Recording and Analyzing Moving Objects

This week's lab is focused on the basic setup for recording the motion of moving objects and the method for importing videos into Python and using the computer to extract physical quantities, such as the position of the object. We will begin by working with a video that is provided, and learn how to import this video into Python and to do the image processing involved in tracking the object. The technique that we will be using to analyze the frame-by-frame motion of the object will be used extensively during the rest of the course in order to extract quantitative data, such as velocities and accelerations using the position of the object at each time point. The second part of the is lab will be for you to take your own recording of an object moving and to use the video to determine the position of the object as a function of time. An important aspect of this project is in setting up how the video is taken so that it is easy to determine where the object is and how far it has moved.

Goals of this lab:

- Import videos into Python using OpenCV
- Use Numpy to manipulate the images and extract the center of the object
- Use Matplotlib to make plots of individual frames from the video and graphs of the position of the object as a function of time
- The technique that we will be using to analyze frame-by-frame motion of the object will be used extensively during the rest of the course in order to extract quantitative data, such as velocities and accelerations using the position of the object at each time point

Lab equipment:

Personal Computer or Laptop

Phone with Camera or PC with Webcam

In order to record the object for this lab (and subsequent ones), you will need a camera that is able to record digitally at at least 30 frames per second (fps).

Can of Food

You will need one can of food (e.g., beans, soup, etc.) to roll along a surface.

Wood plank

You will need a wood board or a piece of drywall to roll the can along.

Ruler

A ruler or tape measure will be used to measure distance along the board.

Blue Painter's Tape

Tape will be used to make the color of the top of the can blue

Lab Procedures

I. Analyzing a video of a rolling can

IA. Opening the video

In the provided online materials, you will find a video entitled "Rolling Can.mp4". Download this video into a folder titled Lab 2 on the computer that you have Anaconda Python loaded on. Open and watch the video so that you know what it looks like.

Open Spyder and navigate to the Lab 2 folder.

We are going to start by learning some of the commands and techniques that we will use to analyze this video. We will perform these initial tasks using the Console window in Spyder. We will then take the tools that we learn here and write a code that we can use to analyze the whole video.

We will be using three Python packages: Numpy, Matplotlib, and OpenCV. Numpy is a package that has a programs that carry out a large number of different mathematical operations, and it also has some nice data structures. Matplotlib is a plotting package which is good for visualizing data and making graphs, etc. OpenCV is a package that has a number of useful image processing programs in it. In order to use these packages, we must import them into memory. From the console prompt, type

```
In [1]: import numpy as np
In [2]: import cv2
In [3]: import matplotlib.pyplot as plt
```

You will note in the first line that we are renaming numpy as np. This is a very common choice that many programmers use. OpenCV is known as cv2 in Python, which is imported in the second line. Finally, in the third line, only part of the matplotlib library is needed for this project. This is a part called pyplot, which contains a number of plotting and graphing routines. It is also very common to rename this plt.

Now we are ready to import the video. The first thing that we will do is create a structure that opens the video file and gives us access to the information stored in it. This is done with the function VideoCapture in OpenCV. We will call the structure VidObj and we define it like this:

```
In [4]: VidObj = cv2.VideoCapture('Rolling Can.mp4')
```

This command not only opens the file 'Rolling Can.mp4', but it also contains some information about the video itself. For example, if we wanted to find the number of frames per second that this video was recorded at, we can type

```
In [5]: VidObj.get(5)
```

Once you have typed this and hit return, you should get 30.0 printed to the console window, which means that the video was recorded at 30 frames per second.

The number 5 that is fed to the get command is what requests the frames per second that the video was recorded at. Other integers return other information. For example, a 3 would return the number of pixels in the width of the images in the video. A 4 returns the number of pixels in the height of the images.

IB. Accessing and viewing a frame from the video

We will now open the first frame of the video. We first need to read the frame into memory. We will store the frame as a variable called `frame`. To read in the first frame of the movie, we just need to type

```
In [6]: ret,frame = VidObj.read()
```

The read command outputs two things. The first one, which we have defined as `ret`, is a flag that tells whether or not the computer was able to read the image. The second output is the data for the image, which is stored as an array of numbers. This is what we are calling `frame`.

Type `frame.shape` at the next console prompt. You should get an output of `(1080, 1080, 3)`. This tells you about the structure of `frame`. Digital images are stored as an array of pixels. The image that we see is created by breaking the image up into a number of boxes (pixels). Each box is assigned a color. The image in this video is composed of 1080 boxes (pixels) in the horizontal direction and 1080 boxes (pixels) in the vertical direction. You will notice that the first number in the shape of `frame` is 1080 (the height) and the second number is 1080 (the width). Colors in images can be stored a few different ways. One of the most common ways is to break the color up into its primary components. In dealing with light, the primary colors are Red (R), green (G), and blue (B). Therefore, we can uniquely define a color by defining how much red, green, and blue are in it by assigning a number for the amount of red, green and blue. That is, it takes three numbers to define the color of each pixel, which is why the last number in `frame.shape` is a `3`. Storing colors in terms of their red, green, and blue content is known as an RGB color space.

The data for the first image of the video is stored in `frame`. `frame` is an array of numbers, and that array has three dimensions. The first dimension labels the vertical components (pixels) of the image, the second dimension is the horizontal components (pixels), and the third dimension gives the color of the pixels.

To access information that is stored in this type of array (which is actually a numpy array), you use square brackets after the variable name: `frame[x,y,z]` gives the x,y,z entry in the array. If we want to know what the RGB value for the pixel located 40 down from the top and 28 over from the left side, then we can type

```
frame[39,27,:]
```

(You should get an output of `array([69, 88, 99], dtype=uint8)`). This tells you that the color is stored as an array with values 69, 88, and 99. OpenCV reads in the RGB values in the order blue, green, red. Therefore, the amount of blue in this pixel is 69, the amount of green is 88, and the amount of red is 99. The final thing tells you that the number are stored as a *data type* of unit8. Many digital cameras and images store colors on an integer scale that goes from 0 to 255, with 0 meaning that there is none of that color, and 255 being the maximum amount of a given color. Hence, if the

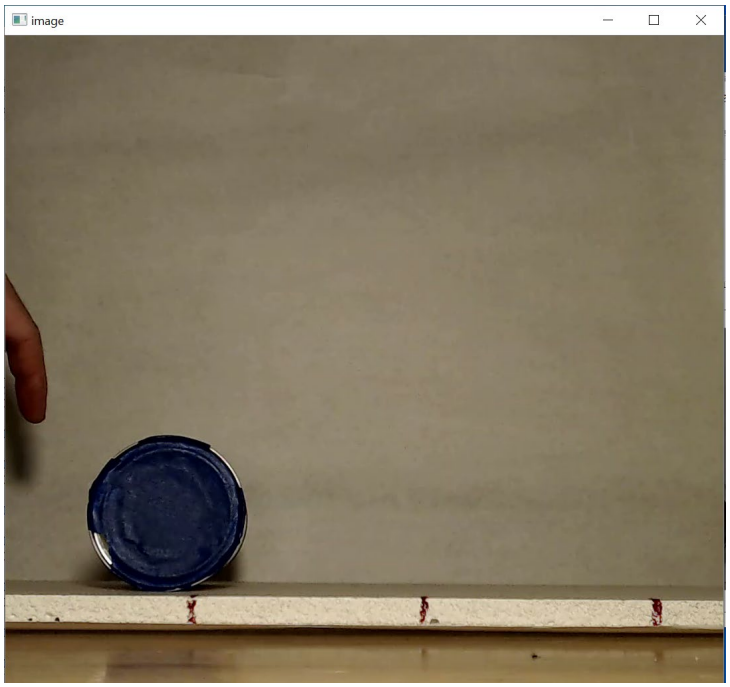
array was `[0, 0, 255]`, it would mean that the color was bright red and no blue or green were present, and so on. The colon in the last entry tells Python to give all the information along that dimension (which in this case is the color dimension).

One more thing to note is that we used 39 as the first entry into frame and 27 as the second, in order to access the pixel 40 down from the top and 28 over from the left side. Python numbers arrays starting with 0. It also numbers them starting from the top of the array and having numbers increase as it goes down, or starting from the left with numbers increasing to the right. Therefore, `frame[0,0,0]` would be the amount of blue in top-left pixel in the image.

Now let's actually look at the image we have stored in `frame`. To view the first frame of the video, type

```
cv2.imshow('image',frame)
```

A new window should have popped up after you hit return, which should look like this:



You will notice that we used two inputs into the imshow command. The first (`'image'`) tells imshow that we want to open a new image window. The second (`frame`) tells it to plot the image data that is stored in `frame` in that window.

Now let's do a couple manipulations of the image. Let's look only at the red *channel* of the image. We will first store a copy of the data in a new variable that we will call `red`:

```
red = frame
```

Then we will set all the blue and green colors in red to zero:

```
red[:,:,0] = 0
```

sets the blue value of all the pixels to zero. That is, by using a colon in the first entry of red, we are accessing all the vertical pixels at a given horizontal location. Because we are also using a colon in the second entry, that will access all the horizontal locations. Therefore, we have accessed all the pixels, and set the first color entry (blue) to zero. Likewise,

```
red[:,:,1] = 0
```

sets all the green components to zero. Now view the new image by typing

```
cv2.imshow('image',red)
```

which should look like

IC. Extracting the can and line spacing from the image

From watching the video, you should have noticed that the background in the image is white, and the board that the can is sitting on has red lines drawn on it (these are drawn at intervals of 4 inches). In addition, the can has blue tape on the top of it. These color choices were made to make extracting the can and the lines from the video easy. The lines are put in the image so that distances in the image can be determined. We now need to extract the can and the lines from the image.

Pure white in an RGB image has full intensity of all three colors. Therefore, the color array for white in unit8 format is `[ 255, 255, 255 ]`. Gray colors all have the same amount of red, green, and blue. As the intensities of these colors decreases, the gray becomes darker and darker. In the image below, the gray on the left has RGB code `[ 180, 180, 180 ]`, the middle gray is `[ 90, 90, 90 ]`, and the right square is black = `[ 0, 0, 0 ]`.

A grayscale image can be made by averaging the red, green, and blue intensities in each pixel and defining that as the gray intensity value. We can use numpy to compute the average of an array over a given dimension, using the function `mean`. We will store the grayscale intensity in each pixel in an array called `gray`:

```
gray = np.mean(frame,axis=2,dtype=float)
```

The axis is the dimension along which we want to take the average. Frame is currently in uint8, which is an integer format. We will define the grayscale intensity as a `float`, which will allow us to do mathematical operations on it. You can view the grayscale image using,

```
cv2.imshow('image',gray.astype('uint8'))
```

The `astype` command converts `gray` back to `unit8` so that `imshow` plots it correctly.

Because this gray image is the average of the red, green, and blue intensities in the image, a red object will have a larger red intensity than this average value. We can compare the red intensity to the gray intensity in order to make red objects brighter. Let's define the difference between the red and gray images:

```
RminusG = red[:,:,2] - gray
```

Because we have defined the array `RminusG` using subtraction, it can have positive or negative values. When we plot this using `imshow`, the negative values will be shown as pure white:

```
cv2.imshow('image',RminusG.astype('uint8'))
```

Notice that the lines (and the finger) are brighter than the background. The can top looks very bright; this is because it's values are negative (it is blue not red). The combination of these two things means that we can extract both the lines and the can using the array `RminusG`. The can can be identified because its values are negative, and the lines can be extracted because they are brighter than the background. Therefore, we will select these features by defining *Threshold* values. As was mentioned, the can is negative, therefore if we set a Threshold value of 0 and look for locations in RminusG that are less than 0, we should select the can. We will do this selection by a process called *segmentation*. Pixels less than 0 will be kept, and pixels brighter than that will be excluded. To do this, we will set the excluded pixels to 0 (also known as *False*) and the negative pixels to 1 (or *True*).

We will begin by defining this Threshold value to be 0:

```
Thresh = 0
```

Then, to segment the image, we just set any pixels less than `Thresh` to `True`; all the remaining pixels are then `False`, by default. We will store this True/False information in an array called `Mask` (it *masks out* the pixels we don't care about). There is a simple syntax for doing this:

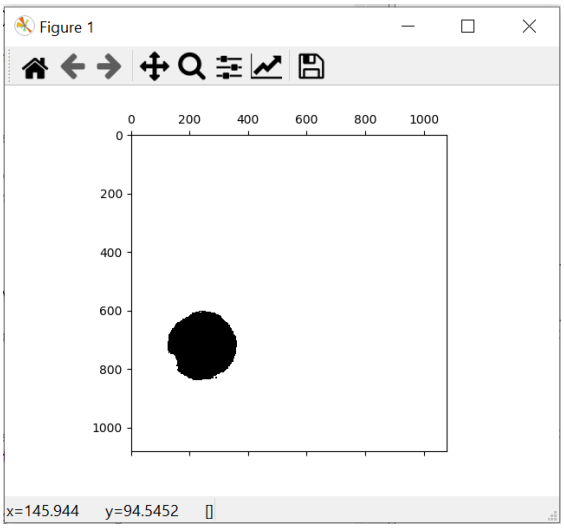
```
Mask = ( RminusG < 0)
```

This statement says, set `Mask` equal to `True` wherever `RminusG` is less than `Thresh`.

`Mask` is a binary array: The entries are either `True` or `False` (or 0 or 1, equivalently). To view a binary array, we would like to see which pixels are `True`. Matplotlib has a nice visualization for this called spy. To view the `Mask`, type

```
plt.spy(Mask)
```

A new window should open with a figure in it:

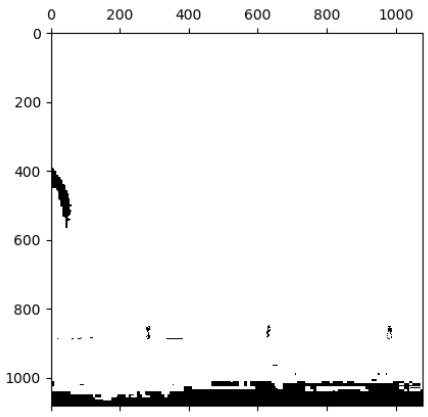


We can use a similar technique to isolate the lines in order to measure the distance between them. For this, we need to select pixels that are *brighter* than a certain Threshold. To find a good threshold value, let's find out what the average intensity in gray is:

```
AvgInt = np.mean(RminusG)
```

Now call `AvgInt` to find out what the value is (It should be 17.037…). Let's choose something a little brighter than this as the Threshold:

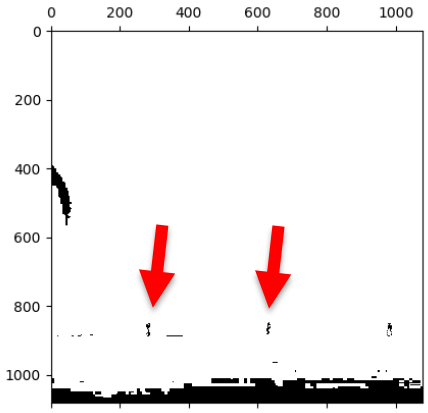
```
Thresh2 = 35
```



While there are a lot of spurious pixels included in this mask, the locations of the red lines are easily identifiable. We just need to be able to see them clearly, so that we can measure the distance between them. We will manually extract the distance between the lines that are drawn on the board. Matplotlib has a function `ginput(n)` that allows the user to click on `n` points in a plot, and the function then returns the pixel values of those `n` points. We only need to measure the distance between two of the lines in the plot of the Mask that we just made. At the console prompt, type

```
points = plt.ginput(2)
```

which will allow us to click on two neighboring lines in the image. After you type the preceding statement, go back to the figure. You can right click on the image and it will store points. Click on the two lines shown below:



After you click twice on the image, the points will be stored in memory. You can see the values of the points by typing `points` at the command prompt. The values should be approximately:

```
[(282.0612362321222, 863.9024330100278),
 (631.2310537563701, 863.9024330100278)]
```

Note, the values you get could be somewhat different depending on where on each line you clicked. The first entry shown above (282.06…) is the horizontal coordinate, x_1, of the first point that was clicked on, and the 863.90… is the vertical coordinate, y_1, of the same point. The second line contains the coordinates of the second point that was selected (x_2, y_2).

Each of these lines was drawn 4 inches apart. Therefore, 4 inches corresponds with the horizontal displacement of the two points we selected. The number of inches per pixel is then

$$\text{inches per pixel} = \frac{4 \text{ in.}}{x_2 - x_1}$$

There are 2.54 cm per inch, so if we want to work in terms of cm (which is what we want to do):

$$\text{cm per pixel} = \frac{10.16 \text{ cm}}{x_2 - x_1}$$

For the values that are listed above, there are

$$\text{cm per pixel} = \frac{10.16\text{ cm}}{631-282} = 0.0291\frac{\text{cm}}{\text{pixel}}$$

Using this factor, we can convert from pixels into actual distance that the can moves. We will call this factor `Pix2cm`, or the pixel to cm conversion factor.

The last thing that we need to do to be able to track the can in the video, is to determine the location of the center of the can. While the mask we made for the can is quite good, and seems to have done a good job at only identifying pixels on or very near the can, it is best to make sure that we truly isolate the pixels associated with the can in the image. Since the can is the largest feature in the image, if we identify which pixels are included in this largest group of `True` pixels in the `Mask`, then we can use the centroid of this group of pixels to define the center of the can lid. In other words, we need to determine which of the `True` pixels in the `Mask` are connected to each other. OpenCV has a function that will do this for us called `connectedComponentsWithStats`. The Stats part includes information about the regions that are found, such as the size of the region and the center of the region. We will store the information about this connectivity in a variable that we will call `Regions`:

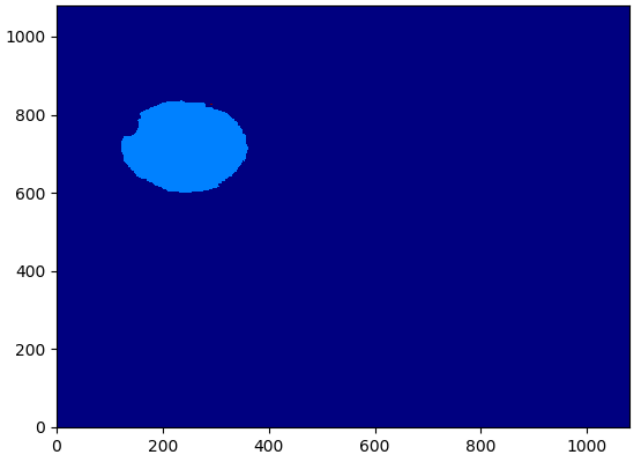
```
Regions = cv2.connectedComponentsWithStats(Mask.astype('uint8'))
```

The `connectedComponentsWithStats` function needs `Mask` to be treated as a uint8 data type. `Regions` will be a 4 component structure:

1. The first component, `Regions[0]`, is an integer that tells how many different unconnected regions were found in `Mask`. If you type `Regions[0]` at the console prompt, you should get 5 as an output. This number includes the background, so we see that there are 4 non-background regions that were identified.
2. The second component, `Regions[1]`, is an array that contains the same number of pixels as are in the original image. This array *labels* the different regions. Pixels that are part of the background get labeled as 0. The first connected region is labeled with a 1 in each pixel corresponding to that region, and so on. To view this Label Matrix, type

   ```
   plt.pcolormesh(Regions[1],cmap='jet')
   ```

   which should produce



The different colors show that different regions are assigned different numbers. You will also notice that the image is upside down. This is because pcolormesh draws things in an *x*-*y* coordinate system with *y* increasing upward, where the image data is numbered top-down.

3. The third component, `Regions[2]`, is an *N* x 5 array with statistics for each of the *N* regions that were found. We will store these stats in a new variable `Stats`:

   ```
   Stats = Regions[2]
   ```

   As mentioned Stats is an array of size [N,5], where N is the number of rows in the array and 5 is the number of columns. The first 4 columns give information on the leftmost pixel, the topmost pixel, the width of the region, and the height of the region, respectively. The 5th column is the area of each region. We want to select the region with the largest area, as this corresponds to the top of the can. In order to find which region has the largest area, we can use Numpy's argmax command:

   ```
   CanLabel = np.argmax(Stats[:,4])
   ```

   You should find that `CanLabel` has the value 0. Recall that the background is labeled 0, so what we have found is that the background has the largest area. We want the next largest area. There are a few ways to handle this but one easy way would be to set the area of the background equal to zero and then re-run this command:

   ```
   Stats[0,4] = 0
   CanLabel = np.argmax(Stats[:,4])
   ```

   You should now find that `CanLabel` has the value 1.

4. The last component, `Regions[3]`, gives the centroid coordinates for each region. We will use this to define the location of the center of the top of the can.

We will now create a new masked array that only includes the top of the can. We first copy `Mask` into our new mask, which we will call `CanMask`:

```
CanMask = Mask
```

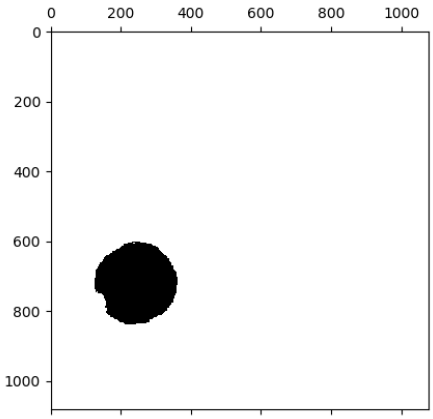
Then, we set all the entries that are not labeled with the same value as `CanLabel` to be equal to zero:

```
CanMask[Regions[1]!=CanLabel] = 0
```

The "`!=`" means "not equal to" in Python. Therefore, the statement in the brackets says "Find the locations where the array `Regions[1]` are not equal to the value `CanLabel`." By putting this into the bracket for `CanMask`, we are saying "set the values of the array `CanMask` equal to zero at locations that correspond with where `Regions[1]` is not equal to `CanLabel`."

Look to see what the CanMask looks like:

```
plt.spy(CanMask)
```



Since `Regions[3]` contains the centroids of all the regions, we can define the location of the centroid of the can top:

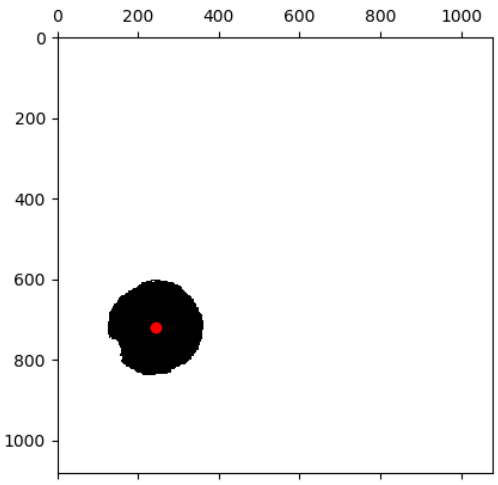
```
Center = Regions[3][CanLabel]
```

Note how we had to call the component of `Regions` with the first square bracket and the entry number we wanted with a second square bracket.

To make sure that everything has worked correctly, we will use the Matplotlib function `plot` to put a red circle at the location of the centroid:

```
plt.plot(Center[0],Center[1],'or')
```

The first entry of `plot` is the *x* coordinate, the second entry is the *y* coordinate, and the third entry tells what kind of point to put at that (*x*,*y*) position. Since we want a red (r) circle (o), we use `'or'`. You should now see:



II. Writing a program to analyze the full video

You now have all the tools that you need to go into each frame from the video and extract the center of the can. Here we will put together a piece of code that allows a user to import their movie, threshold it, find the distance between the lines in the image, and then track the motion of the center of the can frame-by-frame.

1. Open a new file in the Spyder text editor. Save this file as AnalyzeVideo.py.
2. We will start the file with a comment telling what the functions in the file do:

```
"""
This file contains two functions. The first is ReadFrame which
reads a specific frame from a video file.  The second is called
TrackMotion. This function tracks the largest object in the image
that is darker than the background. The TrackMotion function also
assumes that there are scale markers included in the image that the
user will click on during the first frame to define the pixel to cm
ratio.  The distance (Dist) between these markers should be input in cm.
"""
```

As is mentioned in this comment, we are going to define two functions. We want to be able to open a specific frame from the movie. We will put together a short function to do this for us. But, first, we need to import the packages that we are going to use. After the comment add:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

3. We can now define our function to open a specific frame from the movie. This function will take a VideoCapture object and open the FrameNum image in the video file:

```
def ReadFrame(VidObj,FrameNum):

    VidObj.set(1,FrameNum)
```

```
    ret,frame = VidObj.read()

    return frame
```

As always, pay attention to the tabs. The only new command is `set`. This works just like `get` did, except that we are *setting* one of the values in `VidObj`, instead of *getting* it. The `1` tells `set` that we want to set the frame number that should be opened next. The output of this function is the `frame` array.

4. We can now make the TrackMotion function. This function will be read in a user-defined File named Filename, and will use a threshold value Thresh to segment the images in that file. The user will also supply the distance between the markers in the image that are used to set the pixel to cm scale. This function will be broken into 3 parts. The first part initializes some of the information that will be used. The second part will process the first image and determine the pixel to cm scale. Finally, the third part will loop through the images in the video and track the position of the can.

We will start this function, similar to the flow of commands that were used in the last section. We are going to request 4 inputs from the user: the filename, the two Threshold values, and the distance between the lines that are drawn on the board in cm.

```
def TrackMotion(Filename,Thresh,Thresh2,Dist):

    VidObj = cv2.VideoCapture(Filename)

    # determine the size of the video images and the number of frames
    Width = VidObj.get(3)
    Height = VidObj.get(4)
    FPS = VidObj.get(5)
    NumFrames = int(VidObj.get(7)) # determines number of frames in video

    # setup arrays to store the coordinates
    # of the center of mass at each time point
    Xcm = np.zeros((NumFrames,1))
    Ycm = np.zeros((NumFrames,1))
```

These two commands are initializing two arrays to store the center of mass coordinates for the can. The *x*-coordinate will be stored in an array called `Xcm` that has `NumFrames` entries. The np.zeros function is making an array of zeros that has size `NumFrames` × `1`. The same thing is being done for the y coordinates. We also want to create an array to store the time point for each frame:

```
    # define the time of each frame
    Time = np.array([i for i in range(NumFrames)],dtype='float')/FPS
```

This command is defining a list of the time points for each frame. `Time` is being defined as a Numpy array. That array is being made using a list of integers `i` the run over a `range` from 0 up to the value NumFrames-1. Note that the function `range`, when it is given a single input, makes a list of integers that start at 0 and go up to the value of the input minus one.

Next we will read in the first frame. We will convert it to float, and then threshold our image as we did before:

```
# read in first frame and convert to float

frame = ReadFrame(VidObj,0)
frame = frame.astype('float')

# compute the grayscale and red images
gray = np.mean(frame,axis=2,dtype=float)
red = frame[:,:,2]

# find values of the image created by subtracting gray from red
# that are less than Thresh or greater than Thresh2
Mask = (red - gray < Thresh)
Mask2 = ( red – gray > Thresh2 )

# have user measure the distance between lines in image

fig = plt.figure()
plt.spy(Mask2)
print('Click on two marker points in the image that are Dist apart')
points = np.array(plt.ginput(2))

# define the pixel to cm conversion scale

Pix2cm = Dist/(points[1,0] - points[0,0])
```

The one difference here is that we have defined the input points from the user as an array. This is done so that these points can be subtracted from one another.

We then use our mask to isolate the top of the can and find its center of mass:

```
# find the connected regions in the Mask

Regions = cv2.connectedComponentsWithStats(Mask.astype('uint8'))

# determine which region has the largest area
Stats = Regions[2]
Stats[0,4] = 0
CanLabel = np.argmax(Stats[:,4])

# remove unwanted regions from the Mask
CanMask = Mask
CanMask[Regions[1]!=CanLabel] = 0

# find the Center of Mass of the object
Xcm[0] = Regions[3][CanLabel,0]
Ycm[0] = Regions[3][CanLabel,1]
```

So that the user can see how the program is working, we plot `CanMask` and the center of mass as we did before:

```
# plot the mask and its center of mass
plt.clf()
plt.spy(CanMask)
plt.plot(Xcm[0],Ycm[0],'or')
plt.pause(0.1)
```

The first command is used to clear the existing figure, and the last command gives the computer an 0.1 second pause to draw the figure.

5. All that is left to do is to *loop* through each of the remaining frames and find the center of mass of the top of the can. We will use a *for loop* to do this. We want to start at the 2nd frame, find the center of the can, and then move to the next frame. If we want to loop through in this manner, we can type:

```
# loop through the remaining frames of the video
for i in range(1,NumFrames):
```

This looks like what we used before when we defined `Time`. The difference is that we have fed the `range` function two inputs. The first input now tells `range` what integer it should start with, and the second input tells it to stop at `NumFrames-1`.

Now we just redo the calculations to find the can and its center of mass. In order for these commands to be part of the `for` loop, we need to tab them in with respect to the `for` command line:

```
    # read in first frame and convert to float

    frame = ReadFrame(VidObj,i)
    frame = frame.astype('float')

    # compute the grayscale image
    gray = np.mean(frame,axis=2,dtype=float)
    red = frame[:,:,2]

    # find values of the grayscale image greater than Thresh
    Mask = (red - gray > Thresh)

    # find the connected regions in the Mask

    Regions = cv2.connectedComponentsWithStats(Mask.astype('uint8'))

    # determine which region has the largest area
    Stats = Regions[2]
    Stats[0,4] = 0
    CanLabel = np.argmax(Stats[:,4])

    # remove unwanted regions from the Mask
```

```
        CanMask = Mask
        CanMask[Regions[1]!=CanLabel] = 0

        # find the Center of Mass of the object
        Xcm[i] = Regions[3][CanLabel,0]
        Ycm[i] = Regions[3][CanLabel,1]

        # plot the mask and its center of mass
        plt.clf()
        plt.spy(CanMask)
        plt.plot(Xcm[i],Ycm[i],'or')
        plt.pause(0.1)
```

The only things new here are that `ReadFrame`, `Xcm` and `Ycm` are called with `i` as an index instead of `0`, and we do not have to define `Mask2` again.

Now we can do the final aspects of the program. We first will redefine `Xcm` and `Ycm` so that they are in centimeters not pixels. We will then make a graph of the *x* position of the center of mass of the can as a function of time. Before the program is completely done, `VidObj` needs to be closed. Finally, the program should return `Xcm`, `Ycm`, and `Time` as outputs. Since we are no longer in the `for` loop, we exit the loop by using one less tab than what was just being used.

To define `Xcm` and `Ycm` in centimeters:

```
        # convert Xcm and Ycm to centimeters
        Xcm = Pix2cm*Xcm
        Ycm = Pix2cm*Ycm
```

Then we open a new figure to put our plot in:

```
        # plot Xcm as a function of time
        plt.figure()
        plt.plot(Time,Xcm)
```

We can add text to the axes of this plot to label what each axis is:

```
        plt.xlabel('Time (s)',fontname='Arial',fontsize=16)
        plt.ylabel('Distance (cm)',fontname='Arial',fontsize=16)
```

close `VidObj:`

```
        VidObj.release()
```

and return the outputs:

```
        return Xcm,Ycm,Time
```

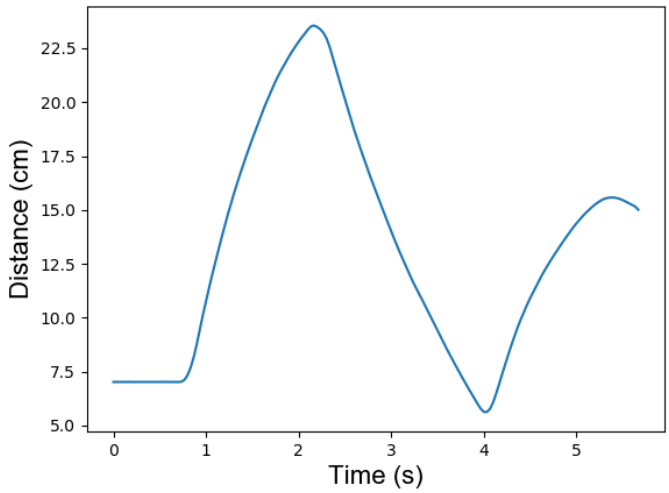
The full code will be given below. To run your code, save it, and then go to the console prompt and type:

```
from AnalyzeVideo import TrackMotion
```

Then type,

```
Xcm,Ycm,Time = TrackMotion('Rolling Can.mp4',0,35,10.16)
```

The code should ask you to identify click on the two line points in the first frame. It should then start creating a video of the masked region corresponding to the top of the can with a red circle drawn at the center. After it has run through all the frames, it should output a plot of the distance vs. time, which should look like this:



You should save this figure by clicking on the save icon at the top of the figure frame. It will be submitted with the other materials for this project.

AnalyzeVideo.py

```
"""
This file contains two functions. The first is ReadFrame which
reads a specific frame from a video file.  The second is called
TrackMotion. This function tracks the largest object in the image
that is darker than the background. The TrackMotion function also
assumes that there are scale markers included in the image that the
user will click on during the first frame to define the pixel to cm
ratio.  The distance (Dist) between these markers should be input in cm.
"""

import numpy as np
import cv2
import matplotlib.pyplot as plt

def ReadFrame(VidObj,FrameNum):

    VidObj.set(1,FrameNum)

    ret,frame = VidObj.read()

    return frame

def TrackMotion(Filename,Thresh,Thresh2,Dist):

    VidObj = cv2.VideoCapture(Filename)

    # determine the size of the video images and the number of frames
    Width = VidObj.get(3)
    Height = VidObj.get(4)
    FPS = VidObj.get(5)
    NumFrames = int(VidObj.get(7)) # determines number of frames in video

    # setup arrays to store the coordinates
    # of the center of mass at each time point
    Xcm = np.zeros((NumFrames,1))
    Ycm = np.zeros((NumFrames,1))

    # define the time of each frame
    Time = np.array([i for i in range(NumFrames)],dtype='float')/FPS

    # read in first frame and convert to float

    frame = ReadFrame(VidObj,0)
    frame = frame.astype('float')

    # compute the grayscale and red images
    gray = np.mean(frame,axis=2,dtype=float)
    red = frame[:,:,2]

    # find values of the image created by subtracting gray from red
    # that are less than Thresh or greater than Thresh2
    Mask = (red - gray < Thresh)
    Mask2 = ( red – gray > Thresh2 )

    # have user measure the distance between lines in image

    fig = plt.figure()
    plt.spy(Mask2)
    print('Click on two marker points in the image that are Dist apart')
```

```
points = np.array(plt.ginput(2))

# define the pixel to cm conversion scale

Pix2cm = Dist/(points[1,0] - points[0,0])

# find the connected regions in the Mask

Regions = cv2.connectedComponentsWithStats(Mask.astype('uint8'))

# determine which region has the largest area
Stats = Regions[2]
Stats[0,4] = 0
CanLabel = np.argmax(Stats[:,4])

# remove unwanted regions from the Mask
CanMask = Mask
CanMask[Regions[1]!=CanLabel] = 0

# find the Center of Mass of the object
Xcm[0] = Regions[3][CanLabel,0]
Ycm[0] = Regions[3][CanLabel,1]

# plot the mask and its center of mass
plt.clf()
plt.spy(CanMask)
plt.plot(Xcm[0],Ycm[0],'or')
plt.pause(0.1)

    # loop through the remaining frames of the video
    for i in range(1,NumFrames):

    # read in first frame and convert to float

     frame = ReadFrame(VidObj,i)
     frame = frame.astype('float')

     # compute the grayscale image
     gray = np.mean(frame,axis=2,dtype=float)
      red = frame[:,:,2]

     # find values of the grayscale image greater than Thresh
     Mask = (red - gray > Thresh)

     # find the connected regions in the Mask

      Regions = cv2.connectedComponentsWithStats(Mask.astype('uint8'))

      # determine which region has the largest area
      Stats = Regions[2]
      Stats[0,4] = 0
      CanLabel = np.argmax(Stats[:,4])

      # remove unwanted regions from the Mask
      CanMask = Mask
      CanMask[Regions[1]!=CanLabel] = 0

      # find the Center of Mass of the object
      Xcm[i] = Regions[3][CanLabel,0]
      Ycm[i] = Regions[3][CanLabel,1]

      # plot the mask and its center of mass
```

```
 plt.clf()
 plt.spy(CanMask)
 plt.plot(Xcm[i],Ycm[i],'or')
 plt.pause(0.1)

# convert Xcm and Ycm to centimeters
Xcm = Pix2cm*Xcm
Ycm = Pix2cm*Ycm

# plot Xcm as a function of time
plt.figure()
plt.plot(Time,Xcm)
plt.xlabel('Time (s)',fontname='Arial',fontsize=16)
plt.ylabel('Distance (cm)',fontname='Arial',fontsize=16)

VidObj.release()

return Xcm,Ycm,Time
```

III. Recording you own movie to analyze

The last part of the project will be for you to record your own video and analyze it. You will need to make sure that the device that you are using is setup to record video at the appropriate speed. We want to use at least 30 frames per second (30 fps) now. Later on, it may be better to use a higher frame rate (e.g., 60 fps), but for now, we will just set the frame rate to 30 fps. Here we provide examples for how to do that on an IPhone, an Android, or for the Logitech Capture Software. If you are using a different device than these, you will need to look at the help pages or user manual to determine how to set the video rate on your device.

IPhone

1. **Go to Settings**
2. **Go to Camera**
3. **Select Record Video**
4. **Choose a resolution of at least 720p (if your device can do 1080p, that is better) and 30 fps.**

Android

1. **Launch Open Camera**
2. **Jump into "Settings → Video Frame Rate"**
3. **Select 30**

Logitech Capture Software for Webcam

1. **Open the software**
2. **On the lefthand side, choose the camera settings button** 
3. **Under FPS, choose 30**

III.A Recording your video.

As was done in the sample video, you are going to record the motion of a can rolling across a surface. You will need to use a board on which you mark out equally spaced lines. Use a ruler or tape

measure to carefully mark off lines with a red marker or pen. Make sure the lines are large enough that they will be easy to see in the video.

Using blue painter's tape, cover the top lid of a can so that it mostly covered with the blue tape.

Choose a bright location for your setup. It is preferable to have the light coming in from behind where your camera will be placed. This helps to reduce shadows and also brings out more of the color of objects. Place your board on a flat surface, either a table top or the floor, etc. Make sure the board is facing such that the lines that are drawn on it will be seen directly from the camera. If possible, hang a light colored sheet, blanket, or piece of paper in the background so that the background is as uniformly colored as possible. Avoid, a background color that might be too red or blue. The way that I set up my space for recording is shown below:

Setup your camera or phone so that it will record a region of about 12 inches along the board you are using. The camera or phone should be placed in a fixed position, not held in your hand. You can use a tripod if you have one, or you can prop the camera or phone up. Place the can on its side such that the lid that is covered in the blue tape is facing directly towards the camera. Now record a roughly 5 second video where you roll the can back and forth a number of times. Save the movie, and then load it into the Lab 2 folder on your computer.

Use Spyder to open up the first frame of your video. You will need to determine whether the same Thresholds that were used for the sample video will also work for yours. Create the red and gray images like we did previously, and subtract the gray image from the red one. Check to see whether the lines that you drew on the board stand out brighter than the background in the resulting subtracted image. Also check to see if the intensity values in the region of the can are negative.

If the can is negative and the red lines are brighter than the background, then you can find Threshold values that will let you properly segment the lines and the can. If the can and lines are not

properly adjusted, you may need to try retaking your video with either a different background or different lighting.

Once you have found proper Threshold values, run the TrackMotion program with your video to make a plot of the distance vs. time for the can in your video. Make sure that the can tracks well through all frames of the video.

Assignment

For this project, you will need to submit 4 items:

1. your VideoAnalyze.py file
2. the Distance vs. Time plot that was generated by TrackMotion for the sample movie
3. the video that you took and analyzed
4. the Distance vs. Time plot for this video