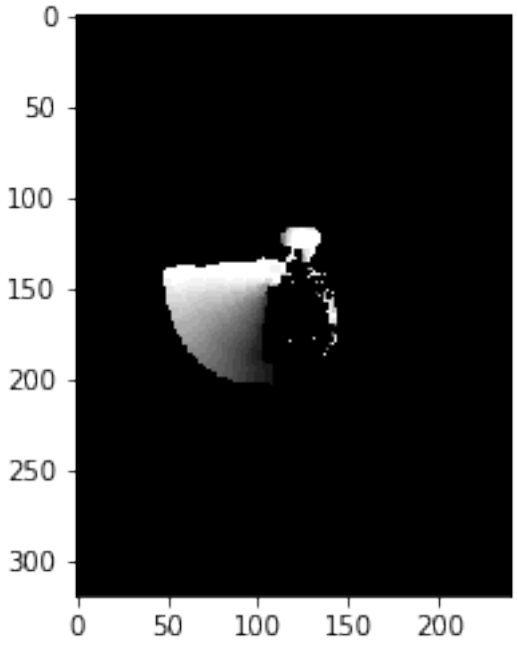
```
In [17]: from numpy import mgrid, sum


         def similitudeMoments(image):

             ### Make sure the image is a grayscale image
             assert len(image.shape) == 2

             ## Temp grid for storing intermediate operations
             x, y = mgrid[:image.shape[0],:image.shape[1]]
```

```
## Our final dictionary that contains the
moments = {}


moments['mean_x'] = sum(x*image)/sum(image)
moments['mean_y'] = sum(y*image)/sum(image)

## Spatial moments: Spatial moments  often used to describe region shape

# Zeroth Order
moments['m00'] = sum(image)

# First Order
moments['m01'] = sum(x*image)
moments['m10'] = sum(y*image)

# Second Order
moments['m11'] = sum(y*x*image)
moments['m02'] = sum(x**2*image)
moments['m20'] = sum(y**2*image)

# Third Order
moments['m12'] = sum(x*y**2*image)
moments['m21'] = sum(x**2*y*image)
moments['m03'] = sum(x**3*image)
moments['m30'] = sum(y**3*image)

## Central moments: Translation Invariant

# First Order (Seem useless, in terms of calculating the final nu moments. But stil
moments['mu01']= sum((y-moments['mean_y'])*image)
moments['mu10']= sum((x-moments['mean_x'])*image)

# Second Order (Moment Ellipse Orientation)
moments['mu11'] = sum((x-moments['mean_x'])*(y-moments['mean_y'])*image)
moments['mu02'] = sum((y-moments['mean_y'])**2*image)
moments['mu20'] = sum((x-moments['mean_x'])**2*image)

# Third Order (Skewness of the Image)
moments['mu12'] = sum((x-moments['mean_x'])*(y-moments['mean_y'])**2*image)
moments['mu21'] = sum((x-moments['mean_x'])**2*(y-moments['mean_y'])*image)
moments['mu03'] = sum((y-moments['mean_y'])**3*image)
moments['mu30'] = sum((x-moments['mean_x'])**3*image)

# Similitude Moments: Invariant to translation and scale

moments['eta11'] = moments['mu11'] / sum(image)**(2/2+1)
```

```
            moments['eta12'] = moments['mu12'] / sum(image)**(3/2+1)
            moments['eta21'] = moments['mu21'] / sum(image)**(3/2+1)
            moments['eta02'] = moments['mu02'] / sum(image)**(2/2+1)
            moments['eta20'] = moments['mu20'] / sum(image)**(2/2+1)
            moments['eta03'] = moments['mu03'] / sum(image)**(3/2+1)
            moments['eta30'] = moments['mu30'] / sum(image)**(3/2+1)

            return moments
```

In [18]:
```
lastMHI = MHI_deltaT[-2]
lastMHI = lastMHI/T

plt.imshow(lastMHI, cmap='gray')
```

Out[18]:
```
<matplotlib.image.AxesImage at 0x7f58b1721a20>
```



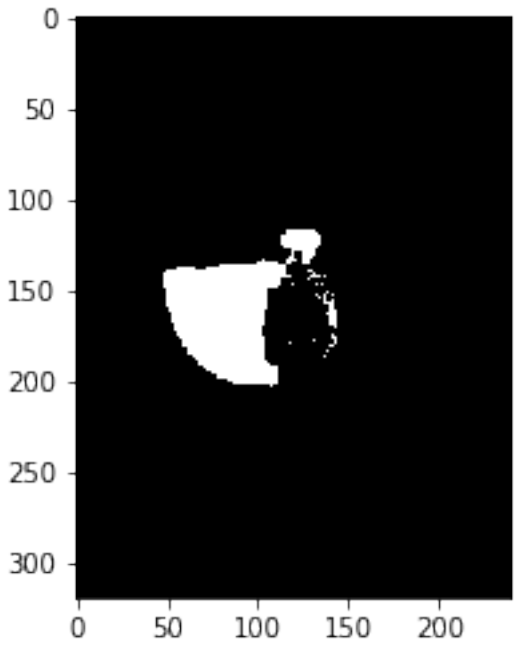
In [19]:
```
lastMHI_moments = similitudeMoments(lastMHI)
print(lastMHI_moments)
```

```
{'mean_x': 154.03915051895743, 'mean_y': 86.05553257989888, 'm00': 2266.8920454545455, 'm01': 34
```

We can see that as a consequence of being smoother textured, the moments reflect this in their means, where the mean is lower than the one for MEI, reflecting the change in perceived center of attention.

```
In [20]: lastMEI = MEI[-1]
         lastMEI_normalized = np.zeros(lastMEI.shape)
         for i in range(lastMEI.shape[0]):
             for j in range(lastMEI.shape[1]):
                 t = (lastMEI[i][j]*(20. - 1.)/21.)
                 if t > 0:
                     lastMEI_normalized[i][j] = t

         plt.imshow(lastMEI_normalized, cmap='gray')
```

```
Out[20]: <matplotlib.image.AxesImage at 0x7f58b1b6c470>
```




```
In [21]: lastMEI_moments = similitudeMoments(lastMEI_normalized)
         print(lastMEI_moments)
```

```
{'mean_x': 161.99532324621734, 'mean_y': 86.74140302613479, 'm00': 3288.809523809524, 'm01': 532
```

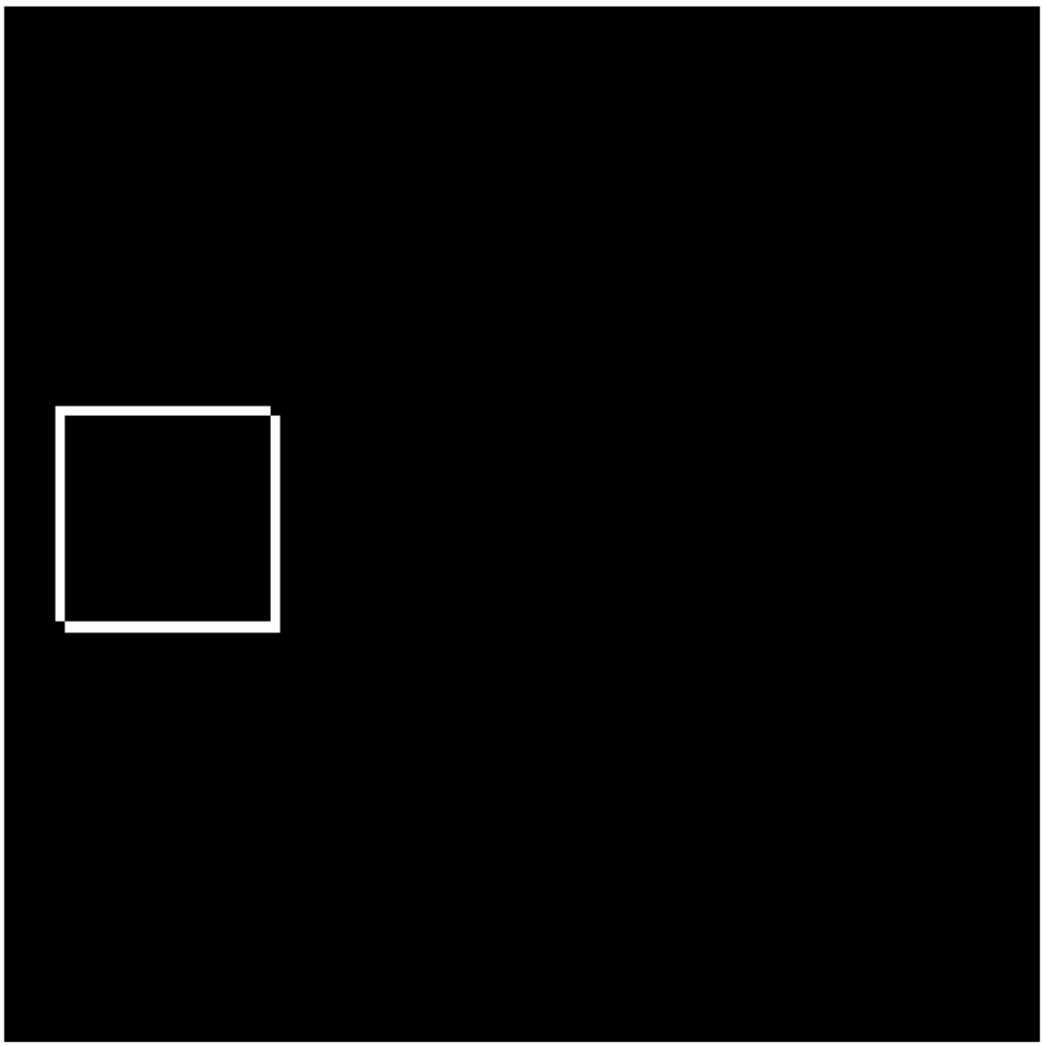
Also notice that other features are largely similar since the shape itself resembles the one generated by MHI.

3 Perform Optic Flow on the set of images using the best differencing technique

```
In [22]: box1 = np.zeros((101,101))
         box2 = np.zeros((101,101))
```

```
size = 21
box1[39:39+size, 5:5+size] = 1
box2[40:40+size, 6:6+size] = 1

box1 = img_as_float(box1)
box2 = img_as_float(box2)

plt.figure(figsize=(10,10))
plt.axis('off')
plt.imshow(np.abs(box2-box1), cmap='gray')
```

Out[22]: <matplotlib.image.AxesImage at 0x7f58b1c43470>



Just a plot of image diff, to show that it is indeed shifted by just 1 pixel down and right.

```
In [23]: from scipy import signal
         def optical_flow(I1g, I2g, window_size):

             kernel_x = 0.25 * np.array([[-1., 1.], [-1., 1.]])
             kernel_y = 0.25 * np.array([[-1., -1.], [1., 1.]])
             kernel_t = 0.25 * np.array([[1., 1.], [1., 1.]])
             kernel_x = np.fliplr(kernel_x)
             mode = 'same'
             fx = (signal.convolve2d(I1g, kernel_x, boundary='symm', mode=mode))
             fy = (signal.convolve2d(I1g, kernel_y, boundary='symm', mode=mode))
             ft = (signal.convolve2d(I2g, kernel_t, boundary='symm', mode=mode) +
                   signal.convolve2d(I1g, -kernel_t, boundary='symm', mode=mode))

             #ft = I2g - I1g

             u = np.zeros(I1g.shape)
             v = np.zeros(I1g.shape)

             window = np.ones((window_size,window_size))

             denom = (signal.convolve2d(fx**2, window, boundary='symm', mode=mode) *
                      signal.convolve2d(fy**2, window, boundary='symm', mode=mode) -
                      signal.convolve2d(fx*fy, window, boundary='symm', mode=mode)**2)
             denom[denom == 0] = 1

             u = ((signal.convolve2d(fy**2, window, boundary='symm', mode=mode) *
                   signal.convolve2d(fy*ft, window, boundary='symm', mode=mode) +
                   signal.convolve2d(fx*fy, window, boundary='symm', mode=mode) *
                   signal.convolve2d(fy*ft, window, boundary='symm', mode=mode) ) /
                 denom)


             v = ((signal.convolve2d(fx*ft, window, boundary='symm', mode=mode) *
                   signal.convolve2d(fx*fy, window, boundary='symm', mode=mode) -
                   signal.convolve2d(fx**2, window, boundary='symm', mode=mode) *
                   signal.convolve2d(fy*ft, window, boundary='symm', mode=mode)) /
                 denom)


             return (u,v)

         u,v = optical_flow(box1, box2, 3)
```

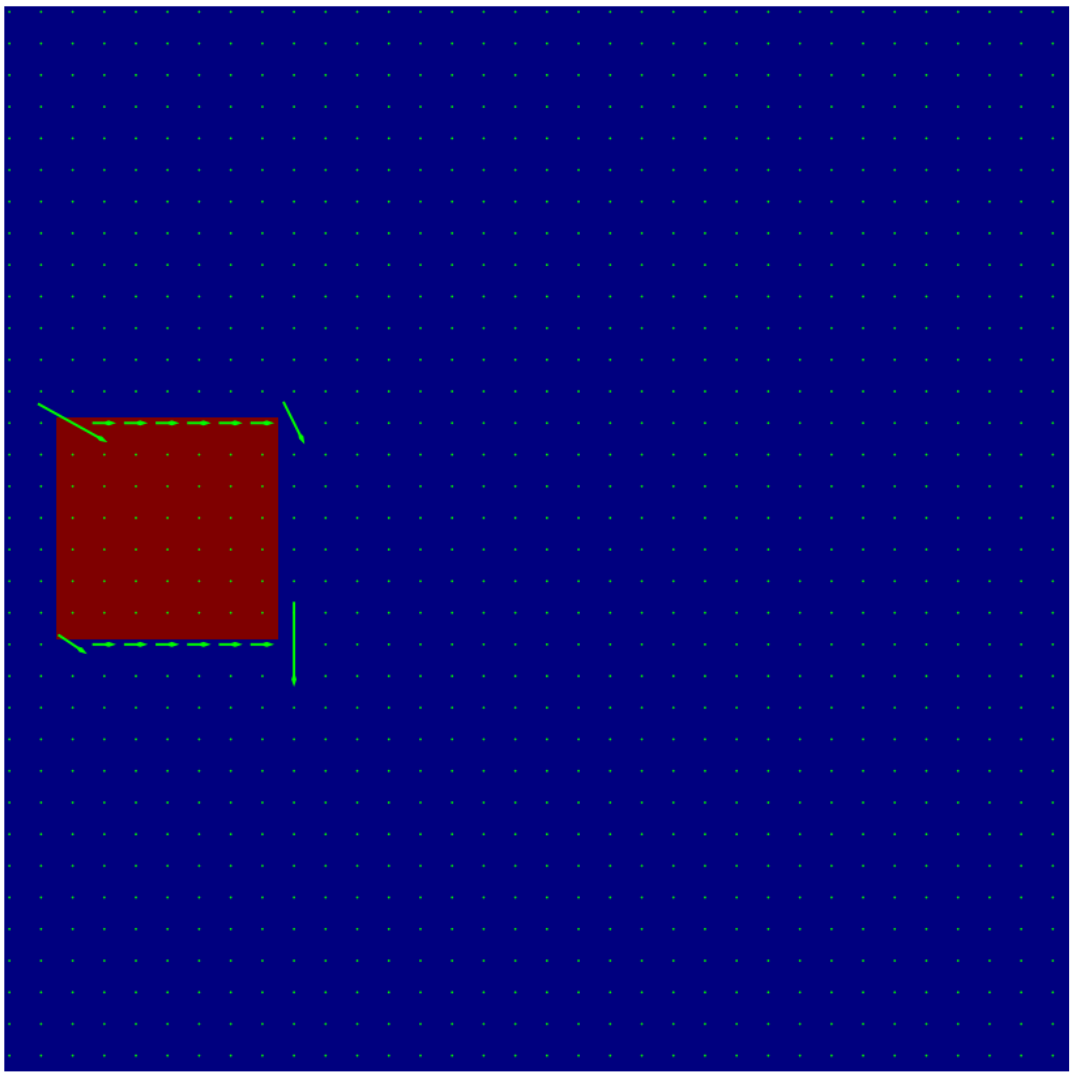
The optical flow calculation function. Notice that I use convolution to calculate the vectors. There might be easier ways to do it, and even ways to do it that's computationally a lot cheaper, but in-order to preserve the scales when plotting and make sure the formula used is a direct translation of the ones present in slides, I decided to go with this approach.

```
In [24]: x = np.arange(0, box1.shape[1], 1)
```

```
        y = np.arange(0, box1.shape[0], 1)
        x, y = np.meshgrid(x, y)
        delta = 3
```

```
In [25]: plt.figure(figsize=(15,15))
         plt.axis('off')
         plt.imshow(box1, cmap='jet')
         plt.quiver(x[::delta, ::delta], y[::delta, ::delta],
                    u[::delta, ::delta], v[::delta, ::delta],
         color='lime', pivot='middle', headwidth=2, headlength=3, scale=25)
```

```
Out[25]: <matplotlib.quiver.Quiver at 0x7f58b1bc0080>
```



Notice that the arrows are pointed in the direction of motion.