Week 2 Documentation: Maze Traversal

UCSD COSMOS 2024
Cluster 10: Robot Inventors

Michael Sun, Clarisse Arceo

July 19, 2024

Contents

Chapter 1

Theory

1.1 Purpose

The objective of this document is to clearly and formally outline the entire process of developing our maze solving robot. Such a document helps both the authors and instructors understand the idea and implementation robot, as well as our thought process to reach such functionality. Well-written documentation can provide insight into the entire design process rather than simply viewing the end result and abstracting away much of our struggles and early ideas.

1.2 The Challenge

The challenge that this document focueses on is to build a self-driving robot that will navigate a maze constructed with white walls and a floor of white paper. The robot must navigate the maze from a specified start to a specified finish using *only* the camera as the input device without external assistance of any kind. The robot that completes the maze in the shortest amount of time wins the competition.

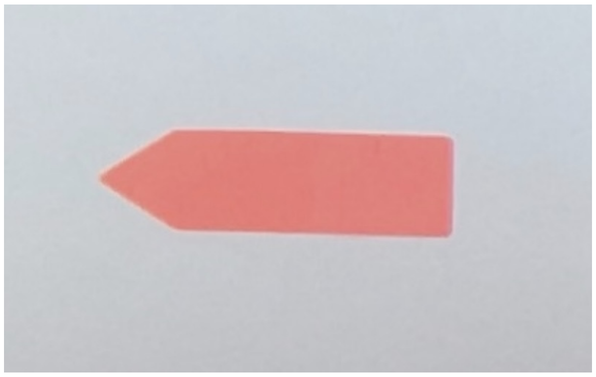
Teams will be provided a maximum of 20 sticky notes in the shape of an arrow to place around the maze to guide their robot. Each team will be given a total of seven minutes to set up arrows and run their robot. Within the allotted time, teams are *not* limited on the number of trials; robots can be ran as many times as needed. The timer may be paused if teams wish to modify their code. Teams may not see the maze until their competition time.

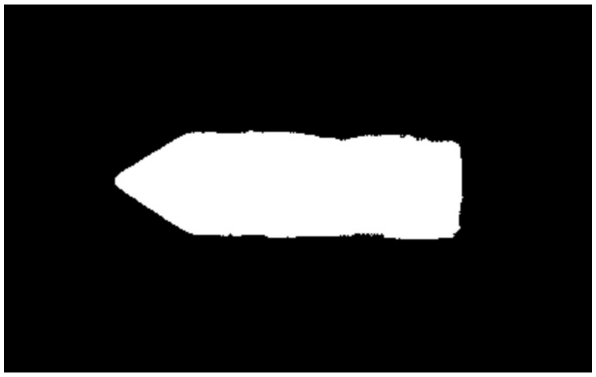
1.3 Preliminary Ideas

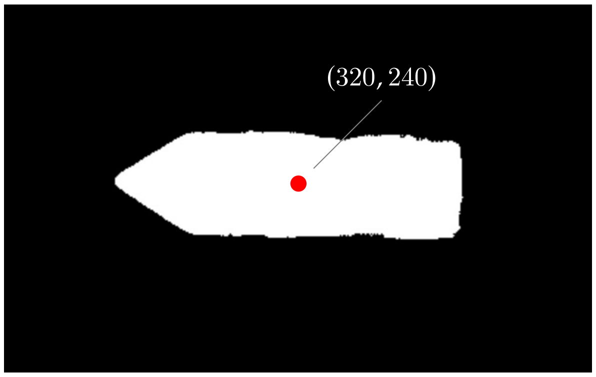
Our fundamental idea for the maze challenge is for the robot to move based on a repeating sequence of turns and going straight, placing sticky notes on the walls to give it a sense of direction and orientation. Consequentially, the camera must be mounted facing forward.

In the 7/12 lecture on OpenCV image processing, instructor Jonathan Van Hyning established a foundation for image processing and specifically emphasized on thresholding and extracting objects from an image. After his second lecture on 7/15, which demonstrated color thresholding, the first idea we thought of is to extract the color of an arrow, and based on a predefined mapping, turn left or right based on the detected color.

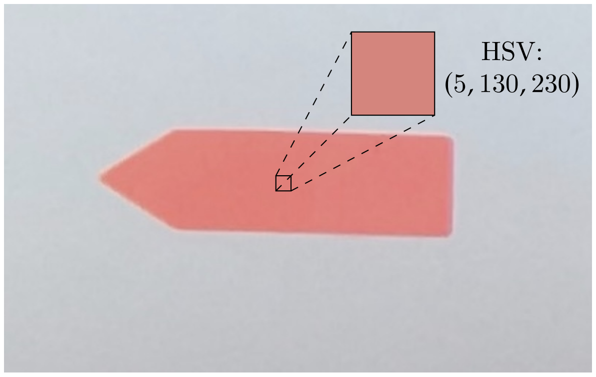
This idea seemed promising to us becuase the instructors had taught us color thresholding and centroid detection, which would give us access to the color of a pixel inside the arrow. (Looking back, we could have simply done a NumPy `count_nonzero()` to check for a color.) The steps of this idea is illustrated in Figure 1.1.


(a) Original image


(b) Color thresholded image



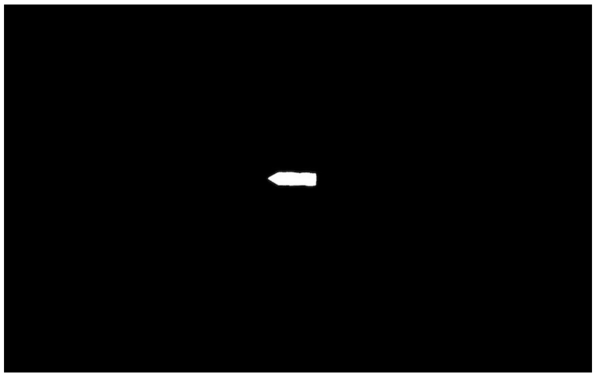
(c) The centroid (labeled with a red dot) computed from the binary image



(d) Extracted middle pixel using centroid coordinates to read HSV values of the arrow

Figure 1.1: Diagram of our hypothetical color extraction process

Another important function is to make the robot move forward until the arrow is sufficiently close. Since the arrow appears bigger on the frame when the camera is closer to it, Clarisse proposed to use pixel counts of a binary image to detect distance. The robot would move forward and continuously capture frames until a certain fraction of the thresholded image consists of white pixels.


(a) Thresholded image of an arrow that is not close enough. Very few pixels are white.

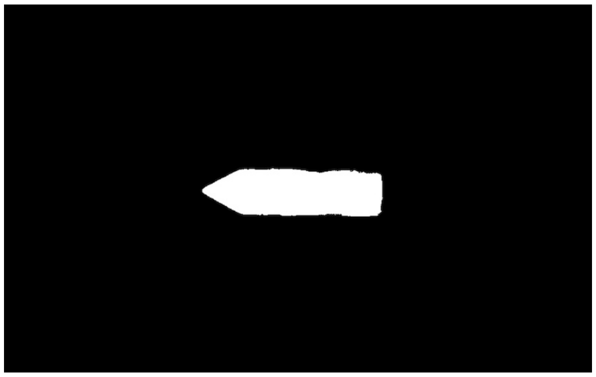

(b) Thresholded image of an arrow that is close enough. Approximately 1/25 of the pixels are white.

Figure 1.2: Images of arrows that are and are not close enough

The transition between the two scenarios above completes the idea of our robot's movement, which is how we are going to tell the robot to stop turning and start moving forward. Since we want the arrow to be in the center of the frame before moving, we quickly realized that the centroid can be used again. When the centroid, which represents the arrow's approximate center, is in the approximate center of the frame, the arrow will also be near the center.

Using these two propositions, we simulated the challenge from the camera's point of view and quickly discovered a problem with only using two colors:

Challenge 1.3.1

How will the robot differentiate between the current arrow it's looking at and the next arrow it should turn to? From our current process, both arrows will be thresholded into the same binary image.

Chapter 2

Revisions

Since there must be a distinction between the current and next arrow that is *independent* from any indications of direction, we decided to change our use of colors for differentiation. Because the arrows' shapes have a clear indication of direction, we can use the *shape* instead of color to tell the robot which way to turn. This idea also only uses two colors, which may reduce possible mistakes or innacuracies with color detection.

2.1 Arrow Differentiation

To distinguish between two arrows, they must be different colors, otherwise they will still be identical in the perspective of the robot. From this reasoning, we established an additional rule stating that **arrows must be placed alternating in color, with sequentially adjacent arrows never being the same color**.

To distinguish between the current arrow of color 1 and the next arrow of color 2, we would apply a color threshold to filter out one of the arrows. For example, after turning toward the centroid of an arrow of color 1 and moving forward until sufficiently near, the robot would read the direction it's pointing to and apply a color threshold that permits only color 2, entirely blocking out the arrow immediately in front of the robot. After applying the filter for color 2, the robot would be told to apply the filter for color 1.

2.2 Revising the Turning Algorithm

Our initial turning algorithm only considered the case where there will always be an object with contours and therefore a centroid to detect. However, there will be times where the robot will not see any colors and threshold the frame into only black pixels. To resolve this, the simplest and safest idea we thought of is to artifically set the centroid to the left or right based on the direction read from the last arrow if no contours are detected. Isolating the movement to only depend on the centroid—which is now *guarenteed* to exist—will simplify the logic for both turning and proportional control, which was our next supplementary idea.

2.3 Proportional Control

Since the PiCamera must process every frame of video, the robot must move slowly to avoid skipping or blurring frames. However, if we know that the centroid is far from the center of the frame, we can, ideally, increase the speed and slow down when the centroid is closer.

The code for proportional control and normalization of the PWM duty cycle is shown in Listing 2.1. Note that the code will be placed inside a `camera.continuous_capture()` loop. `XRES` is the x-value of the camera resolution.

```
1 # The difference between the x-value of the centroid and center of the frame
2 dist = abs(centroidx - XRES / 2)
3
4 # Change the PWM duty cycle proportional to the distance
5 drive.PWMA.ChangeDutyCycle(dist * 40 / XRES + 20)
6 drive.PWMB.ChangeDutyCycle(dist * 40 / XRES + 20)
```

Listing 2.1: PWM proportional control

Note:-

The following is a derivation of the duty cycle expression

$$D = \text{dist} \cdot \frac{40}{\text{XRES}} + 20$$

used in Listing 2.1 on lines 5 and 6.
Since we decided beforehand, it is given that the duty cycle will be normalized between 20 and 40 to have a meaningful speed decrease while not speeding or not generating enough torque. Formally, $D \in [20, 40]$ and $\text{dist} \in [0, XRES]$. We can use these upper and lower bounds to get our desired proportional control equation in the form of

$$D = k_p \cdot (\text{dist}) + c.$$

Plugging in the lower bound yields

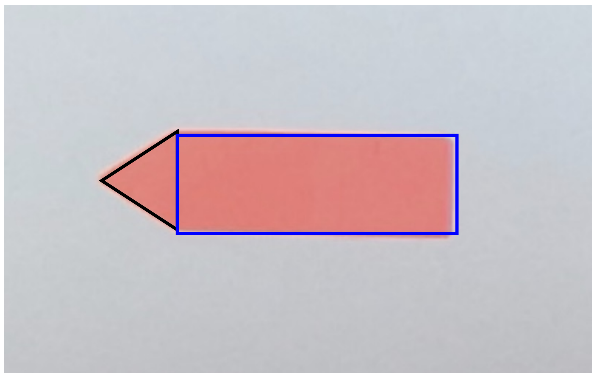
$$\begin{aligned} 20 &= k_p \cdot 0 + c \\ \Longrightarrow c &= 20. \end{aligned} \tag{2.1}$$

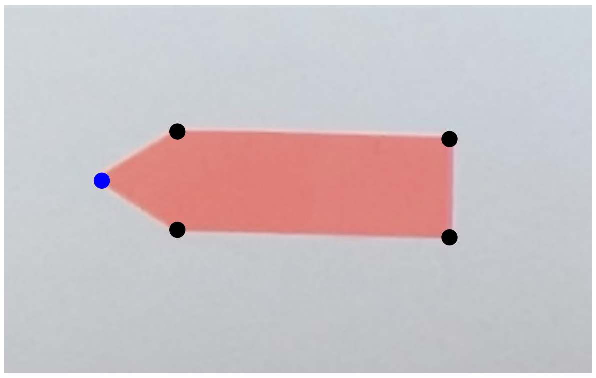
Plugging in the upper bound and using Equation (2.1), we get

$$\begin{aligned} 40 &= k_p \cdot \frac{\text{XRES}}{2} + 20 \\ \Longrightarrow k_p &= \frac{40}{\text{XRES}} \\ \Longrightarrow &\boxed{D = \text{dist} \cdot \frac{40}{\text{XRES}} + 20}. \end{aligned} \tag{2.2}$$

2.4 Detecting an Arrow's Direction

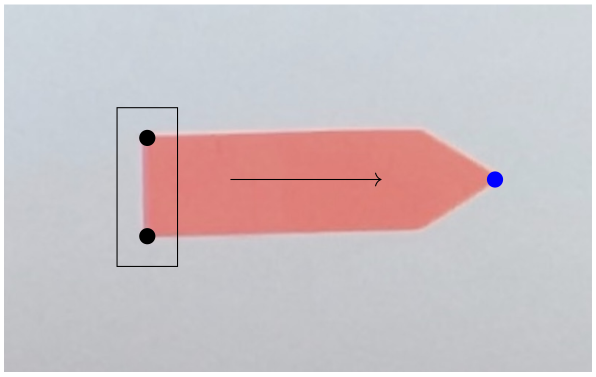
The arrows for this challenge are essentiallly a rectangle with a triangle on one side to determine direction. If we can find the position of the vertex that is not shared with the rectangle relative to the other points, we can determine which direction the arrow is pointing in.


(a) An arrow segmented into its defining triangle and rectangle


(b) Vertices of the arrow, with the defining vertex highlighted in blue

We both observed two sets of two points with x-values that are almost identical, and that the directional vertex (highlighted blue in Figure 2.1b) has a distinct x-value. Also, it is guarenteed that the blue vertex will always have the minimum or maximum x-value. Therefore, if there are two points containing the maximum x-value, the arrow points to the left because the blue vertex must be on the two points' left side. Similarly, if there is only one vertex whose x-value is the maximum, the arrow will point right.

Using a shape's vertices, naturally, leads us to corner detection. We decided to use the Shi-Tomasi Corner Detector mainly because the number of detected corners can be limited to a specific number. In this case, we chose to detect only three vertices, which are the two right angles and the directional angle.


(a) Right-pointing arrow

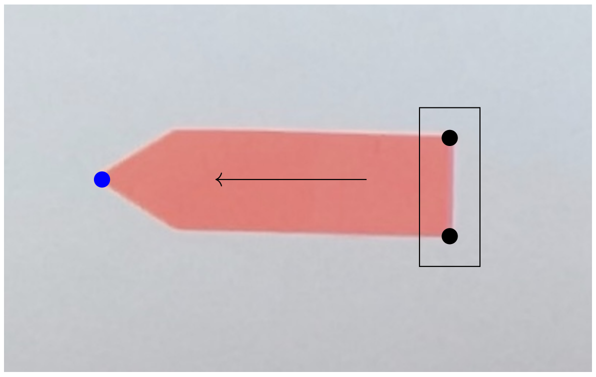

(b) Left-pointing arrow

Figure 2.2: Corner layouts of arrows going left and right

Additional details part of accurate corner detection include:

- Filtering the frame with a Gaussian blur before thresholding to smooth the image
- Filtering the thresholded image with a small median blur to smooth jagged edges while still protecting corners
- Increase the block size to only detect the big corners and ignore edge roughness.

2.5 Pseudocode

The following is the rough pseudocode for our strategy.

Algorithm 1: Maze Challenge Strategy

```
Set up camera;
dir ← RIGHT;
curcolor ← COLOR1;
Turn in the direction of dir;
while 1 do
    /* Turn to dir */
    while frame ← capture video do
        bin ← color threshold on frame using curcolor;
        contours ← of bin;
        if there are contours then
            cx ← the x-value of the centroid of the contours;
            if cx is within 40 pixels of the middle of the frame then
                Continue;
        else
            cx ← 0 or the width of the frame depending on dir;
        end
        Set the PWM duty cycle proportionally between 20 and 40. See Listing 2.1;
    end
    /* Move until close enough */
    Drive forward;
    while frame ← capture video do
        bin ← color threshold on frame using curcolor;
        if more than 1/20 of the pixels of bin are white then
            arrowpic ← bin;
            Break;
    end
    dir ← get the direction in arrowpic;
    Toggle curcolor;
end
```

2.6 Arrow Placement Example

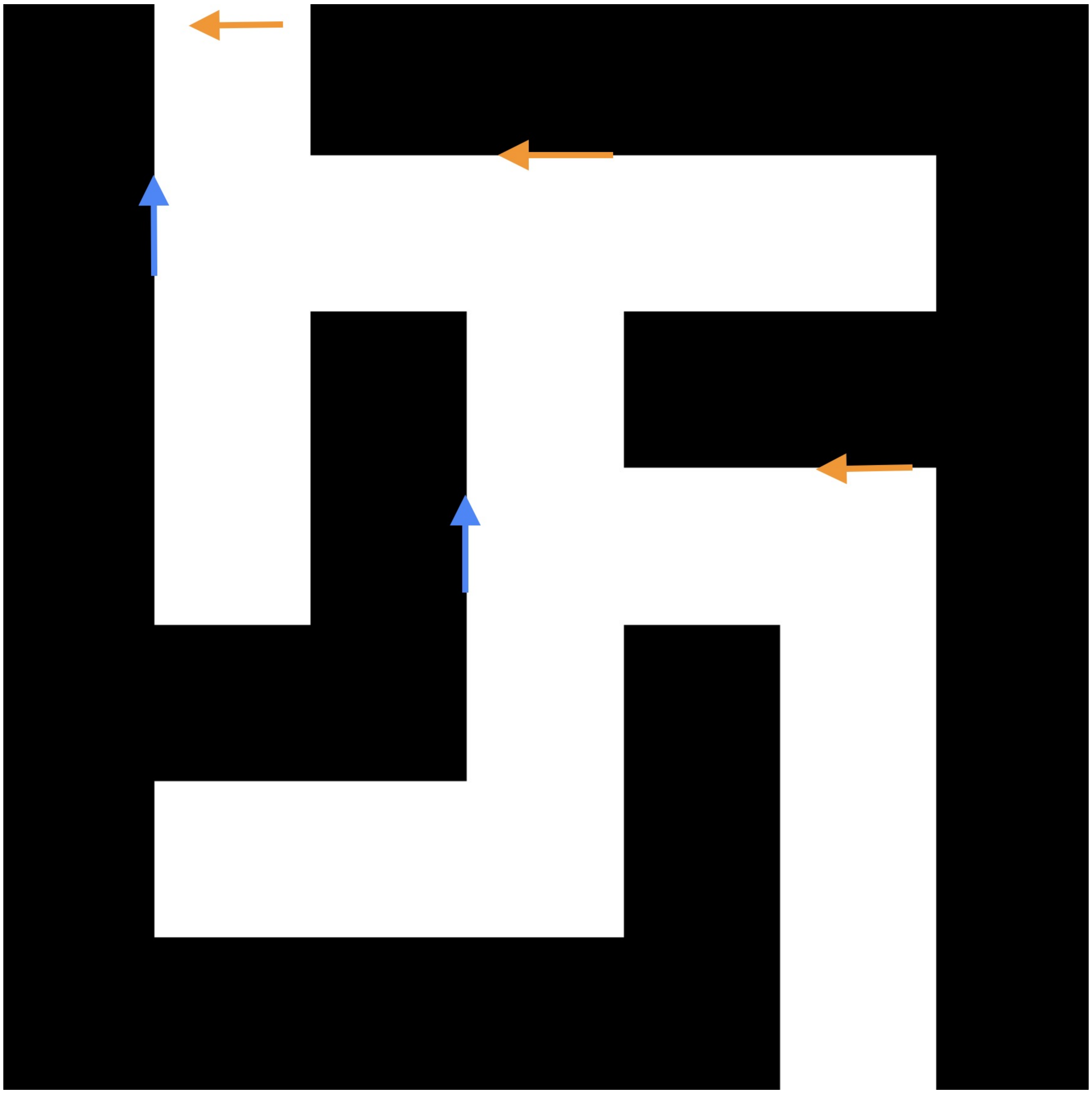
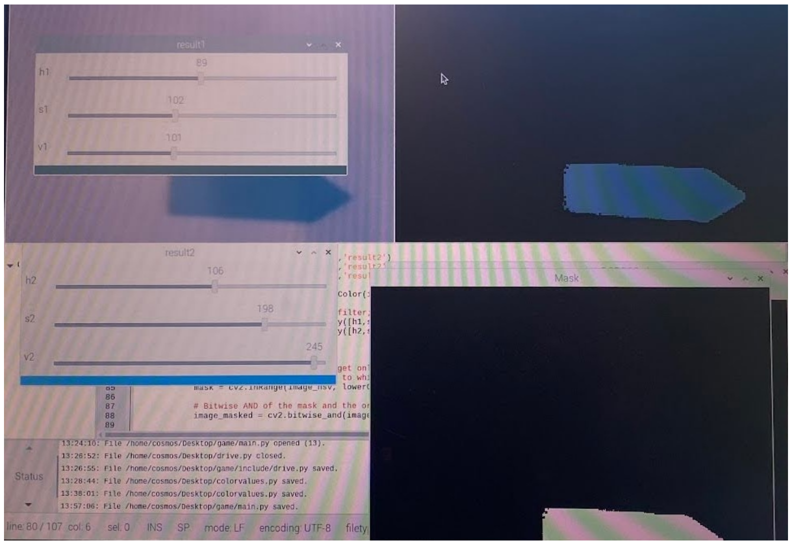


Figure 2.3: A maze marked with the arrows that would be placed to guide the robot using our strategy. Black squares represent walls and white squares represent paths. The start and end is the bottom left and top right, respectively.

Chapter 3

Testing

We used the file `camera07_detectcolortest_cv2proc.py` provided in the sample code to find the upper and lower bounds for the thresholds of both colors (shown in the figure below). We also selected to use orange and blue for our two differentiation colors since they are relatively far apart on the HSV scale.

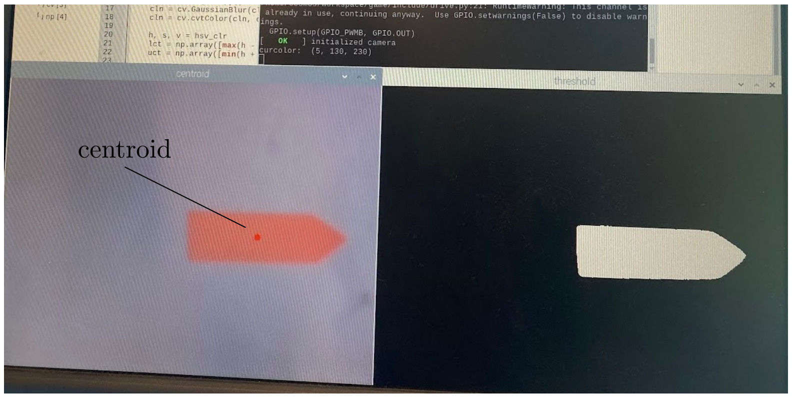


```
    mask = cv2.inRange(image_hsv, lower

    # Bitwise AND of the mask and the or
    image_masked = cv2.bitwise_and(image
```

```
13:24:10: File /home/cosmos/Desktop/game/main.py opened (13).
13:26:52: File /home/cosmos/Desktop/drive.py closed.
13:26:55: File /home/cosmos/Desktop/game/include/drive.py saved.
13:28:44: File /home/cosmos/Desktop/colorvalues.py saved.
13:38:01: File /home/cosmos/Desktop/colorvalues.py saved.
13:57:06: File /home/cosmos/Desktop/game/main.py saved.
```

```
line: 80 / 107  col: 6   sel: 0   INS   SP   mode: LF   encoding: UTF-8
```

We then tested the centroid detection, thresholding, and filter switching, which made us notice and fix a few minor errors with the code. The terminal output of this test is shown in Figure 3.1.



```
    cln = cv.GaussianBlur(c
    cln = cv.cvtColor(cln, c

    h, s, v = hsv_clr
    lct = np.array([max(h -
    uct = np.array([min(h +
```

```
 already in use, continuing anyway.  Use GPIO.setwarnings(False) to disable warnings.
  GPIO.setup(GPIO_PWMB, GPIO.OUT)
[   OK   ] initialized camera
curcolor:  (5, 130, 230)
```

Figure 3.1: The camera's POV during centroid detection and thresholding for the orange sticky note. The centroid is labeled in black.

3.1 Run Tests

Since we have only tested our code in an ideal situation without many external influences or motor failures, we had to modify our code a large amount after our first test inside the maze. Most of our problems stemmed from one of two possible core issues: The robot turns way too fast and has imbalanced motors.

If the robot turns too fast, we will not be able to take stable photos and all of our image detection will fail. The frame rate will also drop because OpenCV image processing takes time. However, if we decrease the duty cycle too much, the robot will lose torque as well as speed, which will cause it to not move the robot. Our final but still suboptimal solution is to move in slow pulses and follow a cycle of

$$\text{turn} \longrightarrow \text{stop} \longrightarrow \text{capture photo} \longrightarrow \text{process photo} \longrightarrow \text{repeat}$$

Instead of continuously capturing video with a `time.sleep()` inside the loop, we decided to only capture photos in hopes of improving quality. This would also ensure that the captured frames are in sync with motor pauses.

The other major issue is that the strength of the right motor is noticably weaker than the left, so we had to increase the duty cycle of the right wheel to maintain a straight forward motion. Without matching the wheel offset, the robot would veer off track from the arrow and crash into the wall next to it.

In addition to the core issues, we realized that the camera would detect multiple sets of contours if there are multiple isolated blobs of white in a binary image. We combatted this by using the first one in the list because it appears to be the strongest.

Note:-

Michael investigated this further with a concrete example. For every list of contours, they will be drawn onto the image in their default order with the brightness of the drawings decreasing with every increase in index. The test results are as follows, which reinforces the idea that using the first contour has a higher chance of picking the arrow.

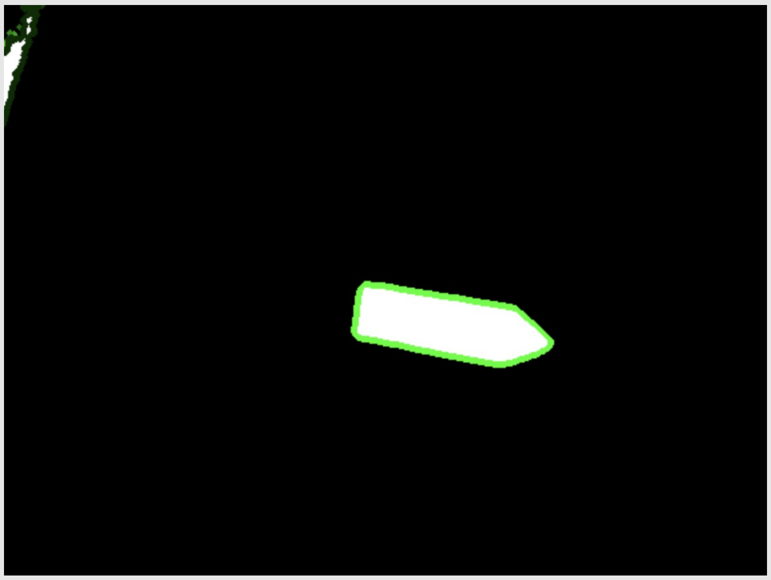


Figure 3.2: The image with contours drawn on it. The border around the arrow is a very bright shade of green while the noise in the top left are much darker.

The code for this test can be found in the chapter 5 of this document.

Much of our testing process was playing around with certain values (e.g. duty cycle, white pixel count)

Chapter 4

Final Strategy and Code Explanations

This chapter is a summary of our final strategy, showing the Python code we used for each step. Our code is split into four files in total:

- Driver code `main.py`
- Module file for movement: `drive.py`
- Module file for corner detection: `include/corners.py`
- Module file with color interfaces: `include/color.py`

Note:-

The following code snippets are used to illustrate the concept and may exclude unimportant details. Please refer to chapter 5 for complete code.

The first step inside the loop is to pulse the motors, shown below.

```python
# main.py

while True:
    sleep(0.5)  # sleep half a second to wait for the camera to stablize

    camera.capture(rawframe, format="bgr")
    frame = rawframe.array  # convert to numpy array

    # the right duty cycle is larger to counteract the weak motor
    if direc == Dir.LT:
        drive.fwd(60, -80)
    else:
        drive.fwd(-60, 80)

    # turn the motors on for 0.1 seconds -- a "pulse"
    sleep(0.1)
    drive.stop()
```

Now we apply the threshold, and find and process the centroid.

```python
    bin = color_thresh(frame, curcolor)  # color threshold
    contours, _ = cv.findContours(bin, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)

    centroidx = XRES  # set the default centroid x-value to be the frame's width
    if contours:  # arrow is in the frame, find centroid
        # use the first (biggest/strongest) contour if there are multiple
        mmt = cv.moments(contours[0])
        if mmt["m00"] == 0:
            rawframe.truncate(0)
            continue
        centroidx = int(mmt["m10"] / mmt["m00"])

        # if the centroid is within the 60 middle pixels of the frame
        if centroidx > XRES / 2 - 30 and centroidx < XRES / 2 + 30:
            rawframe.truncate(0)
            break

    elif direc == Dir.LT:  # blank frame (no arrow), set centroid to edge
        centroidx = 0
```

At this point, we will have broken out of the loop and must start moving forward and capturing frames again.

```
# main.py

# move forward until close enough
drive.fwd(40, 50)
for rframe in camera.capture_continuous(
    rawframe, format="bgr", use_video_port=True
):
    frame = rframe.array

    # color threshold the image again to extract black and white pixels
    bin = color_thresh(frame, curcolor)

    # if more than 1/20 of the pixels in the binary image are white
    if np.count_nonzero(bin) > XRES * YRES // 20:
        arrowpic = bin
        print("close enough, break loop")
        rawframe.truncate(0)
        break
```

Finally, we need to read the next direction and toggle curcolor;

```
# get direction of the arrow
direc = getdir(arrowpic)

# filter again
curcolor = COLOR2_HSV if curcolor == COLOR1_HSV else COLOR1_HSV
```

If we examine `include/color.py`, we see the following code

```
# HSV values of colors 1 and 2
COLOR1_HSV = (5, 130, 230)    # orange-red
COLOR2_HSV = (100, 130, 160)  # blue


# an enum for direction
class Dir(Enum):
    LT = 0
    RT = 1


# applies an HSV color threshold
def color_thresh(img_bgr, hsv_clr):
    cln = img_bgr.copy()
    cln = cv.GaussianBlur(cln, (7, 7), sigmaX=0, sigmaY=0)  # blur/smooth the image
    cln = cv.cvtColor(cln, cv.COLOR_BGR2HSV)

    h, s, v = hsv_clr
    lct = np.array([max(h - 20, 0), s - 90, v - 80])  # lower bound
    uct = np.array([min(h + 20, 255), 255, 255])      # upper bound

    return cv.inRange(cln, lct, uct)
```

Now if we look at `include/corners.py`, we have the code to detect the direction of an arrow.

```
# main callable function
# accepts a BINARY image
def getdir(img):
    corners = getcorners(img)       # returns a numpy array of corners
    assert corners.shape == (3, 2)  # asserrt three arrays of 2 elements (y and x)

    xmax, _ = (np.max(corners, axis=0)).ravel()  # maximum x value in the array

    # arrow is guarenteed to be horizontal
    # a boolean array representing which cells within 10 pixels of xmax is a pair
    bool_arr = corners[:, 0] > xmax - 10
    if np.count_nonzero(bool_arr) == 2:  # if there is a pair, return left
        return Dir.LT
    else:  # no pair, return right
        return Dir.RT

# technical and more important code
def getcorners(img):
    bin = cv.medianBlur(img, 5)

    # shi-tomasi corner detection
    # relatively high tolerance and high block size to read big and strong corners
    corners = cv.goodFeaturesToTrack(bin, 3, 0.2, 40, blockSize=11)

    # convert to 3x2 numpy 2d array
    arr = np.empty((3, 2))
    it = 0
    for i in corners:
        x, y = i.ravel()
        arr[it][0] = x
        arr[it][1] = y
        it += 1

    return arr
```

4.1 Lingering Issues

Although we had a significant amount of time to work on the robot, we were still not able to resolve some issues. Below are some current and hypothetical issues that still remain to be solved.

- Arrows must be placed on walls, which could make it difficult for a robot to move through long corridors. The arrow must be placed at the end of the hall, so it may appear too small to be properly detected.
- With slow pulsing, the motors are inconsistent and do not move at a uniform speed.
- The weak motor may stagger and cause the robot to not move straight. An unaligned camera may cause the arrow to leave the frame while counting white pixels and cause the robot to crash into the wall.
- The captured images are still slightly blurred, which causes our direction detection procedure to be very inconsistent.

Chapter 5

Appendix

5.1 main.py

```python
from time import sleep

import cv2 as cv
import picamera
import picamera.array

import include.drive as drive
from include.color import *
from include.corners import *

XRES = 640
YRES = 480
CAM_RES = (XRES, YRES)

camera = picamera.PiCamera()
camera.resolution = CAM_RES
camera.framerate = 60

rawframe = picamera.array.PiRGBArray(camera, size=CAM_RES)

try:
    sleep(1)
    print("[  \033[32;1mOK\033[0m  ] initialized camera")

    direc = Dir.RT
    curcolor = COLOR1_HSV
    while True:
        print("curcolor: ", curcolor)

        arrowpic = None

        # capture photos in a loop
        while True:
            sleep(0.5)  # sleep for half a second to wait for the camera to stablize

            camera.capture(rawframe, format="bgr")
            frame = rawframe.array

            if direc == Dir.LT:
                drive.fwd(60, -80)
            else:
                drive.fwd(-60, 80)

            sleep(0.1)  # run the motors for 0.1 seconds
            drive.stop()

            bin = color_thresh(frame, curcolor)
            contours, _ = cv.findContours(bin, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)

            centroidx = XRES  # set the default centroid x-value to be the frame's width
            if contours:  # arrow is in the frame, find centroid
                # use the first (biggest/strongest) contour if there are multiple
                mmt = cv.moments(contours[0])
                if mmt["m00"] == 0:
                    rawframe.truncate(0)
                    continue
                centroidx = int(mmt["m10"] / mmt["m00"])

                # if the centroid is within the 60 middle pixels of the frame
                if centroidx > XRES / 2 - 40 and centroidx < XRES / 2 + 40:
                    rawframe.truncate(0)
                    print("centroid near center, break loop")
                    break

            elif direc == Dir.LT:  # blank frame (no arrow), set centroid to edge
                    centroidx = 0

            # draw the centroid
            cv.circle(frame, (centroidx, YRES // 2), 5, (0, 0, 255), -1)

            cv.imshow("threshold", bin)
            cv.imshow("centroid", frame)
            cv.waitKey(1)

            rawframe.truncate(0)

        cv.destroyAllWindows()

        # move forward until close enough
        drive.fwd(40, 50)
        for rframe in camera.capture_continuous(
            rawframe, format="bgr", use_video_port=True
        ):
            frame = rframe.array

            # color threshold the image again to extract black and white pixels
            bin = color_thresh(frame, curcolor)
            if np.count_nonzero(bin) > XRES * YRES // 20:
                arrowpic = bin
                print("close enough, break loop")
                rawframe.truncate(0)
                break

            cv.imshow("frame", frame)
            cv.imshow("threshold", bin)
            cv.waitKey(1)

            rawframe.truncate(0)

        direc = getdir(arrowpic)  # get direction of the arrow

        # filter again
        curcolor = COLOR2_HSV if curcolor == COLOR1_HSV else COLOR1_HSV

        cv.destroyAllWindows()

except KeyboardInterrupt:
    rawframe.truncate(0)
    cv.destroyAllWindows()
    camera.close()
```

5.2 include/drive.py

```python
import sys

import RPi.GPIO as GPIO

# define GPIO pins
GPIO_AIN1 = 17
GPIO_AIN2 = 27
GPIO_PWMA = 22
GPIO_PWMB = 13
GPIO_BIN1 = 5
GPIO_BIN2 = 6

GPIO.setmode(GPIO.BCM)  # use BCM pin numbers

# set pins as output
GPIO.setup(GPIO_AIN1, GPIO.OUT)
GPIO.setup(GPIO_AIN2, GPIO.OUT)
GPIO.setup(GPIO_PWMA, GPIO.OUT)
GPIO.setup(GPIO_BIN1, GPIO.OUT)
GPIO.setup(GPIO_BIN2, GPIO.OUT)
GPIO.setup(GPIO_PWMB, GPIO.OUT)

# stop the motors on init
GPIO.output(GPIO_AIN1, False)
GPIO.output(GPIO_AIN2, False)
GPIO.output(GPIO_BIN1, False)
GPIO.output(GPIO_BIN2, False)

PWM_FREQ = 50

PWMA = GPIO.PWM(GPIO_PWMA, PWM_FREQ)
PWMB = GPIO.PWM(GPIO_PWMB, PWM_FREQ)

PWMA.start(100)
PWMB.start(100)


def pwm_cleanup():
    PWMA.stop()
    PWMB.stop()


def stop(msg=None):
    GPIO.output(GPIO_AIN1, False)
    GPIO.output(GPIO_AIN2, False)
    GPIO.output(GPIO_BIN1, False)
    GPIO.output(GPIO_BIN2, False)
    if msg != None:
        print(msg)


def fwd(bdc, adc, msg=None):
    if adc < -100 or bdc < -100 or adc > 100 or bdc > 100:
        sys.stderr.write("PWM duty cycle must be in the range [-100,100]")
        return

    PWMA.ChangeDutyCycle(abs(adc))
    PWMB.ChangeDutyCycle(abs(bdc))

    apos = adc > 0
    bpos = bdc > 0
    GPIO.output(GPIO_AIN1, apos)
    GPIO.output(GPIO_AIN2, not apos)
    GPIO.output(GPIO_BIN1, bpos)
    GPIO.output(GPIO_BIN2, not bpos)

    if msg != None:
        print(msg)
```

5.3 include/color.py

```python
from enum import Enum

import cv2 as cv
import numpy as np

# HSV values of colors 1 and 2
COLOR1_HSV = (5, 130, 230)  # orange-red
COLOR2_HSV = (95, 130, 160) # blue


class Dir(Enum):
    LT = 0
    RT = 1

# applies an HSV color threshold
def color_thresh(img_bgr, hsv_clr):
    cln = img_bgr.copy()
    cln = cv.GaussianBlur(cln, (7, 7), sigmaX=0, sigmaY=0)
    cln = cv.cvtColor(cln, cv.COLOR_BGR2HSV)

    h, s, v = hsv_clr
    lct = np.array([max(h - 20, 0), s - 90, v - 80])  # lower bound
    uct = np.array([min(h + 20, 255), 255, 255])       # upper bound

    return cv.inRange(cln, lct, uct)
```

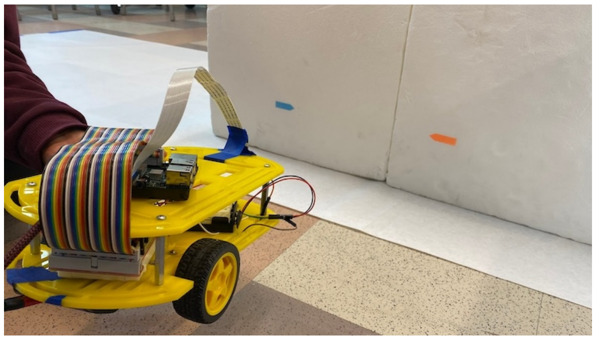
5.4 include/corners.py

```python
import cv2 as cv
import numpy as np
from color import *


def getcorners(img):
    bin = img.copy()
    cv.imshow("threshold", bin)
    cv.waitKey()

    bin = cv.medianBlur(bin, 5)
    cv.imshow("blurred", bin)
    cv.waitKey()

    # shi-tomasi corner detector
    corners = cv.goodFeaturesToTrack(bin, 3, 0.2, 40, blockSize=11)

    # convert to a 3x2 numpy 2d array
    arr = np.empty((3, 2))
    it = 0
    for i in corners:
        x, y = i.ravel()
        arr[it][0] = x
        arr[it][1] = y
        it += 1

    return arr


# accepts a BINARY image
def getdir(img):
    corners = getcorners(img)  # returns a numpy array of corners
    assert corners.shape == (3, 2)  # assert three arrays of 2 elements [y,x]

    xmax, _ = (np.max(corners, axis=0)).ravel()  # max x-value in the array

    # arrow is guarenteed to be horizontal
    # a boolean array representing which cells within 10 pixels of xmax is a vpair
    bool_arr = corners[:, 0] > xmax - 10
    if np.count_nonzero(bool_arr) == 2:  # if there is a vpair, return left
        return Dir.LT
    else:  # no pair means return right
        return Dir.RT
```

5.5 test.py

```
import cv2 as cv

img = cv.cvtColor(cv.imread("2contours.jpg"), cv.COLOR_BGR2GRAY)
_, img = cv.threshold(img, 127, 255, 0)

contours, hierarchy = cv.findContours(img, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)

img_clr = cv.cvtColor(img, cv.COLOR_GRAY2BGR)
it = 0
for c in contours:
    cv.drawContours(img_clr, contours, it, (0, 255 - it * 30, 0), 3)
    print(f"\ncontour {it}:\n{c}")
    it += 1

print("\nhierarchy (RETR_EXTERNAL):\n", hierarchy)

_, hierarchy = cv.findContours(img, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)
print("\nhierarchy (RETR_LIST):\n", hierarchy)

_, hierarchy = cv.findContours(img, cv.RETR_CCOMP, cv.CHAIN_APPROX_SIMPLE)
print("\nhierarchy (RETR_CCOMP):\n", hierarchy)

_, hierarchy = cv.findContours(img, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
print("\nhierarchy (RETR_TREE):\n", hierarchy)

cv.imshow("image", img_clr)
cv.waitKey()
```

5.6 Photos


(a) The PiCamera being held up to both arrow colors capturing frames

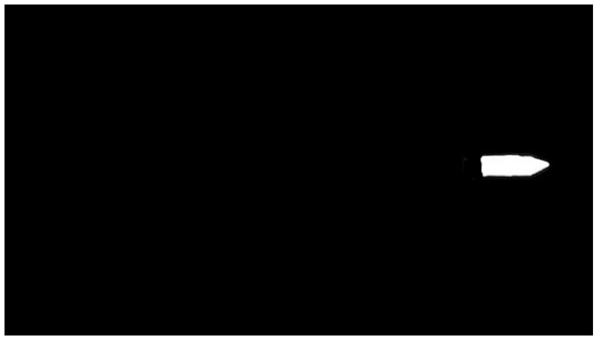

(b) The orange-thresholded frame captured by the camera (cropped to remove noise from the orange floor tile)

Figure 5.1: Testing the color filters of the robot