MECE E3028 Mechanical Engineering Laboratory II
Professor Qiao Lin
Spring 2022

Experiment E6: Raspberry Pi Object Tracking

LABORATORY REPORT

Lab Group 12

Axel Ortega

Bruno Rergis

Anton Deti

Arlene Diaz

Christine Zou

Ashton Buchanan

Samuel Adeniyi

Columbia University
Department of Mechanical Engineering

February 16, 2022

Contents

Abstract

From human-computer interaction to visual surveillance, object tracking methods has increasingly led to rapid development of deep learning over the last few years. This experiment explores two single-object tracking algorithms. Three main programs are executed: a static tracking system that recognizes single-object color profiles, a static tracking algorithm that tracks an object's center using a moving, defined frame, and a dynamic tracking algorithm that attempts to maintain an object's center in the center of the camera's frame using a servo assembly. Color ranges that could be used to track a red ball and a green ball were identified; the static tracking program was successfully executed using these color ranges; and the dynamic tracking program was successfully implemented by modifying a program used to identify faces and by re-using the key components of the static ball-tracking program. The results demonstrate the power of microcontrollers and demonstrate the effectiveness of implementing computer vision in industry.

List of Figures

List of Tables

1 Introduction

Object tracking is an important task when it comes to computer practices that deal with vision. It is a fundamental application and is widely used in the traffic control field as well as military and driver assistance systems. However, there are many different devices that vary in cost, size, time-processing, power consumption, and other imperfections that continue to produce research in this field. The motivation behind this experiment is to learn more about Raspberry Pi and its applications in object work tracking processes.

The objective of this experiment is to implement an object tracking algorithm using a Raspberry Pi Model 4 and a 2-degree-of-freedom camera mount system. The Raspberry Pi is configured by downloading Raspbian operational software, installing and running programs using Linux and the operating system's terminal, and programming the object tracking algorithm in Python. This also requires enabling the Raspberry Pi camera module. This object tracking algorithm works in conjunction with a camera mount system, which is assembled using a camera and a motor. The object tracking algorithm is then adapted to track circular disks of varying colors, paths, and trajectories. To improve the accuracy of the tracking, an Open Source Computer Vision (OpenCV) library for data image processing is implemented to convert the images of the circular disks from an RGB color space to an HSV (hue-saturation-value) grayscale color space as a color filter. When using the camera mount system, the position of the object in this experiment is tracked using a Proportional-Integral-Derivative (PID) control system.

2 Theory

2.1 RGB and HSV Color Spaces

An RGB (red-green-blue) color space is a light-based, additive color space based on the RGB color model (1). This model combines primary colors red, green, and blue in various ways to reproduce a broad array of colors. HSV is an alternative representation of the RGB model. As shown in Figure 2 below, the HSV color space is a cylindrical color model that remaps this RGB model into dimensions that are easier for the human eye to interpret (2). The angular dimension in the cylinder denotes Hue, which specifies the angle of the color on the RGB color circle. At 0° hue maps red, 120° maps green, and 240° maps blue. Saturation, which can be seen in the horizontal dimension, controls the amount of color used (2). This can also be thought of as the amount of gray in a color and ranges from 0 to 100. Lowering this component to 0 introduces more gray into the color, whereas increasing to 100 displays the primary color (2). Finally there is value which represents the brightness of the color (2). This parameter is depicted by the vertical dimension in the HSV cylinder below and can range from 0 to 100, where 0 is completely black and 100 is the brightest.

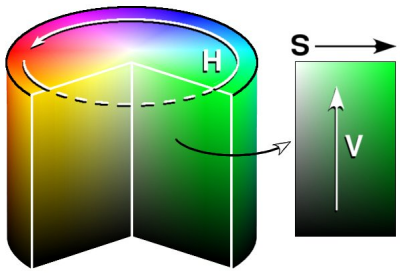

Figure 1: HSV Color Space Cylinder

An image in an RGB or an HSV space contains a collection of pixels which are arrays of numbers(2). Generally, an image undergoing image processing with an OpenCV is characterized by its number of channels, which is the dimension of the array each pixel is, and its depth or maximum bit size of the number stored in the array(2). For an image in RGB or HSV, each pixel contains 3 values where each value ranges from 0 to 255. They both have a depth of 8 bits and three channels(2).

In order to convert an image from an RGB color space to a HSV color space, the following statements are adapted into Python script for this lab (2):

$$V = max(R, G, B) \tag{1}$$

$$S = \frac{max(R, G, B) - min(R, G, B)}{min(R, G, B} \tag{2}$$

where S=0 if V=0

$$H = 60 \times \begin{cases} \frac{0+(G-B)}{\Delta} & \text{if } max(R, G, B) = R \\ \frac{2+(B-R)}{\Delta} & \text{if } max(R, G, B) = G \\ \frac{4+(R-G)}{\Delta} & \text{if } max(R, G, B) = B \end{cases} \tag{3}$$

$$\Delta = max(R, G, B) - min(R, G, B) \tag{4}$$

$$H = H + 360° \text{if } H < 0 \tag{5}$$

2.2 PID Closed-Loop Control System

A proportional-integral-derivative (PID) controller is a closed-loop control mechanism that provides feedback to a system in order to maintain control on process variables in an experiment such as pressure, temperature, level, and flow rate (3). This system continuously calculates an error term as the difference between a desired state and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (2). Figure 2 below illustrates this process.

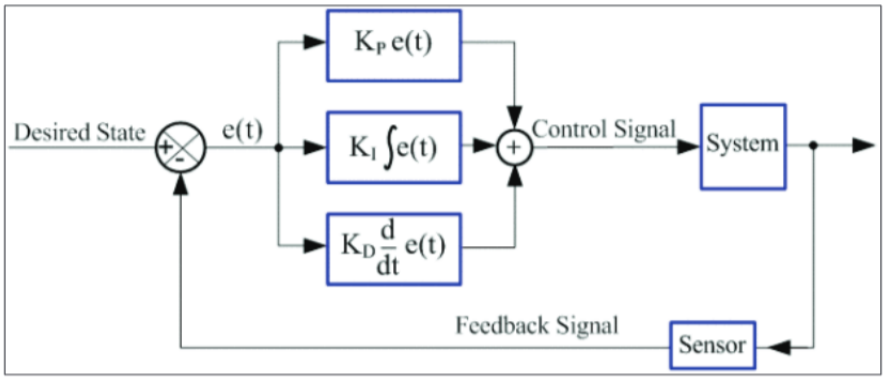

Figure 2: PID Control System Diagram

The system is given a desired value for a process variable, and at any moment within the loop, the difference between the process variable and desired state is used by the control system algorithm to find the desired actuator output that will drive the system (3). The proportional component of the controller depends on an error term, which is the difference between the desired value and the process variable (2). The integral component sums the error term over time. This component will continuously increase until the steady-state error is driven to zero (3). This steady-state error is simply the final difference between the desired value and the process value The derivative component reduces the output if the process variable or measuring element is increasing quickly(3). All these responses work jointly to minimize the amount of error in a system. For this experiment, a PID control system will be implemented in the object tracking algorithm to compensate the error in tracking the positions of the circular disks.

2.3 Servo Motors

A servo motor is an electrical device available in Alternating Current (AC) and Direct Current (DC) configurations that rotates parts of a machine. The motor is coupled with an encoder, potentiometer, and a control circuit (4). It utilizes the PID closed-loop system and provides torque and velocity as commanded from the PID controller. For this experiment, two DC motors are used. They contain gear heads for optimum speed and torque and position sensors which utilize the encoder and potentiometer mechanisms from Figures 3 and 4 (2).

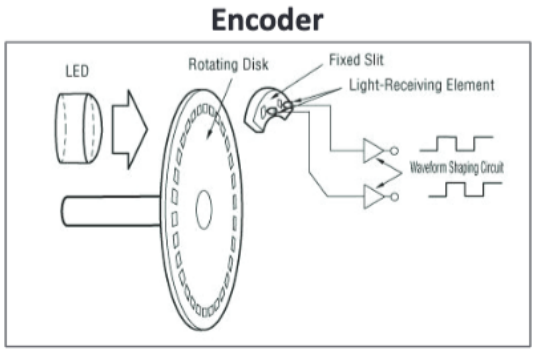

Figure 3: Diagram of servo encoder components.

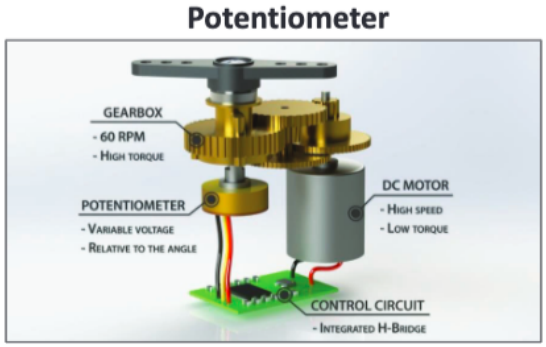

Figure 4: Diagram of servo potentiometer.

The encoder component converts a mechanical input into an electrical impulse that is then transmitted as a signal through a wire(5). This determines the movement of the shaft on the motor(4). The servos are commanded to move according to the positions specified in the tracking algorithm, specifically the PID portion of the program. The potentiometer component in the motor acts as a variable resistor. In the context of the motor, this means that as it rotates, the potentiometer's resistance changes, so the control circuit can dictate the amount of movement and direction (4).

2.4 Basics of Raspberry Pi

The Raspberry Pi is a single printed-circuit-board (PCB) computer that can be connected to a monitor, keyboard, and mouse to function like a desktop computer. The Raspberry Pi computer was developed in 2012 by the Raspberry Pi Foundation to promote basic computer science in developing countries. Since then, the computer has became very popular and there have been upgrades to the system. Python-based libraries and other support systems make it easier for students and engineers to implement mechatronics and other engineering systems using the Raspberry Pi. The computer model has a large online community that makes finding information about the computer very easy by using platforms such as: the Raspberry Pi website, GitHub, YouTube, and other online forums.

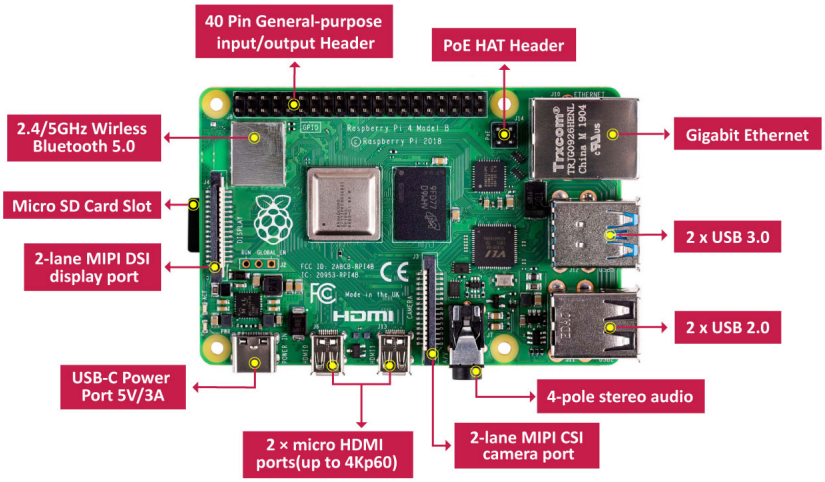

Figure 5: Raspberry Pi Model 4.

As shown in Figure 5, the Raspberry Pi board is a small computer composed of general purpose input-output pins (GPIO), a central processing unit (CPU), random access memory (RAM), a camera connection, a microSD card slot or memory slot, four USB ports, a microHDMI port, and an Ethernet(2).

2.5 Arduino vs. Raspberry Pi

The Raspberry Pi often gets compared to Arduino when it comes to various electronics projects. However, the biggest difference between the two is that an Arduino is a microcontroller whereas the Raspberry Pi is a microprocessor that acts as a computer. For the Arduino, it only contains the CPU, RAM, and ROM. However, the Raspberry Pi has all of that in addition to a processor, memory, storage, graphics driver, and connectors on the board. Because of its processor, it is able to run multiple tasks at once in contrast to the Arduino which is typically used only for running a single task (6). The difference in computational power leads to the higher price to purchase a Raspberry Pi.

Because the Arduino is a microcontroller, it does not require an operating system to run, only the binary of a source code, whereas the Raspberry Pi has its own operating system, Raspberry Pi OS, in order to run. The Raspberry Pi has a significantly more powerful CPU compared to an Arduino. However, because the Raspberry Pi requires an OS as well, it requires more power to run than an Arduino (7). In general, the hardware and firmware for a Raspberry Pi is also closed-source, but has always been open-source for Arduino.

Ultimately the Arduino is best for doing simple or repetitive tasks such as reading sensors or switching something on/off, while the Raspberry Pi is better for complicated tasks such as driving robots or controlling cameras (6).

3 Apparatus and Approach

The equipment that is used in the experiment include a Raspberry Pi Model 4, a Pan-Tilt HAT Servo Motor Assembly, and an Aokin Raspberry Pi Camera Module. This hardware is supplemented by the use of OpenCV within the Raspbian OS to program the ball-tracking algorithm.

3.1 Apparatus

For this experiment, all necessary parts will be provided. Within the Raspberry Pi System, there is the Raspberry Pi Model 4 microcontroller, with 2 GB of RAM allowing for frame-by-frame image processing. This microcontroller includes a microSD card as it uses flash memory to store all files, programs, etc. It also includes numerous cables, ports, and pins that allow the board to be connected to other devices as seen in Fig. 6. This board is attached to a camera mount system, consisting of a Pan-Tilt HAT kit and camera support system (camera system must be enabled before it can be used). The board will then be connected to a keyboard, mouse, and HDMI compatible monitor. The fully assembled Raspberry Pi System will then be connected to a monitor. With a fully assembled Raspberry Pi System, an enabled camera, and all necessary programs available for the system's functionality, data will be able to be collected.

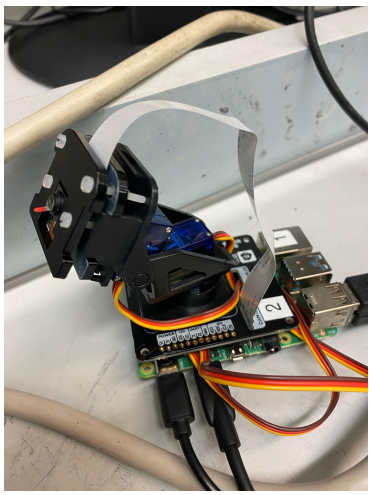

Figure 6: Assembled Raspberry Pi Kit.

3.2 Approach

As stated, there are several software programs that must be installed to perform the experiment. First, on a personal computer, install a version of the operating system recommended by Raspberry Pi manufacturers; for this experiment, Raspbian BusterOS was chosen. The provided microSD card must be flashed with the Rapsbian BusterOS. To insert the microSD into a computer, a microSD-SD card adapter and/or a SD-USB adapter may be necessary.

Then to flash the microSD with the BusterOS, the balenaEtcher software can be used. Alternatively, the Raspberry Pi Imager can be downloaded onto a personal computer. The Raspberry Pi Imager can then be used to select the BusterOS and then directly flash to the microSD card, completely circumventing the need for balenaEtcher.

With the microSD card prepared with a Raspberry Pi operating system, it can then be inserted into the Raspberry Pi and the Raspberry Pi can then be turned on. Successful operation of the Raspberry Pi depends on the use of peripherals such as a USB mouse, a USB keyboard, and a monitor. From here, the majority of the experiment must be done via a terminal or via a Python IDE.

Once the Raspberry Pi has been powered on and is connected to the appropriate peripherals, the camera module must first be enabled. To do so, in a terminal "sudo raspi-config" must first be inputted, so that the filesystem expands. Then under "Interfacing Options" the camera module can be enabled, and the system will reboot. Next, follow the steps outlined on this site to download and install OpenCV in order to work in a virtual environment (8). The necessary commands to run in the Raspberry Pi terminal to successful virtual Python environment to download specific versions of libraries and dependencies for the experiment are:

```
sudo apt-get install build-essential cmake pkg-config
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev
libpng-dev
sudo apt-get install libavcodec-dev libavformat-dev
libswscale-dev libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install libfontconfig1-dev libcairo2-dev
sudo apt-get install libgdk-pixbuf2.0-dev libpango1.0-dev
sudo apt-get install libgtk2.0-dev libgtk-3-dev
sudo apt-get install libatlas-base-dev gfortran
sudo apt-get install libhdf5-dev libhdf5-serial-dev libhdf5-103
sudo apt-get install libqtgui4 libqtwebkit4 libqt4-test python3-pyqt5
sudo apt-get install python3-dev
```

Then, to create a virtual environment, the commands executed in the console are:

```
wget https://bootstrap.pypa.io/get-pip.py
sudo python get-pip.py
sudo python3 get-pip.py
sudo rm -rf ~/.cache/pip
sudo pip install virtualenv virtualenvwrapper
nano ~/.bashrc
```

At this point, the bashrc file will open in which you must add the following code to the bottom of the file:

```
# virtualenv and virtualenvwrapper
```

```
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh
source ~/.bashrc
```

Now a virtual environment may be created. In this case, the virtual environment will be called cv:

```
mkvirtualenv cv -p python3.7
```

Finally, a few more necessary packages had to be installed for object tracking. Input following lines of code into the terminal.

```
cd ~/.virtualenvs/cv/lib/python3.5/site-packages/
ln -s /usr/lib/python3/dist-packages/smbus.cpython-35m-arm-linux-
gnueabihf.so smbus.so
```

Enable the camera again and then perform these next steps in terminal as well.

```
workon cv
pip install --upgrade pip
pip install pantilthat
pip install imutils
pip install numpy==1.21.0
pip install "picamera[array]"
pip install opencv-contrib-python==4.1.0.25
```

With the Raspberry Pi setup and all necessary packages installed, programming for object tracking can begin within the cv virtual environment on the Raspberry Pi. The first task is to write a code to track a ball without the motion of pan tilt module. To do so, follow the steps outlined here (9). A color filter is instantiated within this program to limit what objects can and cannot be identified. By running the program, objects within the detection range can be statically tracked by the camera as indicated by a moving frame that follows the object as it moves.

The final objective was to modify a facial recognition algorithm within the cv virtual environment using the initial ball-tracking program, to dynamically track an object's center on a frame with the pan-tilt hat assembly to physically track the object's center. Two balls of a distinct, uniform color were tracked in this manner by using the color filtering from before. This link can be used as a resource for this step (10).

4 Results and Discussion

As a whole, the investigation into object tracking through the use of a Raspberry Pi was a success. Both objectives - tracking without the motor and with the motor enabled - were fulfilled. The first, running a program to track a moving object on screen without enabling any motors, was achieved through the provided code. Through the use of specific HSV

values that were called upon in that code which filtered out unlike colors, the ability to track specifically-colored objects was gained. The second portion - implementing a program that moves the camera while tracking an object on screen - was also achieved. This was successfully done via the modification of code to identify specific HSV bounds in frame and enclose it with an identifying box as opposed to identifying faces; this was then linked to the servo-motor assembly and the prepared programming through logic changes in the code to enable the camera to move and successfully track the movement of colored balls, even when the movement was jarring and at high-speed.

4.1 Ball Color Identification and Static Tracking

The first set of data collected was the upper and lower HSV-values that filter out colors that were neither red nor green.

This was done by using the code, **pyimagesresearch**, to correctly identify a green and red ball by executing the range-detector program to determine the upper and lower bound of the red color spectrum. The program specifically displays a binary image of the image that the camera gathers and filters out colors; colors that are within the range selected are displayed as white and colors that are outside the range selected are displayed as black. Adjustments were made to capture a slightly larger range of HSV values for both red and green to prevent changes in the lighting and the background from affecting the detection of the balls.

The HSV values selected to track the red ball were an HSV minimum of (0, 180, 70) and an HSV maximum of (15, 255, 255). The HSV values selected to track the green ball were an HSV minimum of (29, 86, 153) and an HSV maximum of (164, 255, 255). The filters were applied in both the ball_tracking.py program and in the modified objcenter.py program. These programs can be found in the Appendix.

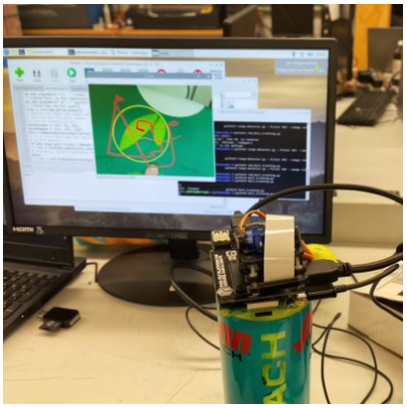

Figure 7: Tracking of a green object.

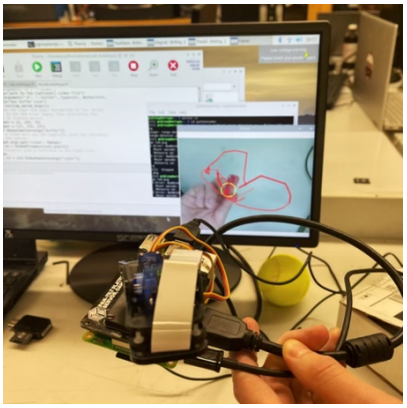

Figure 8: Tracking of a red object.

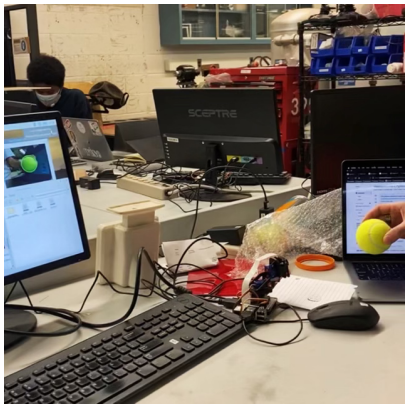

Figure 9: Camera tracking ball's center on a frame.

The program that was run such that static ball tracking was achieved - i.e. without

enabling motors - relied on the colors identified using the range-detector.py code. The program that was executed was ball_tracking.py. This program identified the ball and drew a frame around the ball by using a color filter to sort out what objects to track. All the results for static tracking that follow from the codes provided in the appendix are reasonable, considering the code did a suitable job of identifying the green ball and tracking it while on screen. The modification made to the original ball tracking code such that the bounds of the HSV values reflect that of the color red also did a reasonable job of following a red object across the frame. The use of the range-detector.py code in order to identify those HSV values for the green and red balls allowed for a systemic way of determining the proper values, and thus the code for ball tracking ran in a straightforward manner. Satisfactory static tracking on screen was achieved for both green and red objects.

4.2 Ball Tracking with Enabled Motor

The dynamic tracking, which entailed maintaining the center of the camera's frame on the center of the ball, was a more involved process. The base code was the objcenter.py program that was coded to interpret the camera module's image, search for a human face, and then return the coordinates of the center of the face. This program was modified to use a large portion of the ball_tracking.py program to return the center coordinates of a ball - and a radius slightly larger than the ball - using the color filtering mentioned in the algorithm for the static tracking. See Appendix for the full ball_tracking.py program. This objcenter.py program, in addition to a PID control law in PID.py, is instantiated in the main program pan_tilt_tracking.py program - both programs are also present in the Appendix. By implementing these changes to the code, in which instead of identifying the center of faces, the objcenter.py program which is called upon by the main code pan_tilt_tracking.py identifies the center of circular regions in the video feed of a specific HSV range that is chosen by the programmer, as done in ball_tracking.py, the pan_tilt_tracking.py program can then successfully track green and red balls using generic gains. The final change that was implemented was to change the frame surrounding the ball to that of a circle to make full, accurate use of the center coordinates and radius value returned by the objcenter.py program.

The PID gains, however, required significant tuning to accurately track the balls. The tuning was performed for both the panning servo and the tilting servo by setting the Integral and Derivative gains to 0 and slowly increasing the Proportional gain until large, slow motions of the ball could be accurately tracked and just barely began oscillating; the gain was then set to half the gain value that barely began oscillating. The Integral gain was subsequently increased until the oscillations settled in approximately half a second. Finally, the Derivative gain was increased in much smaller increments than the other gains until rapid, jolting movements could be accurately tracked by the servos. The gains for both servos can be seen in Table 1.

These changes in the PID controls were accurately reflected in the performance of the pan-tilt assembly. Where initially the dynamic tracking of an object was delayed and the movement of the object being tracked would instigate jerks and choppy responses from the tracking apparatus. As the program continually failed to keep up with the movements of

the object, its error in trying to correct itself and maintain the logic in the algorithm was demonstrated through its rapid movements. The adjustments of the PID controller mitigated these responses. The PID controllers entire function is to work in a feedback loop to control what is being output based on the difference between process variable and the desired state. In this case these would equate to where the program believes the object to be and where the object actually is, respectively. The mitigation of this error implies less rapid movements and more measured responses to quickly moving objects that go out of frame.

Table 1: PID Gains

	Proportional	Integral	Derivative
Pan	0.05	0.24	0.0025
Tilt	0.04	0.31	0.003

4.3 Troubleshooting

There were several glitches and issues faced during the demonstration of this experiment.

Most of the issues faced in setting up the experiment were from the Linux terminal. While following the provided experimental procedures and tutorials to install an OpenCV, several libraries were determined to be incompatible with each other or generally deprecated. Release 3.7.0 of Python had to be used to construct the virtual environment in which the programs would be implemented. The relevant libraries were thus rolled back to earlier versions to be compatible with one another and to be compatible with Python 3.7.0. The main issue with compatibility arose with the Python Numpy library. As mentioned at the end of Subsection 3.2, version 4.1.0.25 was used in the experiment.

During the process of implementing the object tracking algorithm for an enabled motor, the instructions were followed such that the program was able to successfully detect and track a moving face. However, this program created rectangular enclosures on detected faces as opposed to circular ones, which are required by this experiment. To modify the original algorithm so that the camera would track circular objects, code was inserted directly into the objcenter.py program. The majority of the objcenter.py was replaced with a portion of the code from ball_tracking.py; the Haar Cascade Classifier model for facial detection in the objcenter.py class was replaced with code listed in Appendix A.

After incorporating this, the camera was able to detect a circular object of a specified color and enclose it with a circle as opposed to a rectangle. Within the main pan_tilt_tracking.py program, the coordinates returned by the modified objcenter.py program to create a circle around the tracked ball was implemented with:

```
if rect is not None:
    (x, y, radius) = rect
```

```
cv2.circle(frame, (int(x), int(y)), int(radius),
(0, 255, 0), 2)
```

4.4 Future Improvements

To improve the experiment for future trials, the apparatus can be placed in a more controlled environment. The camera would often track any object of a specified color from the ball_tracking.py rather than just the tennis ball. Performing the object tracking with a white or plain background can ensure that only the ball gets detected. Additionally, modification of the original instructions is required to successfully implement the virtual environments. The original setup that guided this experiment was agnostic towards the exact version of Python and its libraries that would work; the instructions were written in such a manner so as to make one believe that the versions of Python and its libraries would not matter and that they would immediately interface.

5 Conclusion

This experiment investigated the software and hardware involved in creating an apparatus using Raspberry Pi to track objects of specific colors in a video frame. Computer vision was used to track distinctly-colored balls as a demonstration of the possibilities of industry applications. It also provided a functional understanding of how to implement object tracking through a Raspberry Pi and test the coordination of motors using PID control to ensure the camera smoothly follows a moving object.

The performance of ball tracking without the enabling of motors was a direct result of the prepared program. Its functionality was satisfactory and only small adjustments in the bounds of the HSV color identifiers were necessary to apply the program to different colors. The performance of ball tracking with enabled motors also performed to reasonable standards. The adjustments in programming such that the logic of the pan_tilt_tracking.py code used HSV in relation to a circular image as an identifier rather than a pre-written facial recognition algorithm to locate the center of an object of interest. This allowed for the implementation of an algorithm that tracks objects out of frame and adjusts the camera to follow and was further refined through the tuned PID gains.

References

[1] "Cmyk vs rgb: What color space should i work in?."

[2] W. Dai and Q. Lin, "Raspberry pi object tracking experiment lecture," 2022.

[3] "Pid theory explained."

[4] "How servo motors work."

[5] "The right encoder for a servo motor."

[6] R. Teja, "What are the differences between raspberry pi and arduino?," Apr 2021.

[7] L. Pounder, "Raspberry pi vs arduino: Which board is best?," Jul 2020.

[8] A. Rosebrock, "Install opencv 4 on raspberry pi 4 and raspbian buster," Sep 2019.

[9] A. Rosebrock, "Ball tracking with opencv," Sep 2015.

[10] A. Rosebrock, "Pan/tilt face tracking with a raspberry pi and opencv," Apr 2019.

Contributions by section:

- Abstract: Axel, Bruno
- Introduction: Ashton, Axel, Arlene
- Theory: Axel, Ashton, Arlene, Christine
- Apparatus and Approach: Bruno, Sam, Anton, Christine, Axel
- Results and Discussion: Bruno, Arlene, Anton, Ashton
- Conclusions: Christine, Bruno, Anton, Arlene, Axel, Sam, Ashton
- Appendix: Bruno

A Appendix

Equations (1) through (6) are from the "Raspberry Pi Object Tracking Experiment" lab lecture.

Code from range-detector.py, the Python program used to identify color ranges as filters for static and dynamic camera tracking of differently colored balls:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# USAGE: You need to specify a filter and "only one" image source
#
# (python) range-detector --filter RGB --image /path/to/image.png
# or
# (python) range-detector --filter HSV --webcam

import cv2
import argparse
from operator import xor


def callback(value):
    pass


def setup_trackbars(range_filter):
    cv2.namedWindow("Trackbars", 0)

    for i in ["MIN", "MAX"]:
        v = 0 if i == "MIN" else 255

        for j in range_filter:
            cv2.createTrackbar("%s_%s" % (j, i),
            "Trackbars", v, 255, callback)


def get_arguments():
    ap = argparse.ArgumentParser()
    ap.add_argument('-f', '--filter', required=True,
                    help='Range filter. RGB or HSV')
    ap.add_argument('-i', '--image', required=False,
                    help='Path to the image')
    ap.add_argument('-w', '--webcam', required=False,
```

```
                    help='Use webcam', action='store_true')
    ap.add_argument('-p', '--preview', required=False,
                    help='Show a preview of the image after
                    applying the mask',
                    action='store_true')
    args = vars(ap.parse_args())

    if not xor(bool(args['image']), bool(args['webcam'])):
        ap.error("Please specify only one image source")

    if not args['filter'].upper() in ['RGB', 'HSV']:
        ap.error("Please speciy a correct filter.")

    return args


def get_trackbar_values(range_filter):
    values = []

    for i in ["MIN", "MAX"]:
        for j in range_filter:
            v = cv2.getTrackbarPos("%s_%s" % (j, i), "Trackbars")
            values.append(v)

    return values


def main():
    args = get_arguments()

    range_filter = args['filter'].upper()

    if args['image']:
        image = cv2.imread(args['image'])

        if range_filter == 'RGB':
            frame_to_thresh = image.copy()
        else:
            frame_to_thresh = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    else:
        camera = cv2.VideoCapture(0)
```

```
    setup_trackbars(range_filter)

    while True:
        if args['webcam']:
            ret, image = camera.read()

            if not ret:
                break

            if range_filter == 'RGB':
                frame_to_thresh = image.copy()
            else:
                frame_to_thresh = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        v1_min, v2_min, v3_min, v1_max, v2_max, v3_max =
        get_trackbar_values(range_filter)

        thresh = cv2.inRange(frame_to_thresh,
                 (v1_min, v2_min, v3_min),
                 (v1_max, v2_max, v3_max))

        if args['preview']:
            preview = cv2.bitwise_and(image, image, mask=thresh)
            cv2.imshow("Preview", preview)
        else:
            cv2.imshow("Original", image)
            cv2.imshow("Thresh", thresh)

        if cv2.waitKey(1) & 0xFF is ord('q'):
            break


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

Code from ball_tracking.py, the Python program used to statically track moving green and red balls; to specify which ball color to track, HSV ranges were applied to greenLower and greenUpper:

```
# import the necessary packages
from collections import deque
from imutils.video import VideoStream
import numpy as np
import argparse
import cv2
import imutils
import time
# construct the argument parse and parse the arguments
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 the lower and upper boundaries of the "green"
# ball in the HSV color space, then initialize the
# list of tracked points
greenLower = (29, 86, 80)
greenUpper = (64, 255, 255)
pts = deque(maxlen=args["buffer"])
# if a video path was not supplied, grab the reference
# to the webcam
if not args.get("video", False):
        vs = VideoStream(src=0).start()
# otherwise, grab a reference to the video file
else:
        vs = cv2.VideoCapture(args["video"])
# allow the camera or video file to warm up
time.sleep(2.0)
# keep looping
while True:
        # grab the current frame
        frame = vs.read()
        # handle the frame from VideoCapture or VideoStream
        frame = frame[1] if args.get("video", False) else frame
        # if we are viewing a video and we did not grab a frame,
        # then we have reached the end of the video
        if frame is None:
```

```
        break
# resize the frame, blur it, and convert it to the HSV
# color space
frame = imutils.resize(frame, width=600)
blurred = cv2.GaussianBlur(frame, (11, 11), 0)
hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
# construct a mask for the color "green", then perform
# a series of dilations and erosions to remove any small
# blobs left in the mask
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
# only proceed if at least one contour was found
if len(cnts) > 0:
        # find the largest contour in the mask, then use
        # it to compute the minimum enclosing circle and
        # centroid
        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"]))
        # only proceed if the radius meets a minimum size
        if radius > 10:
                # draw the circle and centroid on the frame,
                # then update the list of tracked points
                cv2.circle(frame, (int(x), int(y)), int(radius),
                        (0, 255, 255), 2)
                cv2.circle(frame, center, 5, (0, 0, 255), -1)
# update the points queue
pts.appendleft(center)
# loop over the set of tracked points
for i in range(1, len(pts)):
        # if either of the tracked points are None, ignore
        # them
        if pts[i - 1] is None or pts[i] is None:
```

```
                        continue
                # otherwise, compute the thickness of the line and
                # draw the connecting lines
                thickness = int(np.sqrt(args["buffer"]/float(i + 1)) *2.5)
                cv2.line(frame, pts[i - 1], pts[i], (0, 0, 255), thickness)
        # show the frame to our screen
        cv2.imshow("Frame", frame)
        key = cv2.waitKey(1) & 0xFF
        # if the 'q' key is pressed, stop the loop
        if key == ord("q"):
                break
# if we are not using a video file, stop the camera video stream
if not args.get("video", False):
        vs.stop()
# otherwise, release the camera
else:
        vs.release()
# close all windows
cv2.destroyAllWindows()
```

Code from pid.py, the Python program used to instantiate a PID control system:

```
import time
class PID:
        def __init__(self, kP=1, kI=0, kD=0):
                # initialize gains
                self.kP = kP
                self.kI = kI
                self.kD = kD

        def initialize(self):
                # intialize the current and previous time
                self.currTime = time.time()
                self.prevTime = self.currTime

                # initialize the previous error
                self.prevError = 0

                # initialize the term result variables
                self.cP = 0
                self.cI = 0
                self.cD = 0

        def update(self, error, sleep=0.2):
                # pause for a bit
                time.sleep(sleep)

                # grab the current time and calculate delta time
                self.currTime = time.time()
                deltaTime = self.currTime - self.prevTime

                # delta error
                deltaError = error - self.prevError

                # proportional term
                self.cP = error

                # integral term
                self.cI += error * deltaTime

                # derivative term and prevent divide by zero
                self.cD = (deltaError / deltaTime) if
                deltaTime > 0 else 0
```

```
# save previous time and error for the next update
self.prevTime = self.currTime
self.prevError = error

# sum the terms and return
return sum([
        self.kP * self.cP,
        self.kI * self.cI,
        self.kD * self.cD])
```

Code from the modified objcenter.py Python program which was used to identify the center of the ball and which returns the coordinates of the ball's center and a radius that surrounds the ball:

```
# import necessary packages
import imutils
import cv2

class ObjCenter:
    def __init__(self, frame, frameCenter):
        self.frame = frame
        self.frameCenter = frameCenter
    def update(self, frame, frameCenter):

        greenLower = (29, 86, 80)
        greenUpper = (64, 255, 255)

        blurred = cv2.GaussianBlur(frame, (11, 11), 0)
        hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
        # construct a mask for the color "green", then perform
        # a series of dilations and erosions to remove any small
        # blobs left in the mask
        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
        # only proceed if at least one contour was found
        if len(cnts) > 0:
            # find the largest contour in the mask, then use
            # it to compute the minimum enclosing circle and
            # centroid
            c = max(cnts, key=cv2.contourArea)
            ((x, y), radius) = cv2.minEnclosingCircle(c)
            M = cv2.moments(c)
            objx = int(M["m10"] / M["m00"])
            objy = int(M["m01"] / M["m00"])
            # only proceed if the radius meets a minimum size
            # if radius > 10:
```

```
            # draw the circle and centroid on the frame,
            # then update the list of tracked points
            #cv2.circle(frame, (int(x), int(y)), int(radius),
                #(0, 255, 255), 2)
        rects = (objx, objy, radius)

        return ((objx, objy), rects)

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

Code from the modified objcenter.py Python program that was not in the original objcenter.py program:

```
greenLower = (29, 86, 80)
greenUpper = (64, 255, 255)
blurred = cv2.GaussianBlur(frame, (11, 11), 0)
hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
# construct a mask for the color "green", then perform
# a series of dilations and erosions to remove any small
# blobs left in the mask
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
# only proceed if at least one contour was found
if len(cnts) > 0:
    # find the largest contour in the mask, then use
    # it to compute the minimum enclosing circle and
    # centroid
    c = max(cnts, key=cv2.contourArea)
    ((x, y), radius) = cv2.minEnclosingCircle(c)
    M = cv2.moments(c)
    objx = int(M["m10"] / M["m00"])
    objy = int(M["m01"] / M["m00"])
    # only proceed if the radius meets a minimum size
    # if radius > 10:
        # draw the circle and centroid on the frame,
        # then update the list of tracked points
        #cv2.circle(frame, (int(x), int(y)), int(radius),
            #(0, 255, 255), 2)
    rects = (objx, objy, radius)
    return ((objx, objy), rects)
# otherwise no faces were found, so return the center of the
# frame
return (frameCenter, None)
```

Code from pan_tilt_tracking.py Python program which was used to send commands to the pan-tilt assembly to move the camera to track the camera using tuned PID gains:

```
# USAGE
# python pan_tilt_tracking.py

# 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
import numpy as np

# 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((0,0), (0,0))

        # 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, radius) = rect
                        cv2.circle(frame, (int(x), int(y)), int(radius),
                        (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 independent PIDs
pan = manager.Value("i", 0)
tlt = manager.Value("i", 0)

# set PID values for panning
panP = manager.Value("f", 0.05)
panI = manager.Value("f", 0.24)
panD = manager.Value("f", 0.0025)

# set PID values for tilting
tiltP = manager.Value("f", 0.04)
tiltI = manager.Value("f", 0.31)
tiltD = manager.Value("f", 0.003)

# 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)
```