MECE E3028 Mechanical Engineering Laboratory II
Professor Qiao Lin
Spring 2022

Experiment 6: Raspberry Pi Object Tracking Experiment
LABORATORY REPORT

Lab Group 6

Chengxi Wang

Justin Tucker

Kyrie Lorfing

Nico Aldana

Nikita Manjuluri

Will Hamilton

Yumna Alrefaei

Columbia University
Department of Mechanical Engineering

2/14/2022

Contents

Abstract **2**

List of Figures **3**

1 Introduction **4**
1.1 Raspberry Pi 4
1.2 Object Tracking Work Processes 4
1.3 Significance of Study 4

2 Theory **5**
2.1 RGB 5
2.2 HSV 5
2.3 Conversion 5
2.4 PID Closed Loop Control System 5
2.5 Servo Motors 5
2.6 Microprocessors 6
2.7 Arduino vs Raspberry Pi 6

3 Apparatus and Approach **6**
3.1 Apparatus 6
3.2 Approach 7

4 Results and Discussion **9**
4.1 Results 9
4.2 Discussion 10

5 Conclusions **11**

6 Appendix **12**

References **20**

Abstract

In this experiment, we will use a Raspberry Pi microcontroller, a camera, and servo motors to track circular objects in real time. We want to study what a Raspberry Pi is and how it works, different color spaces and their conversion, how a PID control system works, and how to control servo motors with Python code. To track the circular object, we will modify the ball tracking algorithm given by the instructors to move the servo motor accordingly. By changing the upper and lower RGB values in the code, we are able to adjust which color the algorithm detects. The final step is using the given pan-tilt code to make sure that the servo motors track the objects even as it moves out of frame according to the position calculated by the color detection algorithm. Our results aligned with our expectations for this experiment, and we were able to execute the code successfully.

List of Figures

1 The camera, Raspberry Pi, and pan-tilt mechanism used is shown. 7
2 The setup for the first part of the experiment, the object identification without camera movement, is shown. 8
3 The setup for the second part of the experiment, the object tracking with the camera movement, is shown. 9

1 Introduction

1.1 Raspberry Pi

The Raspberry Pi single printed-circuit-board (PCB) is a device that was designed to promote widespread computer science education by the Raspberry Pi Foundation [1]. Upgrades to the Raspberry Pi made over the years gave the device the ability to use Python, and the numerous systems and information libraries that are available in that language, making it even more accessible, especially to engineering students. The device can function essentially like a desktop computer when connected to a monitor, keyboard, and mouse. This is possible because of the Raspberry Pi's microcontroller, which contains a Central Processing unit, Graphics Processing Unit, and 2GB of RAM. These components specifically enable the Raspberry Pi to perform computationally expensive tasks, and therefore perform frame by frame image processing on a live video feed.

1.2 Object Tracking Work Processes

The first step to successfully track circular objects of a specified color was to install the OpenCV Python package onto an SD card running Raspberry Pi OS, which would take frame data from the camera to analyze in real-time [2]. The code itself is responsible for converting the input images to an HSV color space and detecting RGB values within a specified range. The Raspberry Pi board itself is what allows the computer, OS (saved on a microSD card), and camera to communicate. The camera became operational after physically connecting it to the Raspberry Pi and enabling the hardware within the Raspberry Pi OS settings (as well as installing one more package in the virtual environment). Any adjustments necessary to accomplishing the different tasks assigned for this lab could be made by altering the code saved on the microSD using the built-in Thonny editor, which would be executed manually through commands in the Linux terminal.

1.3 Significance of Study

Learning how to use the Raspberry Pi to track objects from a live camera feed is important to the development of our education as engineers. Because this experiment takes many topics we have only studied separately so far, and combines them into a group exercise with a practical application, it serves as another step towards being able to accomplish what engineers do in the real world. For this experiment, we were required to break down complex tasks into simple sequential steps. First, we had to figure out how to get a live camera feed, then implement color-detection, then object tracking, until finally implementing the servo functionality to physically move the camera. Face and object-tracking using cameras and tilting motors are commonly used all over the world for industries such as security, research, and consumer electronics. Using simple microcontrollers (not limited to Raspberry Pi) to implement versatile algorithms in a language as common as Python is an extremely valuable skill to have in mechanical engineering.

2 Theory

2.1 RGB

In RGB color space, each pixel stores 3 values that range from 0 to 255. R represents red, G represents green and B represents blue. If a pixel is purely red, then it would have an RGB value of [255,0,0]. Alternatively, if a pixel is white, it would have an RGB value of [255,255,255]. RGB color space has 3 channels and 8 bits of depth [1].

2.2 HSV

In HSV color space, each pixel has 3 values, H from 0 to 360, S, from 0 to 1, and V from 0 to 1. H is hue, representing the angle of the color in the RGB color circle, where 0 degrees is red, 120 degrees is green, and 360 degrees is blue [1]. S is saturation, representing the intensity of a color, where 0 is grayscale and 1 is natural color. V represents value, representing the brightness of a color, where 0 is pure black and 0 is pure white. HSV color space has 3 channels and 8-bit depth [3].

2.3 Conversion

RGB and HSV color spaces can be converted as following:

$R = R/255, G = G/255, B = B/255$

$V = max(R, G, B) S = (max(R, G, B) - min(R, G, B))/min(R, G, B)$

-If $max(R, G, B) = R, H = 60 * (0 + G - B)/(max(R, G, B) - min(R, G, B))$

If $max(R, G, B) = G, H = 60 * (2 + B - R)/(max(R, G, B) - min(R, G, B))$

If $max(R, G, B) = B, H = 60 * (4 + R - G)/(max(R, G, B) - min(R, G, B)) G = H + 360$ if $H < 0$

2.4 PID Closed Loop Control System

The Proportional-Integral-Derivative (PID) closed loop control system follows a set order: First, the input is sent to the controller. Next, the controller sends the control signal to the processor. Then, the processor returns a measuring element to the controller which is forwarded to the output. PID control will calculate the proportion, integration, and derivative of the error, which is fed back to the input to make the necessary corrections. Signals with large errors when making proportional and integration will be discarded before making the derivative.

2.5 Servo Motors

Servo motors work as a type of feedback loop where a supplied current and voltage cause a DC motor to spin at a certain speed. Within the motor, there is a tachometer that measures whether or not the desired speed has been reached. If it hasn't, it adjusts the

power accordingly. Programming servos requires keeping track of velocity, current, voltage, and torque (in our experiment, we installed Python packages to help facilitate this) [4].

2.6 Microprocessors

A Raspberry Pi 3 board consists of an integrated CPU/GPU, 2 GB RAM, peripheral ports (HDMI, USB, microSD), USB-A power input, and a ribbon cable connector for a camera module. The OS is loaded onto a microSD and displayed through the HDMI output, which the user can navigate by connecting an external mouse and keyboard [5].

2.7 Arduino vs Raspberry Pi

The Arduino and Raspberry Pi have different characteristics that can give either device an advantage or disadvantage. The Arduino has a microprocessor that runs at 8-16 MHz and 2-8 kB built-in RAM. Then the Raspberry Pi has a microcontroller/CPU that runs at 1.5 GHz and has 2 GB RAM. The Raspberry Pi's superior processing power allows it to run a full OS with several programs in parallel—unlike the Arduino, which can only run one program at a time. Additionally, the Arduino is easy to program using Arduino IDE, but it must be connected to a separate computer running its own OS to flash programs onto the board. On the other hand, the Arduino has a cost advantage with it being only $37, which is significantly less than the Raspberry Pi 4 Model B's $70 price point [1].

3 Apparatus and Approach

The apparatus in this experiment consisted of a few pieces of hardware, but mainly software libraries in Python that would allow for camera support and image processing. Much of the approach, then, involved adjusting the code. Since the software had to track different colored balls, the most important part was ensuring that we could access the live video feed from the Raspberry Pi GUI. We use the same camera for both the calibration of RGB values, as well as the execution of the tracking code itself.

3.1 Apparatus

In terms of hardware, a microprocessor was needed for the CPU and GPU necessary to process the highly complex objective of object tracking. In this scenario, a 2GB RAM Raspberry Pi Model 4 was used to allow for frame-by-frame image processing, allowing the CPU to detect colors and movement throughout the video [6].

For the second part of the experiment, involving the movement of the camera to keep the object in the center of its frame, a device is necessary to allow for fluid movement of the camera. For this, the Pan-tilt HAT kit was used, which contained two integrated servos and a pan-tilt bracket system. The camera module, of course, connected to the Raspberry Pi as a peripheral to allow for sensing of the object. Other necessary hardware components include

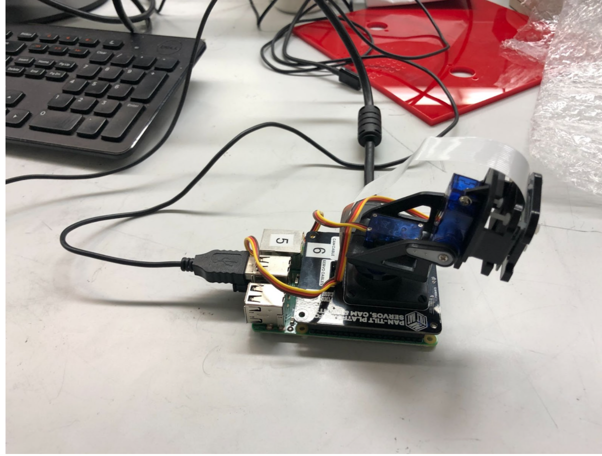

Figure 1: The camera, Raspberry Pi, and pan-tilt mechanism used is shown.

the monitor, keyboard, the connectors between the camera, Raspberry Pi, and monitor, and the microSD card. This is shown in Figure 1 .

Much of the apparatus used in this experiment was software. First, in order to reformat the SD card, the SD card formatter software was used to clear the SD card of data and prepare it for the Raspberry Pi. To process the image, analyze the colors, and track the object, OpenCV is used. This computer vision library allows for conversion between color spaces and object identification and tracking. Once the Raspbian OS is installed to work with the software and Raspberry Pi to run the image processing and object tracking software [7].

3.2 Approach

The first step in completing the experiment is installing the software. In order to work with the SD card, it must be reformatted and cleared. The OpenCV software should be installed on the PC to perform the object tracking. Version four must be downloaded for optimal compatibility with Python 3. Then, to access the operating system that supports the Raspberry Pi and object tracking software, the Raspberry OS must be installed or updated. Then, the camera that is to perform the object tracking must be connected physically, through the connector, and in the software, by enabling the camera as an interfacing option. Then, the camera is ready for a picture of the group to be taken [8].

Once the camera is operational, one can use the Linux command line to install the smbus package in the virtual environment and re-enable the camera and the tilting mechanism.

Since the object tracking software is installed, as well as the servos and tilting mechanism attached to the camera, the object tracking can begin. In the first part, the motion of the red and green balls must be tracked without any movement of the servos. This should be done by adjusting the upper and lower boundaries of the RGB values to fit the color of the green ball. This should involve a guess and check method and lead to a better understanding

of the values. One might find, for example, that lighter colors are related to a higher RGB value and darker colors are related to a lower value. The RGB values for the red ball, again, through estimation, were found to have a lower limit of [0, 50, 50] and an upper limit of [10, 255, 255]. This part of the experiment is shown in Figure 2

Once this is completed, the code must be adjusted to track the movement of the ball. The code in the objcenter.py file should already be formatted to track faces using the Haar Cascade face detector, so this will involve adjustments for the OpenCV software to recognize and analyze the colors of the moving balls [8]. Since the software converts the RGB values to HSV values, the surroundings can be better separated from the object when processing the frames. This involves editing the code that allows for the Pan-Tilt mechanism to work with the face detection and adjusting the identifying HSV values to work for the different balls and their colors. This part of the experiment is shown in Figure 3. Once this is done, one can test the range of motion of the camera and the Pan-Tilt mechanism by moving the balls throughout and past the range of the camera. Using this approach, one can gain a better understanding of the RGB values, the Raspberry Pi, the camera and associated mechanism, and the python code that controls the object detection and tracking operations [2].

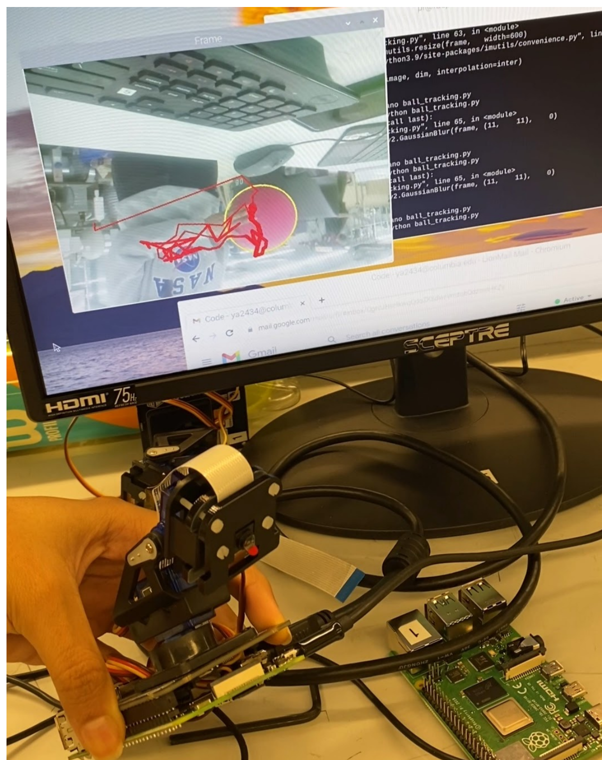

Figure 2: The setup for the first part of the experiment, the object identification without camera movement, is shown.

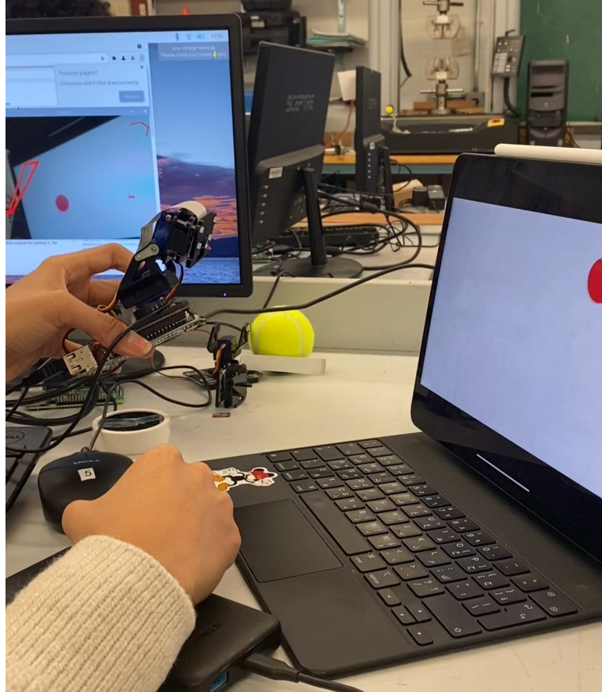

Figure 3: The setup for the second part of the experiment, the object tracking with the camera movement, is shown.

4 Results and Discussion

4.1 Results

By the end of our experiment, we were successfully able to track colors within specified RGB ranges and pan/tilt two servo motors to keep an object within the center of the frame. We began by first tracking the path of a colored object across a still frame, keeping the position of the center of the object as a queue of points updated at a set framerate. Then, we implemented a PID control system to adjust the pan and tilt of the camera to keep the center of the object as close to the center of the video frame as possible. In order to ensure that our motors were able to adjust quickly and smoothly, we needed to tune the parameters of our PID algorithm to suitable values. There are three values that need to be tuned: K_p, K_i, and K_p. It is important to adjust the values by very small increments, since the "sweet spot" of values is very narrow, and poorly-tuned values can cause the motors to "overshoot" while compensating for the object motion leading to unnecessary oscillations and instability in the tracking motion.

The foundation of our algorithm is based on the detection of RGB values within an HSV color space. We accomplished this in a series of steps, beginning with converting the raw input from the camera to a form that is easily parsed by the Raspberry Pi. In our code, we begin by initializing the camera object, taking one frame at a time, blurring the image,

and converting to an HSV color space using built-in OpenCV functions. We apply a blur to the image in order to reduce the complexity of the images and "smooth out" high and low contrast areas. We were able to identify the upper and lower RGB bounds of our desired color by running the range-detector script that is part of the imutils Python library. The conversion to the HSV color space is necessary to simplify the colors within the image to reduce the load on the processor. This allows the program to run with less latency, achieving our goal of analyzing the image in real time.

The final code is given in the Appendix section and the final result of our experiment was demonstrated to the Teaching Assistant.

4.2 Discussion

To trace the ball without enabling motors, the user must first define upper and lower boundary RGB values to specify a color to detect [8]. (In our experiment, the values are tuned to green.) The program then begins collecting each frame captured by the camera module and converting the image to an HSV color space. Then, the program applies a mask to filter out all the colors that are not encompassed by the previously defined RGB boundaries. If contours within the color range are found, the program then computes the minimum size of a circle that would enclose the contour. From there, the program overlays a circle over the video feed and continuously records the x-y position of the center of each bounding circle. By plotting lines through the list of positions, the output video feed appears to track a singular object.

In the original pan tilting program, the objectcenter file detects faces. In this experiment, we want to modify it so that it finds the center of a circle so that the motors can move accordingly. To modify this program, we implement the part of the object tracing code without enabling motors which find the center of circles of a specific color [2]. Specifically, the program finds an object whose color falls in between the upper and lower limit of the expected color, creates a circle contouring the object and finds its center.

To determine the upper limit and lower limit of our expected color, we searched the RGB value for colors similar to our expected color. For example, the RGB value for light green is (144,238,144) and that for deep green is (5,102,8) [3]. Once we have these two limits, we test it on the code and check if the camera can trace the object. If the camera is able to trace the object, we increase the proportion of G value in the lower limit and decrease the proportion of G value in the upper limit. If the camera is not able to trace the object, we decrease the proportion of G value in the lower limit and increase the proportion of G value in the upper limit. Conversion from RGB to HSV is completed by the algorithm.

To stress test the code, we use our phone to increase the lighting on the object and compare the performance of the camera tracking. We also place a background of similar color behind the object and compare the tracking performance. If our algorithm fails in the testing, we decrease the proportion of G value in the lower limit and increase the proportion of G value in the upper limit.

The pan-tilt hat assembly allowed the camera to rotate 180 degrees from side-to-side and tilt up and down to follow the object of interest. It had a delayed response to the movement

of the object due to the CPU delays in processing the code, but it managed to follow first faces, then objects, while keeping the object generally within its frame, rather than at the exact center of the screen. It did seem to accelerate slowly compared to the motion of the object. There are three values that need to be tuned: K_p, K_i, and K_p. These must be adjusted by small increments in order to find the precise value that will tune the movements of the motor and compensate for the object motion. To do so, we followed the manual tuning method as outlined in the given source [9].

There were a few software glitches and issues that had to be resolved in order to finish the experiment. Primarily, many of the connectors were faulty. This includes the connection between the camera and the Raspberry Pi as well as the connection between the monitor and Raspberry Pi. This was fixed by simply changing the connectors. Similarly, the camera was not connecting correctly to the CPU, causing the error that the camera was not “enabled” to repeatedly come up. Once all the connections were working and in the proper place, it was difficult to work with the servo since it was built incorrectly and jammed. This was repaired by assembling the camera, servos, and connections and reassembling them in the right order.

In the future, it may be necessary to slow the motion of the circles that the camera is to track, since by the time the software registered the image of the circles on the screen, they had moved out of range. Also, the hardware used and the OS itself should be updated in order to avoid glitches and constant rebooting of the system. It would also be useful to test the image analysis capabilities of OpenCV with simpler code in order to bypass large glitches in software and focus on these capabilities.

5 Conclusions

The basis of this experiment was to first use a setup consisting of a Raspberry Pi microprocessor/CPU, monitor, keyboard and mouse, and Python code (in conjunction with the OpenCV package library) to create an algorithm that would track a colored ball on a screen. Then, we set out to modify the original program to tilt and pan the input camera using two servo motors. This would allow the program to ”track” an object and keep it in the center of the video frame. We set out to complete this experiment to understand how a PID control system works, as well as how to implement it in position-tracking code. We also had to understand different color spaces and how to to convert between them. We took image data from the camera and converted it into an HSV color space to allow the program to better parse through the color values. The stationary ball-tracking was accomplished by first loading the Raspberry Pi OS onto the board and connecting the necessary hardware. Next, using the framework code provided in the experiment procedure, we were able to implement an object tracking algorithm. The object tracking (with camera movement enabled) was accomplished by using a PID control system. A PID system calculates error in real time and feeds it back into its own input in order to adjust itself until the calculated error is zero. The camera tracks the object’s center and then uses proportional, integral, and derivative means to provide an accurate and responsive correction to a control function in real time. This

prevents wasted movements in the object-tracking movements performed by the camera and leads to a smooth line-of-sight.

6 Appendix

The code can be described in a sequence of events beginning from enabling camera support and ending with tracing a line between a collection of points.

```
from collections import deque
from imutils.video import VideoStream
import numpy as np
import argparse
import cv2
import imutils
import time

ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video",
help="path to the (optional) video file")
ap.add_argument("-b", "--buffer", type=int, default=64,
help="max buffer size")
args = vars(ap.parse_args())

# DEFINE BOUNDARIES
greenLower = (29, 86, 6)
greenUpper = (64, 255, 255)
pts = deque(maxlen=args["buffer"])
if not args.get("video", False):
vs = VideoStream(src=0).start()
else:
vs = cv2.VideoCapture(args["video"])
# allow the camera or video file to warm up
time.sleep(2.0)

while True:
# FRAME CAPTURE
frame = vs.read()
frame = frame[1] if args.get("video", False) else frame
if frame is None:
break
# COLOR SPACE CONVERSION
frame = imutils.resize(frame, width=600)
```

```
blurred = cv2.GaussianBlur(frame, (11, 11), 0)
hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
# COLOR DETECTION
mask = cv2.inRange(hsv, greenLower, greenUpper)
mask = cv2.erode(mask, None, iterations=2)
mask = cv2.dilate(mask, None, iterations=2)
# find contours in the mask and initialize the current
# (x, y) center of the ball
cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
center = None
# CONTOUR DETECTION
if len(cnts) > 0:
# CIRCLE COMPUTATION
c = max(cnts, key=cv2.contourArea)
((x, y), radius) = cv2.minEnclosingCircle(c)
M = cv2.moments(c)
center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
if radius > 10:
# DRAW CIRCLE
cv2.circle(frame, (int(x), int(y)), int(radius),
(0, 255, 255), 2)
cv2.circle(frame, center, 5, (0, 0, 255), -1)
# PATH TRACING
pts.appendleft(center)
for i in range(1, len(pts)):
if pts[i - 1] is None or pts[i] is None:
continue
thickness = int(np.sqrt(args["buffer"] / float(i + 1)) * 2.5)
cv2.line(frame, pts[i - 1], pts[i], (0, 0, 255), thickness)
# UPDATE FRAME
cv2.imshow("Frame", frame)
key = cv2.waitKey(1) & 0xFF
# QUIT CONDITION
if key == ord("q"):
break
if not args.get("video", False):
vs.stop()
else:
vs.release()
cv2.destroyAllWindows()
```

To keep the center of the object when enabling motors, we use the code below.

```
    # import necessary packages
import imutils
import cv2

class ObjCenter:
    def __init__(self, haarPath):
        # load OpenCV's Haar cascade face detector
        self.detector = cv2.CascadeClassifier(haarPath)

    def update(self, frame, frameCenter):
        greenLower = (29, 86, 6)
        greenUpper = (64, 255, 255)
        pts = []
        # convert the frame to grayscale
        frame = imutils.resize(frame, width=600)
        blurred = cv2.GaussianBlur(frame, (11, 11), 0)
        hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
        # COLOR DETECTION
        mask = cv2.inRange(hsv, greenLower, greenUpper)
        mask = cv2.erode(mask, None, iterations=2)
        mask = cv2.dilate(mask, None, iterations=2)
        # find contours in the mask and initialize the current
        # (x, y) center of the ball
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        center = None
        # CONTOUR DETECTION
        if len(cnts) > 0:
            # CIRCLE COMPUTATION
            c = max(cnts, key=cv2.contourArea)
            ((x, y), radius) = cv2.minEnclosingCircle(c)
            M = cv2.moments(c)
            center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
            if radius > 10:
                # DRAW CIRCLE
                cv2.circle(frame, (int(x), int(y)), int(radius),
                (0, 255, 255), 2)
            cv2.circle(frame, center, 5, (0, 0, 255), -1)
    # PATH TRACING
        pts.appendleft(center)
```

```
        # check to see if a circle was found
        if len(pts) > 0:

            (x, y) = pts[0]

            # return the center (x, y)-coordinates of the circle
            return ((x, y))

        # otherwise no circle were found, so return the center of the
        # frame
        return (frameCenter, None)
```

The main code when enabling motors is shown below.

```
    # USAGE
# python pan_tilt_tracking.py --cascade haarcascade_frontalface_default.xml

# import necessary packages
from multiprocessing import Manager
from multiprocessing import Process
from imutils.video import VideoStream
from pyimagesearch.objcenter import ObjCenter
from pyimagesearch.pid import PID
import pantilthat as pth
import argparse
import signal
import time
import sys
import cv2

# define the range for the motors
servoRange = (-90, 90)

# function to handle keyboard interrupt
def signal_handler(sig, frame):
# print a status message
print("[INFO] You pressed ‘ctrl + c‘! Exiting...")

# disable the servos
pth.servo_enable(1, False)
pth.servo_enable(2, False)
```

```
# exit
sys.exit()

def obj_center(args, objX, objY, centerX, centerY):
# signal trap to handle keyboard interrupt
signal.signal(signal.SIGINT, signal_handler)

# start the video stream and wait for the camera to warm up
vs = VideoStream(usePiCamera=True).start()
time.sleep(2.0)

# initialize the object center finder
obj = ObjCenter(args["cascade"])

# loop indefinitely
while True:
# grab the frame from the threaded video stream and flip it
# vertically (since our camera was upside down)
frame = vs.read()
frame = cv2.flip(frame, 0)

# calculate the center of the frame as this is where we will
# try to keep the object
(H, W) = frame.shape[:2]
centerX.value = W // 2
centerY.value = H // 2

# find the object's location
objectLoc = obj.update(frame, (centerX.value, centerY.value))
((objX.value, objY.value), rect) = objectLoc

# extract the bounding box and draw it
if rect is not None:
(x, y, w, h) = rect
cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0),
2)

# display the frame to the screen
cv2.imshow("Pan-Tilt Face Tracking", frame)
cv2.waitKey(1)
```

```
def pid_process(output, p, i, d, objCoord, centerCoord):
# signal trap to handle keyboard interrupt
signal.signal(signal.SIGINT, signal_handler)

# create a PID and initialize it
p = PID(p.value, i.value, d.value)
p.initialize()

# loop indefinitely
while True:
# calculate the error
error = centerCoord.value - objCoord.value

# update the value
output.value = p.update(error)

def in_range(val, start, end):
# determine the input vale is in the supplied range
return (val >= start and val <= end)

def set_servos(pan, tlt):
# signal trap to handle keyboard interrupt
signal.signal(signal.SIGINT, signal_handler)

# loop indefinitely
while True:
# the pan and tilt angles are reversed
panAngle = -1 * pan.value
tltAngle = -1 * tlt.value

# if the pan angle is within the range, pan
if in_range(panAngle, servoRange[0], servoRange[1]):
pth.pan(panAngle)

# if the tilt angle is within the range, tilt
if in_range(tltAngle, servoRange[0], servoRange[1]):
pth.tilt(tltAngle)

# check to see if this is the main body of execution
if __name__ == "__main__":
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
```

```
ap.add_argument("-c", "--cascade", type=str, required=True,
help="path to input Haar cascade for face detection")
args = vars(ap.parse_args())

# start a manager for managing process-safe variables
with Manager() as manager:
# enable the servos
pth.servo_enable(1, True)
pth.servo_enable(2, True)

# set integer values for the object center (x, y)-coordinates
centerX = manager.Value("i", 0)
centerY = manager.Value("i", 0)

# set integer values for the object's (x, y)-coordinates
objX = manager.Value("i", 0)
objY = manager.Value("i", 0)

# pan and tilt values will be managed by independed PIDs
pan = manager.Value("i", 0)
tlt = manager.Value("i", 0)

# set PID values for panning
panP = manager.Value("f", 0.09)
panI = manager.Value("f", 0.08)
panD = manager.Value("f", 0.002)

# set PID values for tilting
tiltP = manager.Value("f", 0.11)
tiltI = manager.Value("f", 0.10)
tiltD = manager.Value("f", 0.002)

# we have 4 independent processes
# 1. objectCenter  - finds/localizes the object
# 2. panning       - PID control loop determines panning angle
# 3. tilting       - PID control loop determines tilting angle
# 4. setServos     - drives the servos to proper angles based
#                    on PID feedback to keep object in center
processObjectCenter = Process(target=obj_center,
args=(args, objX, objY, centerX, centerY))
processPanning = Process(target=pid_process,
args=(pan, panP, panI, panD, objX, centerX))
```

```
processTilting = Process(target=pid_process,
args=(tlt, tiltP, tiltI, tiltD, objY, centerY))
processSetServos = Process(target=set_servos, args=(pan, tlt))

# start all 4 processes
processObjectCenter.start()
processPanning.start()
processTilting.start()
processSetServos.start()

# join all 4 processes
processObjectCenter.join()
processPanning.join()
processTilting.join()
processSetServos.join()

# disable the servos
pth.servo_enable(1, False)
pth.servo_enable(2, False)
```

References

[1] Columbia University. Rasberry pi object tracking experiment: Description, 2022.

[2] Juan Cruz Martinez. Object tracking with opencv. *LCS - Learn and grow with us*, Oct 2021.

[3] Programming Design Systems. Color models and color spaces. https://programmingdesignsystems.com/color/color-models-and-color-spaces/index.html.

[4] Jameco Electronics. Servo Motors. https://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html.

[5] Raspberry Pi Configuration. https://www.raspberrypi.org/documentation/configuration/camera.md, 2012.

[6] Py Image Search. Rapberry pi documentation. https://www.pyimagesearch.com/2019/04/01/pan-tilt-face-tracking-with-a-raspberry-pi-and-opencv/.

[7] Raspberry Pi OS download. https://www.raspberrypi.org/downloads/raspbian/, 2012.

[8] Mechanical Engineering Lab II. *Installation and Procedure of Rasberry Pi Experiment.* Columbia University, 2022.

[9] Pid controller. https://en.wikipedia.org/wiki/PID$_controllerManual_tuning, Feb$2022.