Coupled Pendulum Image Processing
Scott Blankenbaker
Edlund Lab, SUNY Cortland
Spring 2020

The coupled pendulum experiment is being performed to provide a visually intuitive backing for a complex mathematical system, with applications outside of classical mechanics. This report presents Python code used to track the locations of the pendula used in this experiment. The code requires a physical setup with yellow paint marking the center of each mass and a camera viewing the coupled pendula in front view; both dimensions of the pendula's arcs should be visible to the camera. The distance from the camera to the pendula is not critical; the code supports any number of pendula in the field of view, with good performance at five pendula.

The central piece of this analysis is identifying yellow pixels. Pixels are assigned a value based on the formula R + G – B, with R, G, and B the red, green, and blue values assigned to the pixel. This allows for an easy contrast of yellow objects without requiring highly specific lighting or background conditions. The values are compared against a user-specified threshold, which should be tested and adjusted when the experiment's environment changes. Good results have been obtained using a threshold value of 380 with the lab's artificial lighting; natural lighting should be avoided due to its less stable brightness. If more reliable yellow detection is desired, the simplest solution will be applying a black background.

Testing for this contrast value is done using the "writeBW" function, which converts one still image into an image with all yellow pixels colored white and all nonyellow pixels colored black. When setting a new yellow threshold value, at least one image from the sequence should be tested using this function; the only spots visible in the written image should be the yellow marks on each mass. This function should be called in the console, and requires only input and output locations.

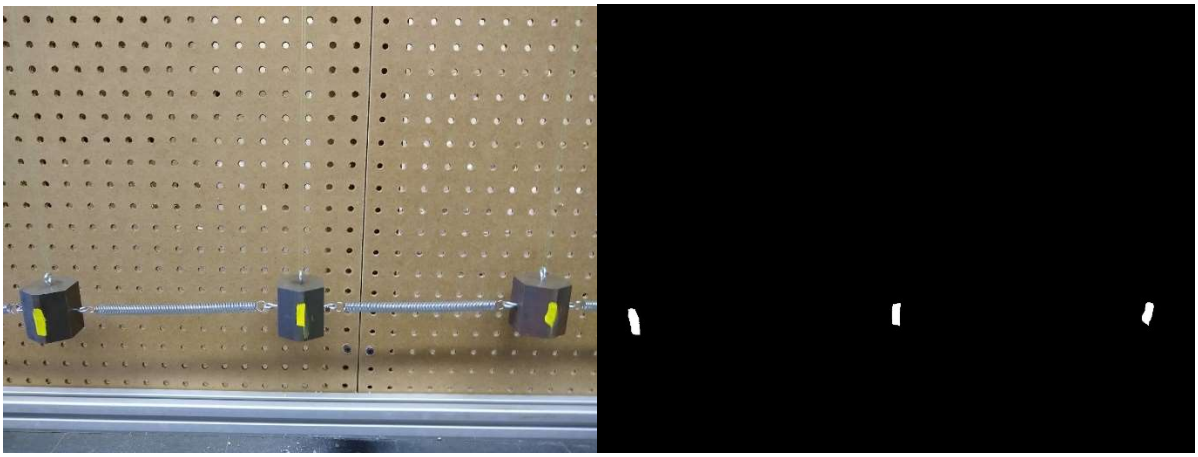

Figure 1: Comparison of base image with black-and-white yellow identification image. This is good contrast and will produce usable data.

With the appropriate threshold value set, two approaches were taken for finding the centers of masses. The first was to find the image row with the most yellow pixels and locate the start and end of each streak of yellow pixels in this row. While the pendula movement is approximately along one horizontal line, this approach is simply less reliable, as it uses too few pixels and runs into errors too easily when physical issues are present – for instance, uneven painting of the yellow markers. This approached used the “findMaxRow” and “findPositions” functions, which are included for completeness but are not used.

The second approach, including the “findPositions2” and “findMassCenter” functions, uses all yellow pixel clusters in the image, giving much more reliable results. The image – which can be restricted to below a certain height for speed, using the “minheight” parameter – is searched for a yellow pixel. The “findMassCenter” function then draws a box with width four times the set parameter “masswidth”, horizontally centered on the first found pixel, and takes the average x-position of every yellow pixel within this box. This reliably captures the average position of yellow pixels, an appropriate proxy for the position of the mass. It does require that the distances between masses be large compared to the width of the paint strips; the current physical setup of the pendula makes this a reasonable assumption. Once this mass position has been recorded, the code shifts past the mass, based on the “masswidth” parameter, and begins searching for the next mass. The center of each mass is stored and reported to the main function.

The main function primarily handles input and output. The “cv2” package is used to handle the video, and the “Image” package is used for image processing. The main function loads the video, then saves one frame of the video as an image file. It then processes that image, finding the center locations of each mass in that frame. If the number of masses detected does not equal the number of masses specified, an error is reported and the frame is ignored. Otherwise the mass locations are stored, and the process is applied to the next frame. Progress can be reported by setting the “reportframes” parameter, which will generate a message when a certain number of frames are completed. Once the video has finished processing, a graph of the mass locations by frame is reported. The position data is kept in the “centerpoint” list of lists, with each list within “centerpoint” containing the location of that indexed mass for each frame. These values are saved in CSV format to a location specified by the “csvname” parameter, for future processing.

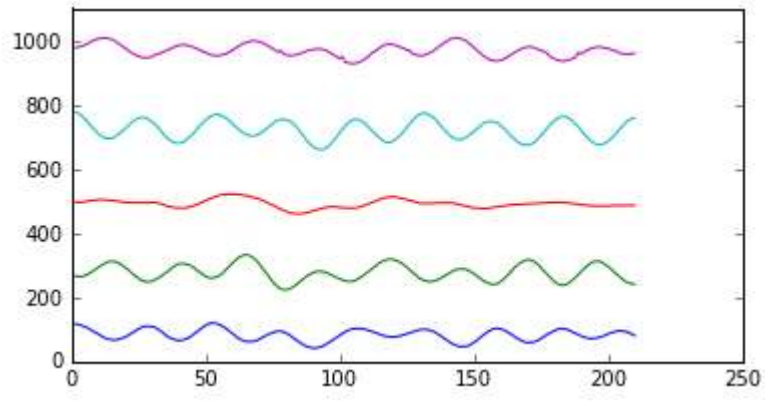

Figure 2: Sample extracted position data for mass movements. Smooth movement is seen, and qualitative comparison to the video seems reliable. Note the symmetry between mass movements away from the central mass. This data was note taken during stable pendulum oscillation.

This code seems reliable and useful for tracking mass positions over time. The yellow identification is effective, and the best way to further improve its reliability would be a physical change to the experimental device; simply putting up black paper behind the masses should solve any difficulties in the future. The major issue with the code at present is the time it takes to run; if videos can be converted to a series of still images, some minor tweaks to the main function should allow much faster throughput – though I'm uncertain if there will be a net gain in performance when including the cost of doing the conversions upfront. Any desired data processing could be appended to this code, but it is likely to be easier to import the CSV into another program.

For future users of the code, I believe the comments should be sufficient. All changeable parameters are included in the list of "set parameters" at the beginning of the program; no changes to any other code are required to use the program as it currently operates.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 14 13:49:34 2020

For processing video data of moving pendula marked with yellow paint,
and extracting information about this motion.

@author: Scott
"""

#Import scipy for graphing, Image for image processing, and cv2 for
video->frame
from PIL import Image
import cv2
import matplotlib.pyplot as plt
import csv

#Set parameters:
vidname = "C:\\Users\\Scott\\Desktop\\CIMG0359.avi" #input video file
tempimgname = "C:\\Users\\Scott\\Desktop\\TempIMG.jpg" #location to store
images during processing
csvname = "C:\\Users\\Scott\\Desktop\\PendLocs.csv" #output location for
CSV file
threshold = 380 #R+G-B value to set yellow pixels; suggest 300-400
depending on light conditions. Test with writeBW
nummasses = 5 #Currently number of masses is hardcoded, and any extra
masses/masses moving out of frame will mess things up
masswidth = 60 #Approximate width of painted stripes in pixels (only
adjust if camera distance changes dramatically)
reportframes = 0 #Gives printout when this many frames are complete. Set
to 0 to disable printout.
minheight = 600 #Pixel value above which masses will not move; set
properly it increses run time and removes errors. Set to 0 to use entire
image.

#The list of center points for each mass, in nested array from,
initialized with no values.
centerpoints=[]
for i in range(nummasses):
    centerpoints.append([])

#A simple function to check if a given pixel meets the threshold (defined
above) for being "yellow"
#Returns True if the pixel is yellow, false otherwise; criteria is r + g
- b
def isYellow(pixelvals):
    if pixelvals[0]+pixelvals[1]-pixelvals[2]>threshold:
        return True
    else:
        return False

# A function for testing, saving a processed image. Good for indicating
if the threshold for
# isYellow() is working properly.
```

```
def writeBW(outputlocation, inputlocation):
    imageToConvert = Image.open(inputlocation)
    bw = Image.new("1",imageToConvert.size,color=0)
    width, height = bw.size
    for i in range(width):
        for j in range(height):
            coordinate = i,j
            if isYellow(imageToConvert.getpixel(coordinate)):
               bw.putpixel(coordinate,1)

    bw.save(outputlocation)

# LEGACY: row with max yellow is no longer used; finding mass positions
uses the entire image.
# A function identifying the row of the image with the most yellow, to be
used for extracting
# mass positions
def findMaxRow(image):
    totyel = 0
    maxrow = 0
    width, height = image.size
    for i in range(height):
        tempyel = 0
        for j in range(width):
            coordinate = (j,i)
            if isYellow(image.getpixel(coordinate)):
                tempyel = tempyel + 1
        if tempyel>totyel:
            totyel = tempyel
            maxrow = i
    return maxrow

# LEGACY: Replaced with findPositions2 and findMassCenter.
# A function locating the center of each stretch of yellow pixels in a
given image, in a given
# row. This returns an array of pixel locations corresponding to these
centers.
def findPositions(image, row):
    startpoints = []
    endpoints = []
    centers = []
    width, height = image.size
    for i in range(width-1):
            coordinate1 = (i, row)
            coordinate2 = (i+1, row)
            if isYellow(image.getpixel(coordinate1))== False:
                if isYellow(image.getpixel(coordinate2)) == True:
                    startpoints.append(i+1)
            else:
                if isYellow(image.getpixel(coordinate2)) == False:
                    endpoints.append(i)
    for i in range(min(len(startpoints),nummasses)):
        centers.append(int((startpoints[i]+endpoints[i])/2))
    return centers
```

```
# Moves across the image searching for yellow pixels, then calls
findMassCenter to
# report the centers of each mass, returning a list of x-positions of
mass centers in the image
def findPositions2(image):
    centers = []
    width, height = image.size
    tempspot = -2*masswidth
    for i in range(width-1):
            for k in range(minheight, height-1):
                if (i-tempspot > 2* masswidth):
                    if isYellow(image.getpixel((i,k)))==True:
                        centers.append(findMassCenter(image,i))
                        tempspot = i
    return centers

#Iterates through locations near a mass, finding the average x-value of
yellow pixels in that area.
def findMassCenter(image, xstart):
    width, height = image.size
    total = 0
    count = 0
    for i in range(max(xstart-masswidth*2,0),
min(xstart+masswidth*2,width-1)):
        for j in range(minheight,height-1):
            if isYellow(image.getpixel((i,j))) == True:
                total += i
                count += 1
    if count == 0:
        return 0
    return total/count

# The main function: loads a video, and loops through each frame, finding
centers of masses.
# These pixel locations are stored to the centerpoints array, and then
plotted and saved to a CSV.
# Errors are reported if a frame does not provide the appropriate number
of mass locations.
try:
    video = cv2.VideoCapture(vidname)
    success, image = video.read()
    frame = 0
    while success:
        cv2.imwrite(tempimgname, image)
        image = Image.open(tempimgname)
        templocs = findPositions2(image)
        if len(templocs)> nummasses:
            print("Error: too many masses found. Frame:" + str(frame))
        if len(templocs) < nummasses:
            print("Error: too few masses found. Frame:" + str(frame))
        if len(templocs) == nummasses:
               for i in range(len(templocs)):
                   centerpoints[i].append(templocs[i])
```

```
        if (reportframes != 0):
            if (frame % reportframes == 0):
                print("Frame " + str(frame) + " complete")
        frame +=1
        success, image = video.read()
    for i in range(len(centerpoints)):
        plt.plot(centerpoints[i])
    with open(csvname, "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerows(centerpoints)

except IOError:
    print("Video read error")
    pass
```