(a) [10 points] Set up a static test scene using common household objects on a table top. You can use Fig. 1 above as inspiration. Try to have objects at different depths. Now capture a video of this scene by moving your cellphone camera in a zig-zag motion as shown below in Fig. 2. The more you cover the plane the better. Ensure that all objects are in focus in your video. Avoid tilting or rotating the camera as it will cause trouble in the algorithm we'll implement in the next part of the problem. The number of frames in this video will determine the time required to compute the output. So, make sure the video is not too long! Also make sure to capture this video in a well lit room with no lighting changes or motion.

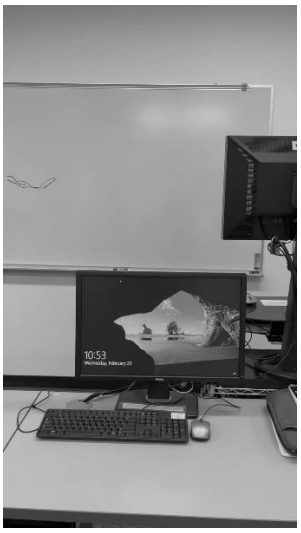
Looking back in hindsight, this part of the assignment was the most influential part of the assignment. Although the fastest to complete, extra movement in the depth of the camera made it so later there would be extra artifacts to appear.

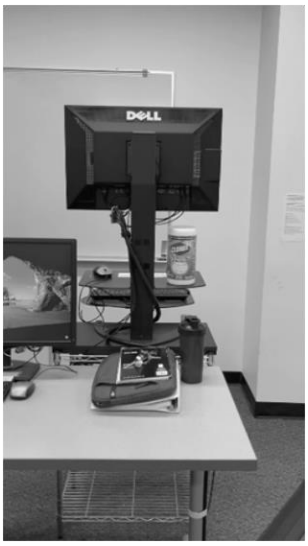
For this part of the assignment, I captured the video using an iphone 11. Since ios videos have a different format, extra processing must be done to convert the video to an mp4 format.

(b) [10 points] Write a script to extract individual frames from this video. Since this video was captured from a cell phone, each frame image is in RGB color. Write a script to convert each frame to gray-scale and save them as a sequence of PNG or JPG files to disk. (There are several ways to do this in Python. For example, here's one possible way to use the imageio library (Links to an external site.).) Display a subset of these images in, say, a grid of rows and columns that follows the camera motion path in Fig. 2. For example, you can choose to show a subset of 9 images in a 3x3 grid that approximately lines up with the motion in the video.

```python
import imageio as iio
import pylab
import cv2

# from documentation in: https://imageio.readthedocs.io/en/stable/

# reads from file
reader = iio.get_reader('/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/001.mp4', 'ffmpeg')

#finds frames in video and appropiate spacing for 9x9 rows
cap = cv2.VideoCapture("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/001.mp4")
length = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
divs = int(length / 8)

#creates array with spaced out frames
nums = []
for i in range(length):
  if i % divs == 0:
    nums.append(i)

#iterate through frames in video
#save those
for num in nums:
    image = reader.get_data(num)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    fig = pylab.figure()
    pylab.imshow(gray)
    cv2.imwrite("/content/drive/MyDrive/CS410-Unconventional_cameras/grayshot_%s.png" % num, gray)
pylab.show()
```

The following figure shows the snippet of code that will produce the grid of 9 images shown below.






(c) [5 points] In the first video frame, select an object that you want to bring into focus. (For example, in Fig. 1, this was the bike helmet.) Crop it out. We will use this as a template that will be searched over all the remaining frames of the video. Display this cropped template in a new figure.

The following figure represents approximately the template that was used as a focus point for the rest of the assignment. One important note is that this image itself was not used as a template, but instead the pixel coordinates that represent this image were used to calculate the template for the rest of the video frames.

(d) [15 points] For each video frame from part (b), find the best matching pixel location for the template from part (c). You can use normalized cross-correlation template matching that we discussed in class. You can write this from scratch or use scikit-image (Links to an external site.). The final result should be an N_{frames} x 2 array of shifts where each row in this array stores the *x* and *y* shifts of the best matching template location for the n^{th} video frame where $1 <= n <= N_{frames}$. Generate a plot of *x* shifts and *y* shifts as a function of frame number *n*. Generate another plot of *x* shift v/s *y* shift. Do these plots agree with you expect from how the video was captured in part (a)?

```
#results for template matching
result1 = match_template(image1, template)
result2 = match_template(image2, template)
result3 = match_template(image3, template)
result4 = match_template(image4, template)
result5 = match_template(image5, template)
result6 = match_template(image6, template)
result7 = match_template(image7, template)
result8 = match_template(image8, template)
result9 = match_template(image9, template)

#creating a list of max values for the template matches
result_coord = []

result_coord.append(np.unravel_index(np.argmax(result1), result1.shape))
result_coord.append(np.unravel_index(np.argmax(result2), result2.shape))
result_coord.append(np.unravel_index(np.argmax(result3), result3.shape))
result_coord.append(np.unravel_index(np.argmax(result4), result4.shape))
result_coord.append(np.unravel_index(np.argmax(result5), result5.shape))
result_coord.append(np.unravel_index(np.argmax(result6), result6.shape))
result_coord.append(np.unravel_index(np.argmax(result7), result7.shape))
result_coord.append(np.unravel_index(np.argmax(result8), result8.shape))
result_coord.append(np.unravel_index(np.argmax(result9), result9.shape))
```

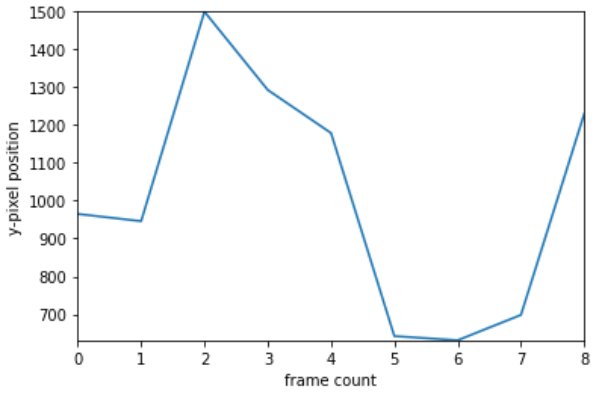
```
#these will be the x-positions
x_pos = []
for x in result_coord:
  x_pos.append(x[0])
#print(x_pos)

plt.figure(1)
plt.margins(x=0,y=0)
plt.plot(x_pos)
plt.ylabel('y-pixel position')
plt.xlabel('frame count')

#these will eb the y-positions
y_pos = []
for y in result_coord:
  y_pos.append(y[1])
#print(y_pos)

plt.figure(2)
plt.margins(x=0,y=0)
plt.plot(y_pos)
plt.ylabel('x-pixel position')
plt.xlabel('frame count')
```

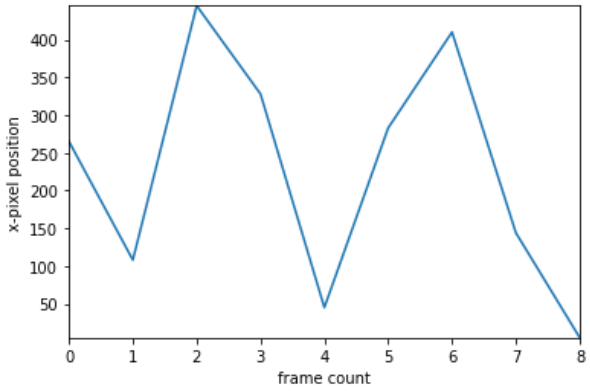
```
#these are the combined positions as a line plot
plt.figure(3)
plt.margins(x=0,y=0)
plt.plot(*zip(*result_coord))
plt.ylabel('x-pixel position')
plt.xlabel('y-pixel position')

#these are the combined positions as a scatter plot
plt.figure(4)
plt.margins(x=0,y=0)
plt.scatter(*zip(*result_coord))
plt.ylabel('x-pixel position')
plt.xlabel('y-pixel position')

plt.figure(5)
plt.imshow(template)
```

The following code snippet shows the code that will produce the plot figures below. The first part of the code imports the necessary files and frames necessary to compute the template matching as well as create the range of pixels that will represent the template. Note that the template is taken from the first frame and will be compared to the rest of the video frames.

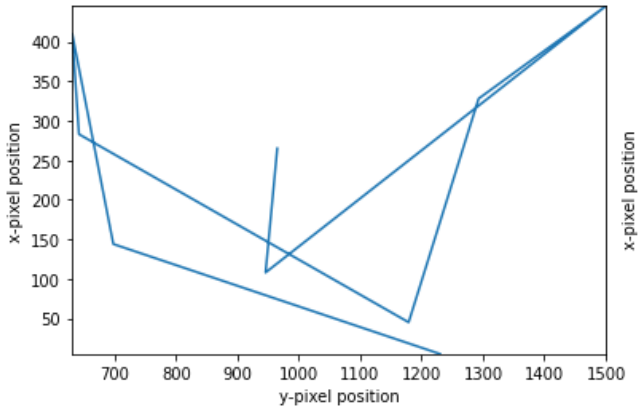
The second part of the code applies the ‘match_template’ function to the images and saves them so their respective maximum coordinate matches can be appended to an array. This array will then contain the best matches from each frame compared to the first images template.

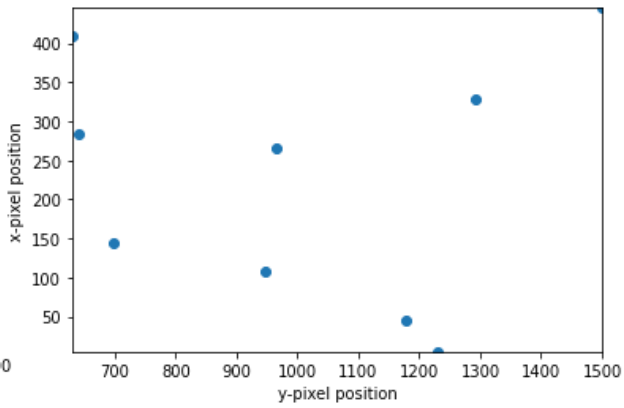
The below figures show both plots of x-positions compared to the number of frames captured, and y-positions compared to the number of frames captured respectively. The plots are labeled as follows.





Below are 2 plots: a line plot, and a scatter plot. Although these plots both represent x-pixel position vs y-pixel position graphs, both are shown to show the key values meant to be displayed.





As we can see from the above plots, the data does match the movement of how the video was captured. Although the y-pixel plot does seem to start and end at a relatively similar positions, I think the normalization of the data seems to be the cause for this. One reason why I think this could be the case is because there are similar patterns from how the plot rises and falls. As we

can see, there are pauses before the y-pixel values rise or fall meaning these values might need better representation.

Looking at the x-positions, the pattern is more noticeable where we can see that the positions do go back and forth. This is a clear sign that we were able to capture the back-and-forth nature of the captured video.

Looking at the y-vs-x position graph, it is much harder to distinguish how the data is represented since the line graph seems to not tell much. I think if we had more data, we could capture the downward lateral movement and the horizontal back and forth movement. This would most likely look like a zig-zag rising diagonal movement since the video was captured moving down.

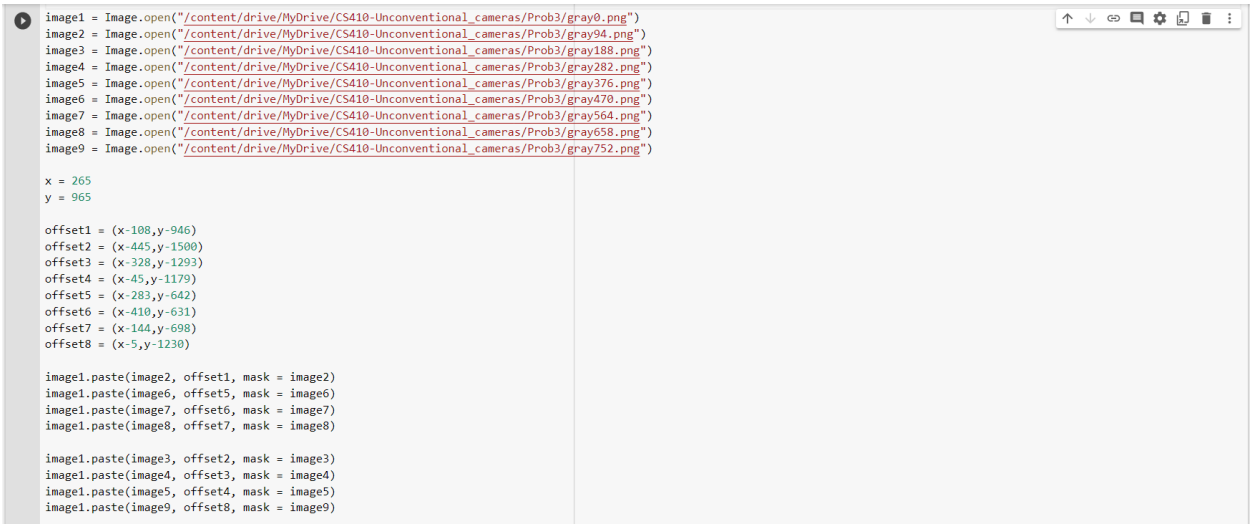
(e) [10 points] Once you have the pixel shifts for each frame, you can synthesize the refocused image by shifting each frame in the opposite direction and then summing up all the frames. Be careful of pixel indexes over/underflowing when applying these shifts. You may need to crop or extend the dimensions of each image frame. Display your final result image with the selected object in focus and background out of focus. If you see any artifacts comment on what may be causing them.

Below two different figures are shown. The leftmost figure shows a selective composite shot which layers half of the capture video frames. This was the best we were able to layer and shift the images. As we can see, the focused image does come out clearer than the rest of the background

To the right we see what a total composite image looks like. This includes all the video frames (about double) of what we were able to shift and layer. As we can see, there are many artifacts that blur and create an unclear composite image.

I think there are different reasons why were not able to create a complete composite image. One of the reasons being that there was a diagonal shift in how the video was captured. This would create a distortion of shape and would cause the 'template_match' function to fail. Another important reason why the complete composite image could have many artifacts are because there is a change in depth from the video. A change in depth could account for a change in shape and another reason for the 'template_match' function to fail.

As we can see from the figure, we can see the same transformations were performed all captured video frames which could explain irregularities in how the video was captured.



```
image1 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray0.png")
image2 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray94.png")
image3 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray188.png")
image4 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray282.png")
image5 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray376.png")
image6 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray470.png")
image7 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray564.png")
image8 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray658.png")
image9 = Image.open("/content/drive/MyDrive/CS410-Unconventional_cameras/Prob3/gray752.png")

x = 265
y = 965

offset1 = (x-108,y-946)
offset2 = (x-445,y-1500)
offset3 = (x-328,y-1293)
offset4 = (x-45,y-1179)
offset5 = (x-283,y-642)
offset6 = (x-410,y-631)
offset7 = (x-144,y-698)
offset8 = (x-5,y-1230)

image1.paste(image2, offset1, mask = image2)
image1.paste(image6, offset5, mask = image6)
image1.paste(image7, offset6, mask = image7)
image1.paste(image8, offset7, mask = image8)

image1.paste(image3, offset2, mask = image3)
image1.paste(image4, offset3, mask = image4)
image1.paste(image5, offset4, mask = image5)
image1.paste(image9, offset8, mask = image9)
```

(f) [5 points] What is the algorithmic complexity of the algorithm in part (d)? Propose some tricks to speed it up. (No need to implement them, just mention some speculative ideas in your write-up).

From the code we used for template matching, we notice that it is relatively low in complexity in terms of big-O notation because there are many repetitive tasks being done by brute force. The most complex parts being the for loops which create and append x-pixel positions to an array and a for loop for the y-pixel positions. Since these loops iterate through the array of best template matching coordinates, we know we iterate through a list only n times. This implies the code is big-O(n).

Because this is brute force however, we notice that there are many instances where objects are created. And, although we might not be able to improve time complexity, we

could reduce the number of objects created, and therefore, reduce spatial complexity. One way we could do this is by creating a single result object and iterate through each video frame, where the result object would hold the value of a single given best fitting template_match coordinate instead of creating a result object for each image.