# SEGMENTATION BY CLASSICAL COMPUTER VISION

ECE 5220: FINAL PROJECT

**Nicholas Rabow**

**Department of Electrical and Computer Engineering**

**Weber State University**

Fall 2021

## Contents

## ABSTRACT

Classical computer vision is used to describe the traditional ways that image processing has been done. Most of the problems encountered today are typically solved using some form of deep learning, however classical computer vision is not completely obsolete. "[Deep Learning] is sometimes overkill as often traditional [Computer Vision] techniques can solve a problem much more efficiently and in fewer lines of code than [Deep Learning]. (3) This paper will explore the use of classical computer vision in order to analyze the results of three different experiments. While some parts are not completely optimized for speed/accuracy, much can be learned from the techniques implemented.

# 1 Introduction

Three experiments were conducted and the results were given to me in order to provide an automated method of analysis. This paper will cover the methods used as well as an analysis of the effectiveness of these methods. It will also explore the nuances of the problems and how solutions were molded to solve them. My solutions are not perfect, however, I hope that in conjunction with other students' work, a preferred solution can be found. [1]

# 2 Splash Detection

The Department of Mechanical Engineering at Weber State University conducted an experiment. The experiment results are shown in Fig 1. The object of the analysis is to provide the percentage area of the paper covered in the dye. The pipeline for this analysis consists of three mains parts: pre-processing,processing, and analysis. They can be further broken down as:

1. Rotation
2. Crop
3. Contrast Enhancement
4. Thresholding
5. Calculations

## 2.1 Preprocessing

Rotation and Cropping are part of this section of the pipeline. I aimed to automate this code as much as I could. The rotation function is not automated, but rather the parameters were discovered by a guess-and-check method. The image was rotated by 1°counter-clockwise. This amount of rotation appears to be correct. The next step in the pre-processing chain was cropping. The cropping method is a two-part method: rough cropping and background removal. The rough cropping method finds the edges of the paper automatically. In order to improve the performance of this algorithm. The image is converted to gray-scale and a Gaussian filtered in applied before using Otsu's threshold to create a binary image. A search of random rows/columns is applied to the altered image to find the suspected edges and add them to a set of respective arrays. If the standard deviation of any of the arrays is greater than 100 pixels then the process is repeated. [2] The image is cropped according to the mean of the edge values found and a buffer of 100 pixels is added. [3] Fig 1 shows the result after this step. The background is removed as the last step of the pre-processing stage. The highest contrast channel will provide the best threshold to create a mask for the paper. By checking each channel for the largest standard deviation, it is decided that the blue channel has the most contrast. Using Otsu's threshold a basic mask is created. There are still some artifacts left over in the image. Morphological dilation with a 3x3 array of 1's cleans up the edges and takes care of the artifacts. The mask is then applied to the rough cropped image. The background is replaced with the average intensity of the image. The image is now ready for processing. 1

## 2.2 Processing

The processing stage consists of contrast enhancement and thresholding. The same process as before to find the highest contrast channel is performed. However, the result in different this time with the red channel having the most contrast. The next step is to change the exposure of the image. This is done by calculating a low/high percentile number. The parameters I found to work best for this case was a low of 3 and a high of 30. The values returned from the function are then used to rescale the intensity of the image, this appears as

[1] I aim for speed optimization in my solution

[2] 100 pixels was chosen because of the keystone effect that exists on the image. If the first row and last row were randomly chosen the difference would be this number. Anything greater proves the edges of the paper were not correctly found.

[3] The reason is the same for the 100 pixels as explained in footnote 2.

an increase in exposure. After increasing the contrast, Otsu's thresholding can be used more effectively. The result is not perfect and has some noise in the corners as see in Fig. 1.

### 2.3 Results

During the pre-processing stage, the area of the paper was found by counting the non-zero components of the mask. It was stored in a variable called, paper area. After the final thresholding, the area of the dye was found by counting the non-zero components. It was stored in a variable called cover area. From these numbers it was calculated that 8.96% of the paper is covered by the dye. There is some error associated with that number as can be seen in Fig. 1. Note that this implementation took approximately 10 seconds to execute.

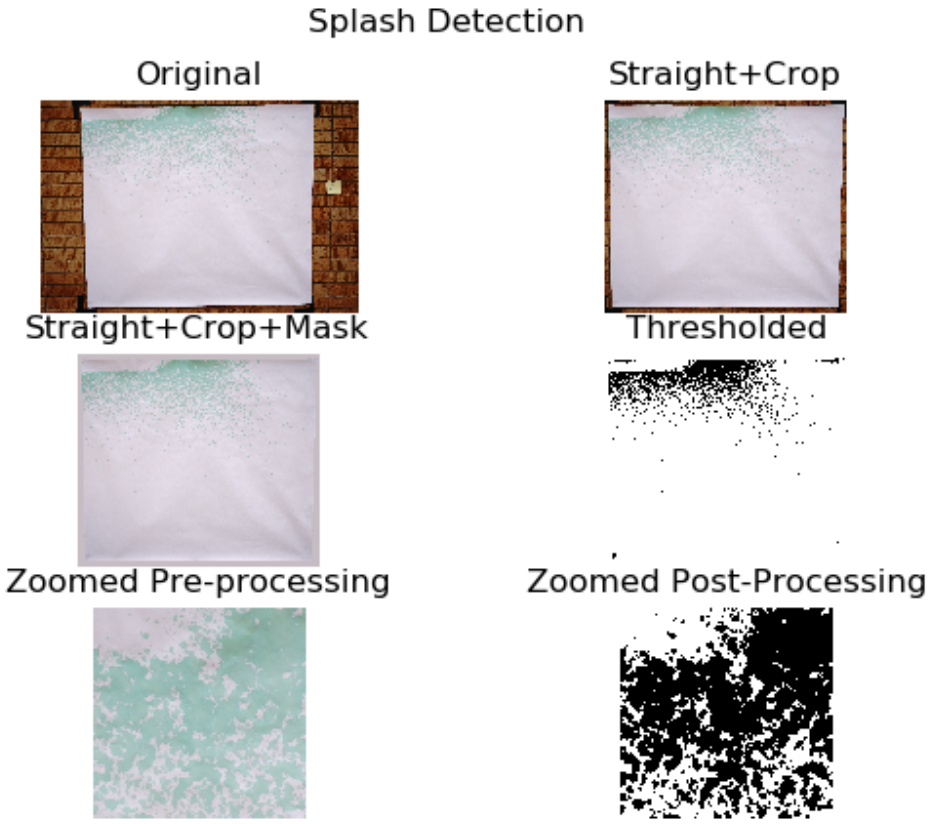



Figure 1: Images

## 3 Ball Drop Experiment

A rubber ball was dropped onto a table and was recorded using a high speed camera. Two scenarios were recorded: a clean, well illuminated white backdrop and another without any backdrop rendering a noisy background. The goal of the analysis was to track the Cartesian position of the ball in the image over time, as well as the deformation of the ball over time. One method is proposed for both situations. First, I will describe the method used. Next, the differences in the two situations will be addressed. Finally, the result images and graphs will be shown.

### 3.1 Methods

The pipeline is as follows:

1. Create replicate background
2. For each image
   (a) Find difference image
   (b) Use edge detection kernel
   (c) Find connected components
   (d) Find largest component (expected to be ball)
   (e) Find the bounds of the ball
   (f) Use them to calculate center, height, and width
3. Plot height, width, and position vs. time

#### 3.1.1 Replicate Background

The background was to be removed to improve segmentation process. In order to accomplish this, a replicate background needed to be created. The process for doing this is as follows:

1. Input two images, $I_1$, $I_2$, that meet criteria that moving object in image is in an entirely unique position.
2. For each input image:
   (a) Take difference of Gaussian blurred current frame and Gaussian blurred next/previous frame.
   (b) Use Sobel kernel to perform edge detection.
   (c) Use Otsu's threshold
   (d) Apply morphological closing to fill in any small gaps/holes.
   (e) Find connected components
   (f) Find largest connected component
   (g) Replace portion of image $I_1$ where the ball exist with the same portion of $I_2$ that is just the background.

Fig. 2 and Fig. 3 show the result of this step.

#### 3.1.2 Segmentation

A similar process is applied that was used to create the replicate background for each image in the video:

1. Take difference of Gaussian blurred current frame and Gaussian blurred replicate background.
2. Use Sobel kernel to perform edge detection.
3. Use Otsu's threshold
4. Apply morphological closing
5. Find connected components
6. Find largest connected component
7. Store information about x and y position, height, and width

Note that the only difference for the two experiments were the parameters of the operations. For the clean background, the Gaussian blur kernels had a sigma of 2, while the noisy background had a sigma of 3. The closing operation used different structural elements.For the clean background, the structural element is a 21x21 array of 1's. For the noisy background, the structural element is a 71x71 array of 1's. The noisy background needed more pre-processing for the connected components algorithm to work.

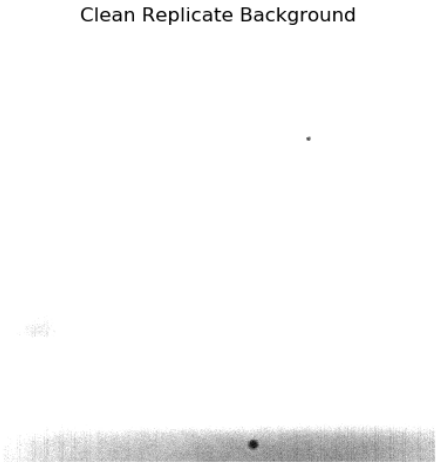



Figure 2: Clean Background


Figure 3: Noisy Background

## 4 Execution Times

I first created the algorithms using the sci-kit image library. This allowed me to create a proof of concept for the pipeline. After I proved functionality, I began to write my own functions. After one design iteration, my code executed very slowly. I never allowed it to run all the way through with this method, due to the time it was taking to process the images. From my estimates, the clean background would have taken 6 hours, while the noisy background would have taken approximately 18 hours. I deemed this unsatisfactory and began researching ways to speed up my code. I was originally using the connected components algorithm as described by Gonzalez and Woods. (2) This proved to work fine, but the computational speed was rather slow. I instead opted to use a more sophisticated version of the same algorithm that is based on performed Union-Find operations.(1) It is a pointer-based version of the first implementation I attempted. All of the code up to this point was written in Python, where pointers do not exist. I decided to learn how to wrap Python with C. The method I implemented is Cython. I wrote a C file that implemented the Union-Find

algorithm as explained in the paper. I then compiled that C file using Cython and imported it as a module into my pure python environment. The results of this algorithm were much better. After only swapping my original implementation for the C-wrapped pointer version, the clean background ran in under 4 minutes and the noisy background in under 8 minutes. After implementing all my own functions, the clean background ran in 14 minutes and the noisy background ran upwards of 3 hours [4]

# 5 Results

## 5.1 Clean Background

This algorithm performed rather well for both cases. The clean background drop was able to measure the height, width and position with only minimal error. The error occurred towards the end of the scene as the ball approached the left edge of the frame.

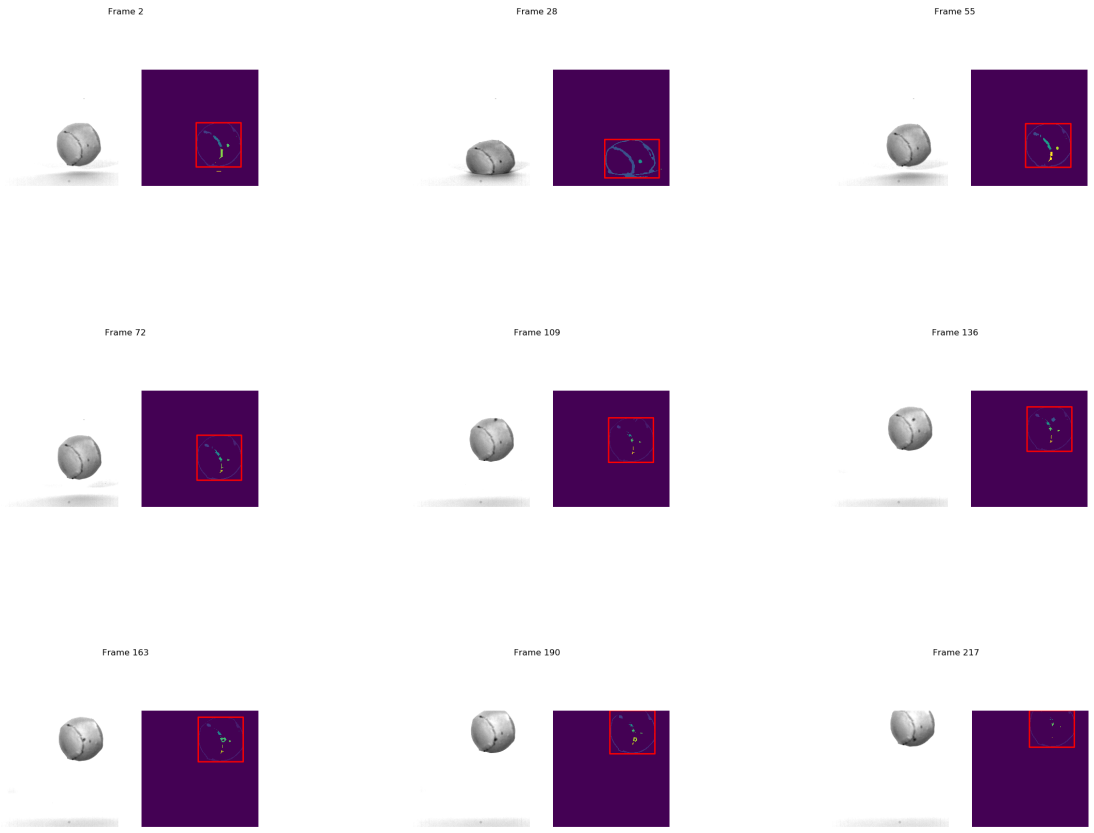
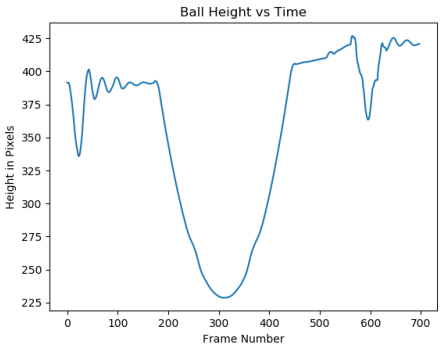
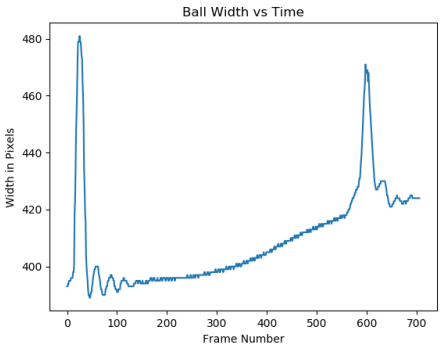



Figure 6: Results of Clean Background Segmentation

[4] I believe this to be due to the large structural element used during the morphological closing step.




(a) Measured height




(b) Measured Width

Figure 7: Clean Background Results

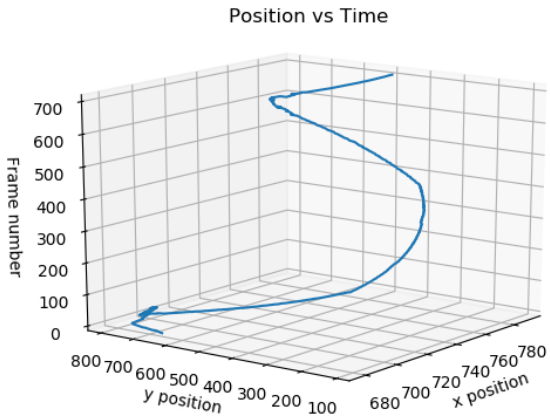



Figure 8: Measured Position Clean Background

### 5.2 Noisy Background

The position tracking of the noisy background experiment worked well. However, the height and width did not work as well as expected. It is still obvious the height/width change when the ball hits the table, but outside of that there is a lot of noise that made thresholding the image difficult.

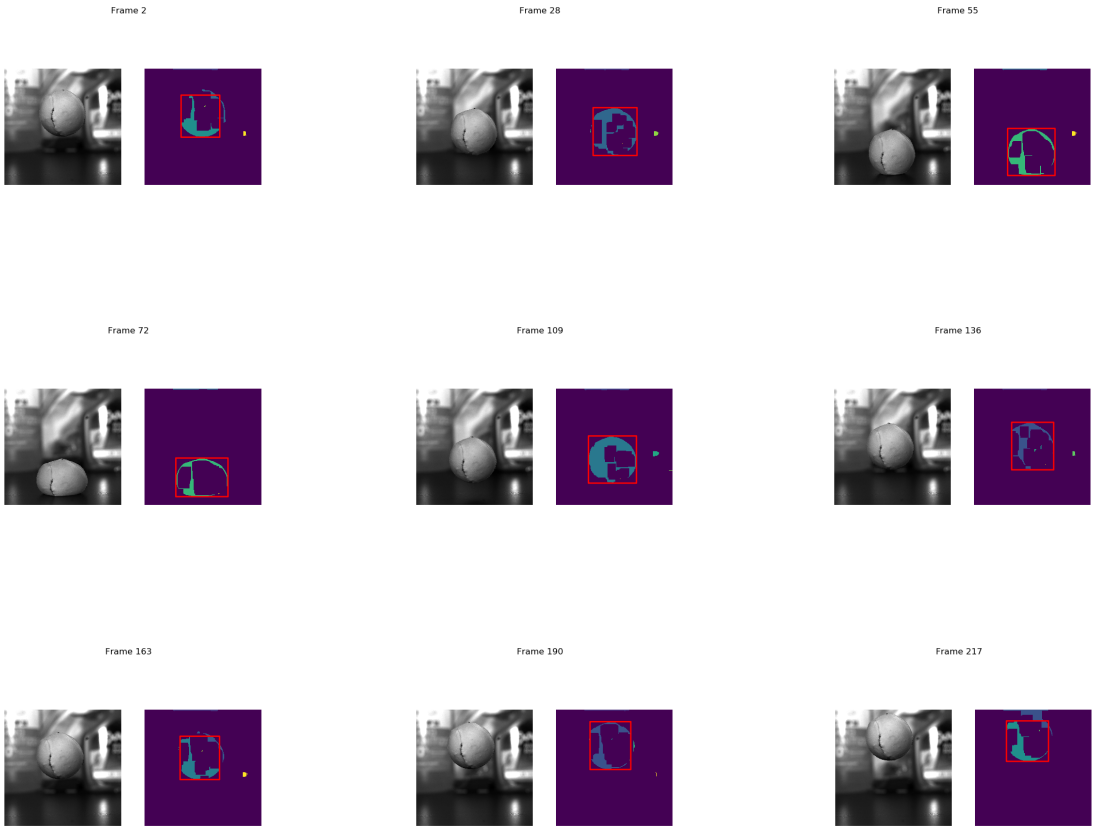
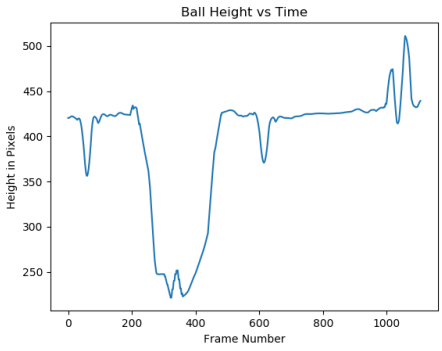
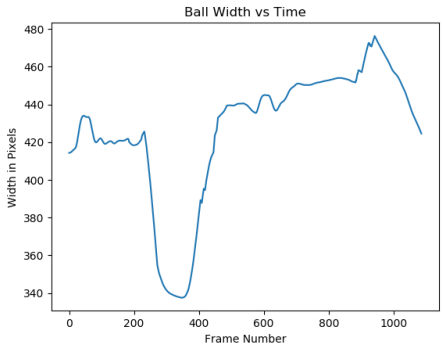



Figure 11: Results of Noisy Background Segmentation




(a) Measured height




(b) Measured Width

Figure 12: Noisy Background Results

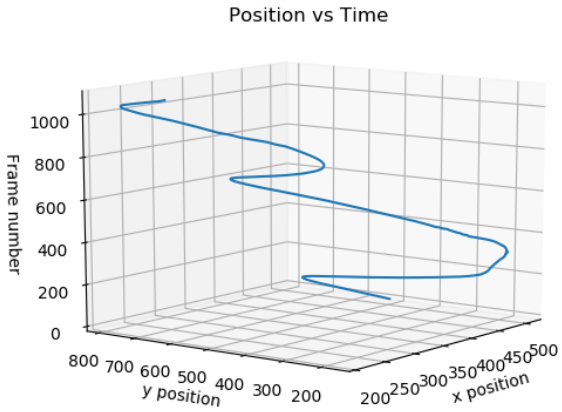



Figure 13: Measured Position Noisy Background

## 6 Conclusion

In conclusion, classical computer vision methods still have a use. They can be implemented to run rather quickly and be rather effective. The experiments worked as I expected them to providing a decent result with a little noise.

## References


[1] Christophe Fiorio and Jens Gustedt. Two linear time union-find strategies for image processing. *Theoretical Computer Science*, 154(2):165–181, 1996.

[2] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

[3] Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep learning vs. traditional computer vision. In *Science and Information Conference*, pages 128–144. Springer, 2019.

## A Appendix A: dye_calculation.py

```
#Author: Nicholas Rabow
#Description: This program is the implentation of segmentation to solve
#an issue for the Mechanical Engineering Department at Weber State University
#An image was provided from an experiment they performed. The image is of a
#piece of paper with dye on it. The overarching system to be created has three
#blocks: Preprocessing-->Processing-->Analysis
#Date: 11/30/2021
#Version:1.0


#Libraries import
import numpy as np
import matplotlib.pyplot as plt
from skimage.util.dtype import dtype_range, dtype_limits
import scipy.stats as st

#constants
DTYPE_RANGE = dtype_range.copy() #DTYPE_RANGE describes the max range for any
                                 #data type that could be input into system.
DTYPE_RANGE.update((d.__name__, limits) for d, limits in dtype_range.items())
DTYPE_RANGE.update({'uint10': (0, 2 ** 10 - 1),
                    'uint12': (0, 2 ** 12 - 1),
                    'uint14': (0, 2 ** 14 - 1),
                    'float': dtype_range[np.float64]})

#Name:image_with_hist
#Parameters: [image]->input image (2D-array like)[title]->title of plot(str)
#Description: Takes normalized histogram of [image] and makes a subplot that
#             contains the image and its' histogram.
#Returns: void (creates fig that can be shown by plt.show())
def image_with_hist(image,title):
    plt.figure()
    hist, bin_edges = np.histogram(image,bins=255,density=True)
    ax1 = plt.subplot2grid(shape=(3,3),loc=(0,0),rowspan=2,colspan=3)
    plt.title(title)
    ax1.axis('off')
    ax2 = plt.subplot2grid(shape=(3,3),loc=(2,0),colspan=3)
    ax1.imshow(image,cmap='gray')
    ax2.bar(bin_edges[0:-1], hist)
    plt.tight_layout()

#Name: rotate_image
#Parameters: [i]->input image (2D or 3D array-like),[degrees]->rotation
# parameter(int), [output]->mode of operation(str) by default is 'crop'
#Description: Takes input image and performs rotation of the that image by
# [degrees] Full mode keeps the entire imge, while Crop mode keeps the inital
# image size
#Returns: Rotated Image->2D or 3D numpy array(uint8)
def rotate_image(i, degrees, output="crop"):

    assert output in ["crop", "full"], "output should be either 'crop' or 'full'"
    rot_rad = degrees * np.pi / 180.0 #convert degres to rad
```

```
rotate_m = np.array([[np.cos(rot_rad), np.sin(rot_rad)],
                     [-np.sin(rot_rad), np.cos(rot_rad)]])
                     #affine transform rotation matrix

# If output_scale = "full", the image must be inserted into a bigger frame,
#so the coordinates would be translated
# appropriately.
gray_scale = False #check for color space (gray or color)
if len(i.shape) < 3:
    img = i.reshape(*i.shape, 1)
    gray_scale = True

h, w, c = i.shape
if output == "full":
    dia = int(np.sqrt(h**2 + w**2))   #the diagonal is the longest line in
    # the rectangle
    i_pad = np.zeros((dia, dia, c))
    c_h = int((dia - h) // 2)
    c_w = int((dia - w) // 2)
    i_pad[c_h:-c_h, c_w:-c_w, :] = i
    i = i_pad
    i_r = np.zeros((dia, dia, c))
    h, w, c = i.shape
else:
    i_r = np.zeros((h, w, c))

# Rotate and shift the indices, from PICTURE to SOURCE
indices_org = (np.array(np.meshgrid(np.arange(h),
                np.arange(w))).reshape(2, -1))
indices_new = indices_org.copy()
# Apply the affineWrap
indices_new = np.dot(rotate_m, indices_new).astype(int)
mu1 = np.mean(indices_new, axis=1).astype(int).reshape((-1, 1))
mu2 = np.mean(indices_org, axis=1).astype(int).reshape((-1, 1))
indices_new += (mu2-mu1)   # Shift the image back to the center

# Remove the pixels in the rotated image, that are now out of the bounds of
# the result image (Note that the result image is a rectangle of shape
# (h,w,c) that the rotated image is inserted into, so in the case of a
#"full" output_scale, these are just black pixels from the padded image...).
t0, t1 = indices_new
t0 = (0 <= t0) & (t0 < h)
t1 = (0 <= t1) & (t1 < w)
valid = t0 & t1
indices_new = indices_new.T[valid].T
indices_org = indices_org.T[valid].T

xind, yind = indices_new
xi, yi = indices_org
i_r[xi, yi, :] = i[xind, yind, :]

if gray_scale:
    i_r = i_r.reshape((h, w))
```

```
    return i_r.astype(np.uint8)

#Name: show
#Parameters: [f]->input image(2D array like),[title]->plot title(str)
#Description: This function was used for debugging purposes. It reduced the amount
#             of time spent debugging
#Returns: void (creates fig to be shown by plt.show())
def show(f,title):
    plt.figure()
    plt.imshow(f,cmap='gray')
    plt.title(title)


#Name: morpho_dilate
#Parameters: [I]->input image (2D array like),[B]->structing element of MxN size
#            M,N need to be odd. (2D-arraylike)
#Description: Performs morphological dilation on image using SE [B]
#Returns: Image after dilation is perfomed.
#input needs background of zeros and foreground of 1. foreground to be grown
def morpho_dilate(I,B):
    m,n = B.shape #shape of SE
    x,y = I.shape #shape of input image
    p = m//2 #center of SE
    q = n//2 #center of SE
    output = np.zeros((x,y),dtype=bool) #init output array
    #pad image to allow 'movement'
    I = np.pad(I,((p,p),(q,q)),'constant',constant_values=0)
    for i in range(m): #loop through each element of B
        for j in range(n):
            output |= (I[i:i+x,j:j+y].astype(bool) & B[i,j].astype(bool))
            #Logical and every element in B with shifted input image, this result
            #is or-ed with the current output array
    return output.astype(int) #convert output from bool to int type array

#Name: rgb2gray
#Parameters: [i]->input image (3D arraylike)
#Description: Converts color image to grayscale image
#Returns: returns grayscale image
def rgb2gray(i):
    return np.dot(i,[0.2989,0.5870,0.1140]) #coeffs to weight colors according
                                            #to human perception
#Name: contrast_highest
#Parameters: [f]->input image(3D arraylike)
#Description: Takes color image and checks for highest contrast channel
#Returns: Returns index of highest constrast channel
def contrast_highest(f):
    std = np.array([]) #init standard dev. array
    for i in range(f.shape[2]): #loop for each channel
        std = np.append(std,np.std(f[:,:,i])) #standard dev. of each channel
    return np.argmax(std) #index of highest constrast image.

#Name: otsu_thresh
#Parameters: [f]->input image (2D-arraylike)
#Description: given input image, calculates otsu threshold
```

```
#Returns: otsu threshold (int)
def otsu_thresh(f):
    hist, b_ed = np.histogram(f,bins=256)
    b_c = (b_ed[:-1] + b_ed[1:]) / 2
    hist = hist / np.sum(hist)
    hist = hist.astype(float)

    # class probabilities for all possible thresholds
    w1 = np.cumsum(hist)
    w2 = np.cumsum(hist[::-1])[::-1]
    # class means for all possible thresholds
    m1 = np.cumsum(hist * b_c) / w1
    m2 = (np.cumsum((hist * b_c)[::-1]) / w2[::-1])[::-1]

    # Clip ends to align class 1 and class 2 variables:
    # The last value of ``weight1``/``mean1`` should pair with zero values in
    # ``weight2``/``mean2``, which do not exist.
    class_v = w1[:-1] * w2[1:] * (m1[:-1] - m2[1:]) ** 2

    idx = np.argmax(class_v)
    T = b_c[idx]
    return T

#Name: rescale_intensity
#Parameters: [image]->input image (2D arraylike),[in_range]->mode of
#            operation(str) [out_range]->mode of operation(str)
#Description:Takes image and rescales the intensity of the image,
#            (contrast enhancement) in_range is used to describe intensity
#            range, out_range is used to determine intensity range of
#             output image
#Returns:Intensity rescale image->(2D array like)
def rescale_intensity(image, in_range='image', out_range='dtype'):
    #find data type for output image
    if out_range in ['dtype', 'image']:
        out_dtype = output_dtype(image.dtype.type, image.dtype)
    else:
        out_dtype = output_dtype(out_range, image.dtype)
    #calculate input range min/max
    imin, imax = map(float, intensity_range(image, in_range))
    #calculate output range min/max
    omin, omax = map(float, intensity_range(image, out_range,
                                            clip_negative=(imin >= 0)))
    #apply input range to image
    image = np.clip(image, imin, imax)

    if imin != imax: #apply input and output range restrictions
        image = (image - imin) / (imax - imin)
        return np.asarray(image * (omax - omin) + omin, dtype=out_dtype)
    else: #apply only output range restrictions
        return np.clip(image, omin, omax).astype(out_dtype)
#Name:output_dtype
#Parameters:[dtype_or_range]->output dtype,[image_dtype]->datatype of image
#Description:Determines output dtype for function rescale_intensity()
#            following rules:
```

```
#       - if ``dtype_or_range`` is a dtype, that is the output dtype.
#       - if ``dtype_or_range`` is a dtype string, that is the dtype used
#       - if ``dtype_or_range`` is a pair of values, the output data type will be
#       ``_supported_float_type(image_dtype)``
#Returns: out_dtype-> date type for desired output (type)
def output_dtype(dtype_or_range, image_dtype):
    if type(dtype_or_range) in [list, tuple, np.ndarray]:
        # pair of values: always return float.
        return utils._supported_float_type(image_dtype)
    if type(dtype_or_range) == type:
        # already a type: return it
        return dtype_or_range
    if dtype_or_range in DTYPE_RANGE:
        # string key in DTYPE_RANGE dictionary
        try:
            # if it's a canonical numpy dtype, convert
            return np.dtype(dtype_or_range).type
        except TypeError:  # uint10, uint12, uint14
            # otherwise, return uint16
            return np.uint16
#Name:intensity_range
#Parameters:[image]->input image(2D arraylike),[range_values]->mode of operation,
#           [clip_negative]->mode of operation(bool)
#Description:Takes image and range values and calculates min and max intensity
#            values of image
#Returns:min and max intensity value (int)
def intensity_range(image, range_values='image', clip_negative=False):
    if range_values == 'dtype': #range if full scale of datatype
        range_values = image.dtype.type

    if range_values == 'image': #use images to find min/max intensity value
        i_min = np.min(image)
        i_max = np.max(image)
    elif range_values in DTYPE_RANGE:
        i_min, i_max = DTYPE_RANGE[range_values]
        if clip_negative:
            i_min = 0
    else:
        i_min, i_max = range_values
    return i_min, i_max


#Main script to be run.
#Pipeline:
#Preprocessing->Processing->Analysis
#Preprocessing:
#   1)Read image
#   2)Rotate image
#   3)Rough Crop Image
#   4)Remove left over background
#Processing:
#   1)Contrast enhancement
#   2)Otsu Threshold
#Analysis:
```

```
#   1)Calculate percentage of paper segmented
def main():
    fig, ax = plt.subplots(3,2) #3x2 subplot
    fig.suptitle("Splash Detection")
    for i in range(ax.shape[0]): #remove axis from all plots
        for j in range(ax.shape[1]):
            ax[i,j].axis('off')
    #Read the input
    f = plt.imread('dye_on_paper.jpg')
    f = f.copy()
    print("Input Read")
    ax[0,0].imshow(f)
    ax[0,0].set_title("Original")


    ###############
    #PREPROCESSING#
    ###############
    print("PREPROCESSING")
    #ROTATE IMAGE
    print("Rotate")
    r = rotate_image(f,-1) #rotate image by -1 deg. clockwise (not automated)

    #ROUGH CROP IMAGE (automated)
    print('rough crop')
    g = rgb2gray(r) #convert to grayscale
    from skimage import filters
    gb = filters.gaussian(g,10) #blur image
    gbt = otsu_thresh(gb) #calculate otsu threshold
    gb = gb > gbt #apply otsu thresh
    flag = True
    while(flag): #crop needs to find three similar numbers
         #find 3 random rows between the 25 and 75 percentile
        rows = np.round(np.random.uniform(0.25,0.75,5)*(g.shape[0])).astype(int)
         #find 3 random cols between the 25 and 75 percentile
        cols = np.round(np.random.uniform(0.25,0.75,5)*g.shape[1]).astype(int)
        el = np.array([]) #arrays to hold edge values (left,right,top,bottom)
        er = el
        et = er
        eb = et

        for i in range(rows.shape[0]): #find left and right edge values and
                                       # add to respective arrays
            tmp = gb[rows[i]:rows[i]+1,].flatten()
            el = np.append(el,np.min(np.where(tmp==True)))
            er = np.append(er,np.max(np.where(tmp==True)))
        for i in range(cols.shape[0]): #find top and bottom edge values and add
                                       # to respective array
            tmp = gb[:,cols[i]:cols[i]+1].flatten()
            et = np.append(et,np.min(np.where(tmp==True)))
            eb = np.append(eb,np.max(np.where(tmp==True)))
        #if std.dev. is greater than 100 then repeat above process
        if np.std(el)>100 or np.std(er)>100 or np.std(et)>100 or np.std(eb)>100:
            pass
```

```
    else:
        flag = False
b = 100 #buffer for rough crop to verify entire image is in frame
cr = (r[int(np.mean(et))-b:int(np.mean(eb))+b,int(np.mean(el))
     -b:int(np.mean(er))+b]) #rough crop according to calculated values
ax[0,1].imshow(cr)
ax[0,1].set_title("Straight+Crop")

#REMOVE LEFT OVER BACKGROUND
ch = contrast_highest(cr) #find which channel has the highest contrast
T = otsu_thresh(cr[:,:,ch]) #find otsu_thresh of crop+straight image
mask = cr[:,:,ch] < T #threshold majority of brick background
mask = morpho_dilate(mask,np.ones((3,3))) #dilate mask to improve edges
mask = np.invert(mask.astype(bool)) #invert to be used as mask
paper_area = np.count_nonzero(mask) #paper area is calculated for use in
                                    #analysis section
for i in range(3): #loop for each channel
    cr[:,:,i][mask == False] = np.mean(cr[:,:,i]) #apply mask to remove
                                                  #brick background
ax[1,0].imshow(cr)
ax[1,0].set_title("Straight+Crop+Mask")
plt.figure()
plt.imshow(cr)
plt.axis('off')
plt.title("Removed␣Background␣after␣Rough␣Crop")

############
#PROCESSING#
############
ch = contrast_highest(cr) #find highest contrast channel for processing
print('PROCESSING')
low = 3 #low percentile used for rescaling intensity
high = 30 #high percentile used for rescaling intensity
pl, ph = np.percentile(cr[:,:,ch],(low,high))
print(pl,ph)
i_ri = rescale_intensity(cr[:,:,ch],in_range=(pl,ph)) #rescale intensity
T = otsu_thresh(i_ri) #take otsu threshold
thresh = i_ri > T #apply otsu threshold
ax[1,1].imshow(thresh,cmap='gray')
ax[1,1].set_title("Thresholded")
cover_area = np.count_nonzero(np.invert(thresh.astype(bool)))
plt.figure()
plt.imshow(thresh,cmap='gray')
plt.axis('off')
plt.title("Threshold")

##########
#ANALYSIS#
##########
print("ANALYSIS")
print("Paper␣Area␣" + str(paper_area))
print("Dye␣Area␣" + str(cover_area))
print(("Percentage␣of␣Paper␣Covered␣"\
     +str(np.round(cover_area/paper_area*100,2))))
```

```
    crop_og = cr[200:700,1200:1700]
    ax[2,0].imshow(crop_og)
    ax[2,0].set_title("Zoomed␣Pre-processing")
    crop_t = thresh[200:700,1200:1700]
    ax[2,1].imshow(crop_t,cmap='gray')
    ax[2,1].set_title("Zoomed␣Post-Processing")

    plt.show()
if __name__ == "__main__":
    main()
```

## B Appendix B: clean_bounce.py

```
#Author: Nicholas Rabow
#Description: The purpose of this program is to segment a video that was filmed
# with a high speed camera. A ball was dropped with a clean background. There are
# 707 images to process. The goal is to segment the image in order to be able to
# track the center of the ball as well as the height and width.
#Date: 12/8/2021

#Libraries
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from matplotlib.patches import FancyBboxPatch
from skimage import filters #used for gaussian and sobel kernels
import con_comps

from skimage import morphology

#Name:moving_average
#Parameters: [x]->1D array,[w]->MA size
#Description: Calculates MA of size [w] on signal [x]
def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w
#Name:morpho_dilate
#Parameters:[I]->input image (2D array like),[B]->structing element of MxN size
#           M,N need to be odd. (2D-arraylike)
#Description: Performs morphological dilation on image using SE [B]
#Returns:Image after dilation is perfomed.
#input needs background of zeros and foreground of 1. foreground to be grown
def morpho_dilate(I,B):
    m,n = B.shape #shape of SE
    x,y = I.shape #shape of input image
    p = m//2 #center of SE
    q = n//2 #center of SE
    output = np.zeros((x,y),dtype=bool) #init output array
    I = np.pad(I,((p,p),(q,q)),'constant',constant_values=0)#pad image to allow
                                                            #'movement'
    for i in range(m): #loop through each element of B
        for j in range(n):
            output |= (I[i:i+x,j:j+y].astype(bool) & B[i,j].astype(bool))
            #Logical and every element in B with shifted input image, this
```

```
            #result is or-ed with the current output array
    return output.astype(int) #convert output from bool to int type array

#Name:morpho_erode
#Parameters:[I]->input image (2D array like),[B]->structing element of MxN size
#           M,N need to be odd. (2D-arraylike)
#Description: Performs morphological erosion on image using SE [B]
#Returns:Image after erosion is perfomed.
#input needs background of zeros and foreground of 1. foreground to be shrunken
def morpho_erode(I,B):
    m,n = B.shape #shape of SE
    x,y = I.shape #shape of input image
    p = m//2 #center of SE
    q = n//2 #center of SE
    output = np.ones((x,y),dtype=bool) #init output array
    I = np.pad(I,((p,p),(q,q)),'constant',constant_values=0)#pad image to allow
                                                            #'movement'
    for i in range(m): #loop through each element of B
        for j in range(n):
            output &= (I[i:i+x,j:j+y].astype(bool) | (not B[i,j]))
            #Logical or every element in (not B) with shifted input image, this
            #result is and-ed with the current output array
    return output.astype(int) #convert output from bool to int type array

#Name:otsu_thresh
#Parameters:[f]->input image (2D-arraylike)
#Description: given input image, calculates otsu threshold
#Returns: otsu threshold (int)
def otsu_thresh(f):
    hist, b_ed = np.histogram(f,bins=256)
    b_c = (b_ed[:-1] + b_ed[1:]) / 2
    hist = hist / np.sum(hist)
    hist = hist.astype(float)

    # class probabilities for all possible thresholds
    w1 = np.cumsum(hist)
    w2 = np.cumsum(hist[::-1])[::-1]
    # class means for all possible thresholds
    m1 = np.cumsum(hist * b_c) / w1
    m2 = (np.cumsum((hist * b_c)[::-1]) / w2[::-1])[::-1]

    # Clip ends to align class 1 and class 2 variables:
    # The last value of ``weight1``/``mean1`` should pair with zero values in
    # ``weight2``/``mean2``, which do not exist.
    class_v = w1[:-1] * w2[1:] * (m1[:-1] - m2[1:]) ** 2

    idx = np.argmax(class_v)
    T = b_c[idx]
    return T

#Psuedocode
#1)Create Replicate Background using difference method
#2)For each image in video:
#   a)Find difference of gaussian(image) and gaussian(background)
```

```
#   b)Use sobel kernel for edge Detection
#   c)Apply closing
#   d)Find connected components
#   e)largest component is suspected ball
#   f)find bounds of ball and use them to calculate center, width and height
#3)Plot height,width,and position vs time
def main():
    plot = 0
    #CREATE REPLICATE BACKGROUND
    #read first image
    f1 = plt.imread('drop1_C001H001S0001\drop1_C001H001S0001000001.tif')
    #read second image
    f2 = plt.imread('drop1_C001H001S0001\drop1_C001H001S0001000002.tif')
    x = filters.gaussian(f2,1)-filters.gaussian(f1,1) #creat difference image
    x = filters.sobel(x)
    T = otsu_thresh(x) #thresh the difference image
    x = x > T
    dil = morpho_dilate(x,np.ones((21,21))) #apply closing to the thresholded image
    ero = morpho_erode(dil,np.ones((21,21)))
    a,bc = con_comps.label_cython(ero,background=False,return_num=True) #use
                                                    # connected components algorithm
    tmp = 0
    for j in range(1,bc+1): #loop to find largest component
        area = np.sum(a[a==j])/j
        if tmp < area:
            tmp = area
            lab_val= j #value of label of largest component
    lab_coords = np.array(np.where(a==lab_val)) #find coordinates of all
                                                    # label pixels
    rows = lab_coords[0] #x-coordinates
    cols = lab_coords[1] #y-coordinates
    #min row, min col, max row, max col of label
    box1 = [np.min(rows),np.min(cols),np.max(rows),np.max(cols)]
    y = np.mean((box1[0],box1[2])) #y coordinate for center
    x = np.mean((box1[1],box1[3])) #x coordinate for center
    h = box1[2]-box1[0]  #height in pixels
    w = box1[3]-box1[1]  #width in pixels
    #repeat same process as above.
    f3 = plt.imread('drop1_C001H001S0001\drop1_C001H001S0001000180.tif')
    f4 = plt.imread('drop1_C001H001S0001\drop1_C001H001S0001000190.tif')
    x = filters.gaussian(f4,1)-filters.gaussian(f3,1)
    x = filters.sobel(x)
    T = otsu_thresh(x)
    x = x > T
    dil = morpho_dilate(x,np.ones((21,21)))
    ero = morpho_erode(dil,np.ones((21,21)))
    a,bc = con_comps.label_cython(ero,background=False,return_num=True)
    tmp = 0
    for j in range(1,bc+1):
        area = np.sum(a[a==j])/j
        if tmp < area:
            tmp = area
            lab_val= j
    lab_coords = np.array(np.where(a==lab_val))
```

```
rows = lab_coords[0]
cols = lab_coords[1]
box2 = [np.min(rows),np.min(cols),np.max(rows),np.max(cols)]
y = np.mean((box2[0],box2[2]))
x = np.mean((box2[1],box2[3]))
h = box2[2]-box2[0]
w = box2[3]-box2[1]
#use portion of f1 to replicate background with no foreground
f3[box2[0]:box2[2],box2[1]:box2[3]] = f1[box2[0]:box2[2],box2[1]:box2[3]]
back = f3


############
#IMAGE LOOP#
############
#READ EACH IMAGE
h = np.empty(0) #initialize analysis arrays
w = np.empty(0)
x = np.empty(0)
y = np.empty(0)
#FOR EACH IMAGE IN VIDEO:
for i in range(1,708):
# for n,i in enumerate([2,28,55,72,109,136,163,190,217]): #used for demo
    print(i)
    if i < 10:
        f = plt.imread('drop1_C001H001S0001\drop1_C001H001S000100000'+str(i)+'.
    elif i >= 10 and i < 100:
        f = plt.imread('drop1_C001H001S0001\drop1_C001H001S00010000'+str(i)+'.t
    elif i >= 100 and i < 1000:
        f = plt.imread('drop1_C001H001S0001\drop1_C001H001S0001000'+str(i)+'.ti

    g = filters.gaussian(f,2)-filters.gaussian(back,2) #Find difference
    g = filters.sobel(g) #apply sobel kernel for edge detection
    T = filters.threshold_otsu(g) #use otsu thresholding
    g = g < T
    dil = morpho_dilate(np.invert(g),np.ones((11,11))) #apply closing
    ero = morpho_erode(dil,np.ones((11,11)))
    #find connected components
    a,bc = con_comps.label_cython(ero,background=False,return_num=True)
    tmp = 0
    for j in range(1,bc+1): #find label with largest area
        area = np.sum(a[a==j])/j
        if tmp < area:
            tmp = area
            lab_val= j #value of pixels in largest label
    #coordinates of all pixels in largest label
    lab_coords = np.array(np.where(a==lab_val))
    rows = lab_coords[0]    #x coordinates of label
    cols = lab_coords[1]    #y coordinates of label
    #find edges of label
    box = [np.min(rows),np.min(cols),np.max(rows),np.max(cols)]
    y = np.append(y,np.mean((box[0],box[2]))) #y coordinate center
    x = np.append(x,np.mean((box[1],box[3]))) #x coordinate center
    h = np.append(h,box[2]-box[0]) #height in pixels
```

```
        w  = np.append(w,box[3]-box[1]) #width in pixels

        if plot:
            fig,ax = plt.subplots(1,2)
            plt.suptitle("Frame "+str(i))
            ax[0].imshow(f,cmap='gray')
            ax[0].axis('off')
            y = np.append(y,np.mean((box[0],box[2])))
            x = np.append(x,np.mean((box[1],box[3])))
            h = np.append(h,box[2]-box[0])
            w  = np.append(w,box[3]-box[1])
            rect = (FancyBboxPatch((box[1],box[0]),w[-1],h[-1],
                    fill=False,linewidth=2,edgecolor='red'))
            ax[1].imshow(a)
            ax[1].add_patch(rect)
            ax[1].axis('off')
    #######
    #PLOTS#
    #######
    print("height")
    print(h.shape)
    plt.figure()
    plt.title("Ball Height vs Time")
    plt.xlabel("Frame Number")
    plt.ylabel("Height in Pixels")
    plt.plot(moving_average(h,10),label='10')
    print("width")
    print(w.shape)
    plt.figure()
    plt.title("Ball Width vs Time")
    plt.xlabel("Frame Number")
    plt.ylabel("Width in Pixels")
    plt.plot(w)
    print("position")
    print(x.shape,y.shape)
    plt.figure()
    ax = plt.axes(projection='3d')
    ax.set_title("Position vs Time")
    ax.plot3D(x,y,np.arange(707))
    ax.set_xlabel('x position')
    ax.set_ylabel('y position')
    ax.set_zlabel('Frame number')




    plt.show()

if __name__ == '__main__':
    main()
```

## C Appendix C: dirty_bounce.py

```
#Author: Nicholas Rabow
```

```
#Description: The purpose of this program is to segment a video that was filmed
# with a high speed camera. A ball was dropped with a noisy background. There are
# 1126 images to process. The goal is to segment the image in order to be able to
# track the center of the ball as well as the height and width.
#Date: 12/8/2021

#Libraries
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import FancyBboxPatch
from mpl_toolkits import mplot3d
from skimage import filters #gaussian and sobel kernel used from this library
import con_comps

from skimage import morphology

#Name: moving_average
#Parameters: [x]->1D array,[w]->MA size
#Description: Calculates MA of size [w] on signal [x]
def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w

#Name: morpho_dilate
#Parameters:[I]->input image (2D array like),[B]->structing element of MxN size
#           M,N need to be odd. (2D-arraylike)
#Description: Performs morphological dilation on image using SE [B]
#Returns:Image after dilation is perfomed.
#input needs background of zeros and foreground of 1. foreground to be grown
def morpho_dilate(I,B):
    m,n = B.shape #shape of SE
    x,y = I.shape #shape of input image
    p = m//2 #center of SE
    q = n//2 #center of SE
    output = np.zeros((x,y),dtype=bool) #init output array
    I = np.pad(I,((p,p),(q,q)),'constant',constant_values=0)#pad image to allow
    # 'movement'
    for i in range(m): #loop through each element of B
        for j in range(n):
            output |= (I[i:i+x,j:j+y].astype(bool) & B[i,j].astype(bool))
            #Logical and every element in B with shifted input image, this result
            #is or-ed with the current output array
    return output.astype(int) #convert output from bool to int type array

#Name: morpho_erode
#Parameters:[I]->input image (2D array like),[B]->structing element of MxN size
#           M,N need to be odd. (2D-arraylike)
#Description: Performs morphological erosion on image using SE [B]
#Returns:Image after erosion is perfomed.
#input needs background of zeros and foreground of 1. foreground to be shrunken
def morpho_erode(I,B):
    m,n = B.shape #shape of SE
    x,y = I.shape #shape of input image
    p = m//2 #center of SE
    q = n//2 #center of SE
```

```
    output = np.ones((x,y),dtype=bool) #init output array
    I = np.pad(I,((p,p),(q,q)),'constant',constant_values=0)#pad image to allow
    # 'movement'
    for i in range(m): #loop through each element of B
        for j in range(n):
            output &= (I[i:i+x,j:j+y].astype(bool) | (not B[i,j]))
            #Logical or every element in (not B) with shifted input image, this
            #result is and-ed with the current output array
    return output.astype(int) #convert output from bool to int type array

#Name:otsu_thresh
#Parameters:[f]->input image (2D-arraylike)
#Description: given input image, calculates otsu threshold
#Returns: otsu threshold (int)
def otsu_thresh(f):
    hist, b_ed = np.histogram(f,bins=256)
    b_c = (b_ed[:-1] + b_ed[1:]) / 2
    hist = hist / np.sum(hist)
    hist = hist.astype(float)

    # class probabilities for all possible thresholds
    w1 = np.cumsum(hist)
    w2 = np.cumsum(hist[::-1])[::-1]
    # class means for all possible thresholds
    m1 = np.cumsum(hist * b_c) / w1
    m2 = (np.cumsum((hist * b_c)[::-1]) / w2[::-1])[::-1]

    # Clip ends to align class 1 and class 2 variables:
    # The last value of ``weight1``/``mean1`` should pair with zero values in
    # ``weight2``/``mean2``, which do not exist.
    class_v = w1[:-1] * w2[1:] * (m1[:-1] - m2[1:]) ** 2

    idx = np.argmax(class_v)
    T = b_c[idx]
    return T

###############
#MAIN FUNCTION#
###############
#Psuedocode
#1)Create Replicate Background using difference method
#2)For each image in video:
#   a)Find difference of gaussian(image) and gaussian(background)
#   b)Use sobel kernel for edge Detection
#   c)Apply closing
#   d)Find connected components
#   e)largest component is suspected ball
#   f)find bounds of ball and use them to calculate center, width and height
#3)Plot height,width,and position vs time
def main():
    plot = 0 #used for creating plots of each image
    #CREATE REPLICATE BACKGROUND
    #read first image
    f1 = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S0001000020.tif')
```

```
#read second image
f2 = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S0001000021.tif')
x = filters.gaussian(f2,5)-filters.gaussian(f1,5) #take difference
T = otsu_thresh(x) #thresh the difference image
x = x > T
dil = morpho_dilate(x,np.ones((21,21))) #apply closing
ero = morpho_erode(dil,np.ones((21,21)))
 #use connected components algorithm
tmp = 0
a,bc = con_comps.label_cython(ero,background=False,return_num=True)
for j in range(1,bc+1): #loop to find largest component
    area = np.sum(a[a==j])/j
    if tmp < area:
        tmp = area
        lab_val= j #value of label of largest component
#find coordinates of all label pixels
lab_coords = np.array(np.where(a==lab_val))
rows = lab_coords[0] #x-coordinates
cols = lab_coords[1] #y-coordinates
#min row, min col, max row, max col of label
box1 = [np.min(rows),np.min(cols),np.max(rows),np.max(cols)]
y = np.mean((box1[0],box1[2])) #y coordinate for center
x = np.mean((box1[1],box1[3])) #x coordinate for center
h = box1[2]-box1[0]  #height in pixels
w = box1[3]-box1[1]  #width in pixels
#repeat same process as above.
f3 = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S0001000330.tif')
f4 = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S0001000220.tif')
x = filters.gaussian(f4,10)-filters.gaussian(f3,10)
T = otsu_thresh(x)
x = x > T
dil = morpho_dilate(x,np.ones((21,21)))
ero = morpho_erode(dil,np.ones((21,21)))
a,bc = con_comps.label_cython(ero,background=False,return_num=True)
tmp = 0
for j in range(1,bc+1):
    area = np.sum(a[a==j])/j
    if tmp < area:
        tmp = area
        lab_val= j
lab_coords = np.array(np.where(a==lab_val))
rows = lab_coords[0]
cols = lab_coords[1]
box2 = [np.min(rows),np.min(cols),np.max(rows),np.max(cols)]
y = np.mean((box2[0],box2[2]))
x = np.mean((box2[1],box2[3]))
h = box2[2]-box2[0]
w = box2[3]-box2[1]
 #use portion of f1 to replicate background with no foreground
back = f3
f3[box2[0]:box2[2],box2[1]:box2[3]] = f1[box2[0]:box2[2],box2[1]:box2[3]]
plt.figure()
plt.imshow(back,cmap='gray')
plt.axis('off')
```

```
plt.title("Noisy Replicate Background")
plt.show()


h = np.array([]) #initilize arrays for plotting
w = np.array([])
x = np.array([])
y = np.array([])

############
#IMAGE LOOP#
############
for i in range(1,1127):
# for n,i in enumerate([2,28,55,72,109,136,163,190,217]): #used for demonstrati
    #READ EACH IMAGE
    print(i)
    if i < 10:
        f = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S000100000'+s
    elif i >= 10 and i < 100:
        f = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S00010000'+st
    elif i >= 100 and i < 1000:
        f = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S0001000'+str
    elif i > 1000:
        f = plt.imread('dirtydrop_C001H001S0001\dirtydrop_C001H001S000100'+str(


    g = filters.gaussian(f,3)-filters.gaussian(back,3) #Find difference
    g = filters.sobel(g) #apply sobel kernel for edge detection
    T = filters.threshold_otsu(g) #use otsu thresholding
    g = g < T
    dil = morpho_dilate(np.invert(g),np.ones((71,71))) #apply closing
    ero = morpho_erode(dil,np.ones((71,71)))
     #find connected components
    a,bc = con_comps.label_cython(ero,background=False,return_num=True)
    tmp = 0
    for j in range(1,bc+1): #find label with largest area
        area = np.sum(a[a==j])/j
        if tmp < area:
            tmp = area
            lab_val= j #value of pixels in largest label
     #coordinates of all pixels in largest label
    lab_coords = np.array(np.where(a==lab_val))
    rows = lab_coords[0]    #x coordinates of label
    cols = lab_coords[1]    #y coordinates of label
    #find edges of label
    box = [np.min(rows),np.min(cols),np.max(rows),np.max(cols)]
    y = np.append(y,np.mean((box[0],box[2]))) #y coordinate center
    x = np.append(x,np.mean((box[1],box[3]))) #x coordinate center
    h = np.append(h,box[2]-box[0]) #height in pixels
    w  = np.append(w,box[3]-box[1]) #width in pixels

    if plot:
        fig,ax = plt.subplots(1,2)
        plt.suptitle("Frame "+str(i))
```

```
            ax[0].imshow(f,cmap='gray')
            ax[0].axis('off')
            y = np.append(y,np.mean((box[0],box[2])))
            x = np.append(x,np.mean((box[1],box[3])))
            h = np.append(h,box[2]-box[0])
            w  = np.append(w,box[3]-box[1])
            rect = (FancyBboxPatch((box[1],box[0]),w[-1],h[-1],
                    fill=False,linewidth=2,edgecolor='red'))
            ax[1].imshow(a)
            ax[1].add_patch(rect)
            ax[1].axis('off')
    #######
    #PLOTS#
    #######
    print("height")
    print(h.shape)
    plt.figure()
    plt.title("Ball Height vs Time")
    plt.xlabel("Frame Number")
    plt.ylabel("Height in Pixels")
    plt.plot(moving_average(h,20),label='20')
    print("width")
    print(w.shape)
    plt.figure()
    plt.title("Ball Width vs Time")
    plt.xlabel("Frame Number")
    plt.ylabel("Width in Pixels")
    plt.plot(moving_average(w,40),label='40')
    print("position")
    print(x.shape,y.shape)
    plt.figure()
    ax = plt.axes(projection='3d')
    ax.set_title("Position vs Time")
    ax.plot3D(moving_average(x,40),moving_average(y,40),np.arange(1087))
    ax.set_xlabel('x position')
    ax.set_ylabel('y position')
    ax.set_zlabel('Frame number')

    plt.show()

if __name__ == '__main__':
    main()
```

## D Appendix D:

```
#Author: Nicholas Rabow
#Description: con_comps.pyx is cython file used to optimize connected components
#algorithm speed increases are to be expected by doing this. static typing and c
#functions are to be wrapped and performed in a multithreaded environment. This
#algorithm was attempted to be written in python first, but the implementation
#proved to slow. Research was done on how to improve the speed.
#See Wu et. al and Fiorio et. al. for algorithms implemented.
#Date: 12/9/2021
#Version: 1.1
```

```
#libraries
import numpy as np
from warnings import warn
cimport numpy as cnp
cnp.import_array()

#global variables
ctypedef cnp.intp_t DTYPE_t
DTYPE = np.intp
cdef DTYPE_t BG_NODE_NULL = -999 #node does not exist, used as placeholder

#objects
cdef struct s_shpinfo

#constructors
ctypedef s_shpinfo shape_info
ctypedef size_t (* fun_ravel)(size_t, size_t, size_t, shape_info *) nogil


######################
#BACKGROUND FUNCTIONS#
######################

# struct used concerning background in one place
ctypedef struct bginfo:
    DTYPE_t bg_val #The value in the image that identifies the background
    DTYPE_t bg_node #Node used to keep track of background
    DTYPE_t bg_label # Identification of the background in the label image

#Name: get_bginfo()
#Parameters: [bg_val]->value of background, *ret->used to update bginfo object
#Description: Update bginfo object with important information for PROCESSING
#Returns: updated bginfo object
cdef void get_bginfo(bg_val, bginfo *ret) except *:
    if bg_val is None:
        ret.bg_val = 0 #assume background value is 0
    else:
        ret.bg_val = bg_val
    ret.bg_node = BG_NODE_NULL  # The node -999 does not exist
    ret.bg_label = 0 #label 0 is the background

#Name: scanBG
#Parameters:[*data_p]->ptr to data_p array,[*forest_p]->ptr to forest_p array,
#           [*shapeinfo]->ptr to shapeinfor object, [*bg]->ptr to bginfo object
#Description: THIS IS A MUTEX LOCKED FUNCTION. All background pixels are dealt
#             with in this function. The purpose is to reduce unecessary backgroun
#              scans.
#            This function updates forest_p and bg parameters inplace
cdef void scanBG((DTYPE_t *data_p, DTYPE_t *forest_p,
                  shape_info *shapeinfo,bginfo *bg)) nogil:
    cdef DTYPE_t i, bgval = bg.bg_val, firstbg = shapeinfo.numels #Variable definit
```

```
    for i in range(shapeinfo.numels): #find all background pixels
        if data_p[i] == bgval:
            firstbg = i
            bg.bg_node = firstbg
            break

      #assign all background pixels to same label
    for i in range(firstbg, shapeinfo.numels):
        if data_p[i] == bgval:
            forest_p[i] = firstbg

#########################
#INPUT IMAGE INFORMATION#
#########################
# A pixel has neighbors that have already been scanned.
# In the paper, the pixel is denoted by E and its neighbors:
# in my code, E is shown with 0 and the neighbors(0-14):
# o_54 represents offset of 5 from 4
# used in get_shape_info function
cdef enum:
    o_54,o_51, o_52, o_53, o_57, o_59, o_510, o_511, o_513,o_COUNT

# Structure for centralized access to shape data
# Contains information related to the shape of the input array
cdef struct s_shpinfo:
    DTYPE_t x
    DTYPE_t y
    DTYPE_t numels #number of elements
    DTYPE_t ndim #dimensions of input array
    DTYPE_t off[o_COUNT] #offsets between elements
    fun_ravel ravel_index #fuction pointer to recalculate index as needed


#Name: get_shape_info
#Parameters: [f]->input image, [*res]->ptr to shape_info object
#Description: Precalculates all needed information about the input image shape
#             and stores it in shape_info object
cdef void get_shape_info(f_shape, shape_info *res) except *:
    res.x = f_shape[1]
    res.y = f_shape[0]
    res.ravel_index = res.x + res.y * res.x
    res.numels = res.x * res.y

    #offsets
    res.off[o_54] = -1
    res.off[o_51] = res.ravel_index(-1, -1, 0, res)
    res.off[o_52] = res.off[o_51] + 1
    res.off[o_53] = res.off[o_52] + 1

#################
#TREE OPERATIONS#
#################
# Tree operations implemented by an array as described in Wu et al.
# The term "forest" is used to indicate an array that stores one or more trees
```

```
# From paper:
# Consider a following tree:
#
# 5 ----> 3 ----> 2 ----> 1 <---- 6 <---- 7
#                 |               |
#          4 >----/               \----< 8 <---- 9
#
# The vertices are a unique number, so the tree can be represented by an
# array where a the tuple (index, array[index]) represents an edge,
# so for our example, array[2] == 1, array[7] == 6 and array[1] == 1, because
# 1 is the root.
# one array can hold more than one tree as long as their
# indices are different. It is the case in this algorithm, so for that reason
# the array is referred to as the "forest" = multiple trees next to each
# other.
#
# In this algorithm, there are as many indices as there are elements in the
# array to label and array[x] == x for all x. As the labelling progresses,
# equivalence between so-called provisional (i.e. not final) labels is
# discovered and trees begin to surface.
# When we found out that label 5 and 3 are the same, we assign array[5] = 3.

#Name:join_trees_wrapper
#Parameters: [*data_p]->ptr to image information,[*forest_p]->ptr to forest,
#            [rindex]->ravel index,[idxdiff]->offset index
#Description:Calls join trees function if necessary operation.
cdef inline void join_trees_wrapper(DTYPE_t *data_p, DTYPE_t *forest_p,
                                    DTYPE_t rindex, DTYPE_t idxdiff) nogil:
    if data_p[rindex] == data_p[rindex + idxdiff]:
        join_trees(forest_p, rindex, rindex + idxdiff)

#Name:find_root
#Parameters: [*forest]->ptr to forest, [n]->node
#Description: Find the root of node n.
cdef DTYPE_t find_root(DTYPE_t *forest, DTYPE_t n) nogil:
    cdef DTYPE_t root = n
    while (forest[root] < root):
        root = forest[root]
    return root

#Name: set_root
#Parameters: [*forest]->ptr to the forest, [n]->node, [root]t]
#Description: Sets all nodes on a path to point to new_root. Will eventually set
# all tree nodes to point to the real root.
cdef inline void set_root(DTYPE_t *forest, DTYPE_t n, DTYPE_t root) nogil:
    cdef DTYPE_t j
    while (forest[n] < n):
        j = forest[n]
        forest[n] = root
        n = j
    forest[n] = root

#Name: join_trees
#Parameters: [*forest]->ptr to forest, [n]->node,[m]->root
```

```
#Description: Join two trees containing nodes n and m.
cdef inline void join_trees(DTYPE_t *forest, DTYPE_t n, DTYPE_t m) nogil:
    cdef DTYPE_t root
    cdef DTYPE_t root_m

    if (n != m):
        root = find_root(forest, n)
        root_m = find_root(forest, m)

        if (root > root_m):
            root = root_m

        set_root(forest, n, root)
        set_root(forest, m, root)
```

```
# Flatten arrays are used to increase performance. Lookup is acheived by using
# precalculated offsets. Always starting at 5 and using this offset 'rindex' can
# be calculated which is the index of the pixel in the original image. offsets
# are located in shapeinfo object
#Name: scan2D
#Parameters: [*data_p]->ptr to input image pixels, [*forest]->ptr to forest,
#            [*shapeinfo]->ptr to shapeinfo obj, [*bg]->ptr to background info obj
#Description: Peforms forward scan on 2D array
cdef void scan2D(DTYPE_t *data_p, DTYPE_t *forest_p, shape_info *shapeinfo,
                 bginfo *bg) nogil:
    if shapeinfo.numels == 0:
        return
    cdef DTYPE_t x, y, rindex, bgval = bg.bg_val #store needed information
    cdef DTYPE_t *off = shapeinfo.off #array of offset values

    # Handle the first row
    for x in range(1, shapeinfo.x):
        rindex += 1
        if data_p[rindex] == bgval: # Nothing to do if we are background
            continue

        join_trees_wrapper(data_p, forest_p, rindex, off[o_54])
    for y in range(1, shapeinfo.y):
        # BEGINNING of x = 0
        rindex = shapeinfo.ravel_index(0, y, 0, shapeinfo)
        # Handle the first column
        if data_p[rindex] != bgval:
            # Nothing to do if we are background
            join_trees_wrapper(data_p, forest_p, rindex, off[o_52])
        # END of x = 0

        for x in range(1, shapeinfo.x - 1):
```

```
            # We have just moved to another column (of the same row)
            # so we increment the raveled index. It will be reset when we get
            # to another row, so we don't have to worry about altering it here.
            rindex += 1
            if data_p[rindex] == bgval: # Nothing to do if we are background
                continue

            join_trees_wrapper(data_p, forest_p, rindex, off[o_52])
            join_trees_wrapper(data_p, forest_p, rindex, off[o_54])
            join_trees_wrapper(data_p, forest_p, rindex, off[o_51])
            join_trees_wrapper(data_p, forest_p, rindex, off[o_53])

        # Finally, the last column
        # BEGINNING of x = max
        rindex += 1
        if data_p[rindex] != bgval: # Nothing to do if we are background

            join_trees_wrapper(data_p, forest_p, rindex, off[o_52])
            join_trees_wrapper(data_p, forest_p, rindex, off[o_54])
            join_trees_wrapper(data_p, forest_p, rindex, off[o_51])
        # END of x = max


######
#MAIN#
######
#Name: label_cython
#Parameters: [f]->input image
#Description: Cythonized version of label function from my original
#             implementation. Takes in image [f] and finds connected components
#              and labels them accordingly. This will be used to
#             find ball in images. Algorithm is describe in Fiorio et al.
#Returns: label image and number of labels
def label_cython(f, bg):
    #IMAGE INFORMATION
    f, swaps = reshape_array(f)
    shape = f.shape

    #OBJECT AND VARIABLE DECLARATION
    cdef cnp.ndarray[DTYPE_t, ndim=1] forest #disjoint union of trees
    out = np.array(f, order='C', dtype=DTYPE) #row major order
    forest = np.arange(data.size, dtype=DTYPE)

    cdef DTYPE_t *forest_p = <DTYPE_t*>forest.data #pointer to forest var
    cdef DTYPE_t *out_p = <DTYPE_t*>cnp.PyArray_DATA(out) #point to data

    cdef shape_info shapeinfo #shape_info structure used in algorithm
    cdef bginfo bg #background info structure used in algorithm

    get_shape_info(shape, &shapeinfo) #get input image info and stores
    get_bginfo(bg, &bg) #get background infor and store in [bg]

    # LABEL OUTPUT
    cdef DTYPE_t count #number of labeled components
```

```
    with nogil: #Global interpreter lock, mutex like funtionality
        scanBG(data_p, forest_p, &shapeinfo, &bg) #perform background scan
        scan2D(data_p, forest_p, &shapeinfo, &bg)
        count = resolve_labels(data_p, forest_p, &shapeinfo, &bg)

    return data, count

#Name: resolve_labels
#Parameters: [*data_p]->ptr to data_p,[*forest_p]->ptr to forest_p,
#            [*shapeinfo]->ptr to shapeinfo object, [*bg]->ptr to background obj
#Description: THIS IS MUTEX LOCKED FUNCTION
#             Second pass of algorithm to resolve any connected components that
#             are two different labels. Final labels are assigned and number of
#             labels are counted
cdef DTYPE_t resolve_labels(DTYPE_t *data_p, DTYPE_t *forest_p,
                            shape_info *shapeinfo, bginfo *bg) nogil:

    cdef DTYPE_t counter = 1, i #variable instantiation

    for i in range(shapeinfo.numels): #search every pixel
        if i == forest_p[i]: #root with new information
            if i == bg.bg_node: #root is background
                data_p[i] = bg.bg_label #assign to background
            else:
                data_p[i] = counter #otherwise assign to current label
                counter += 1
        else:
            data_p[i] = data_p[forest_p[i]] #pixel without new label information
    return counter - 1
```