(i) a specification of the auto-exploring decision logic.

At every intersection, my brain logic follows a tiered decision strategy:

1. Check for any unknown streets at the current intersection (i.e. STATUS.UNKNOWN).If any are found, we prioritize turning toward the smallest heading difference (least rotation) to explore them.
2. If all streets are known but some are marked UNEXPLORED, turn toward and drive down the unexplored path to investigate it.
3. If all directions are either NONEXISTENT or already EXPLORED:
    a. Use Dijkstra's algorithm to find the nearest intersection with unknown or unexplored streets.
    b. Set that location as the goal.
    c. Let the robot auto-drive to that location using the stored optimal direction fields in each Intersection

(ii) a code specification/API/flowchart for the new ultrasound elements and behaviors.

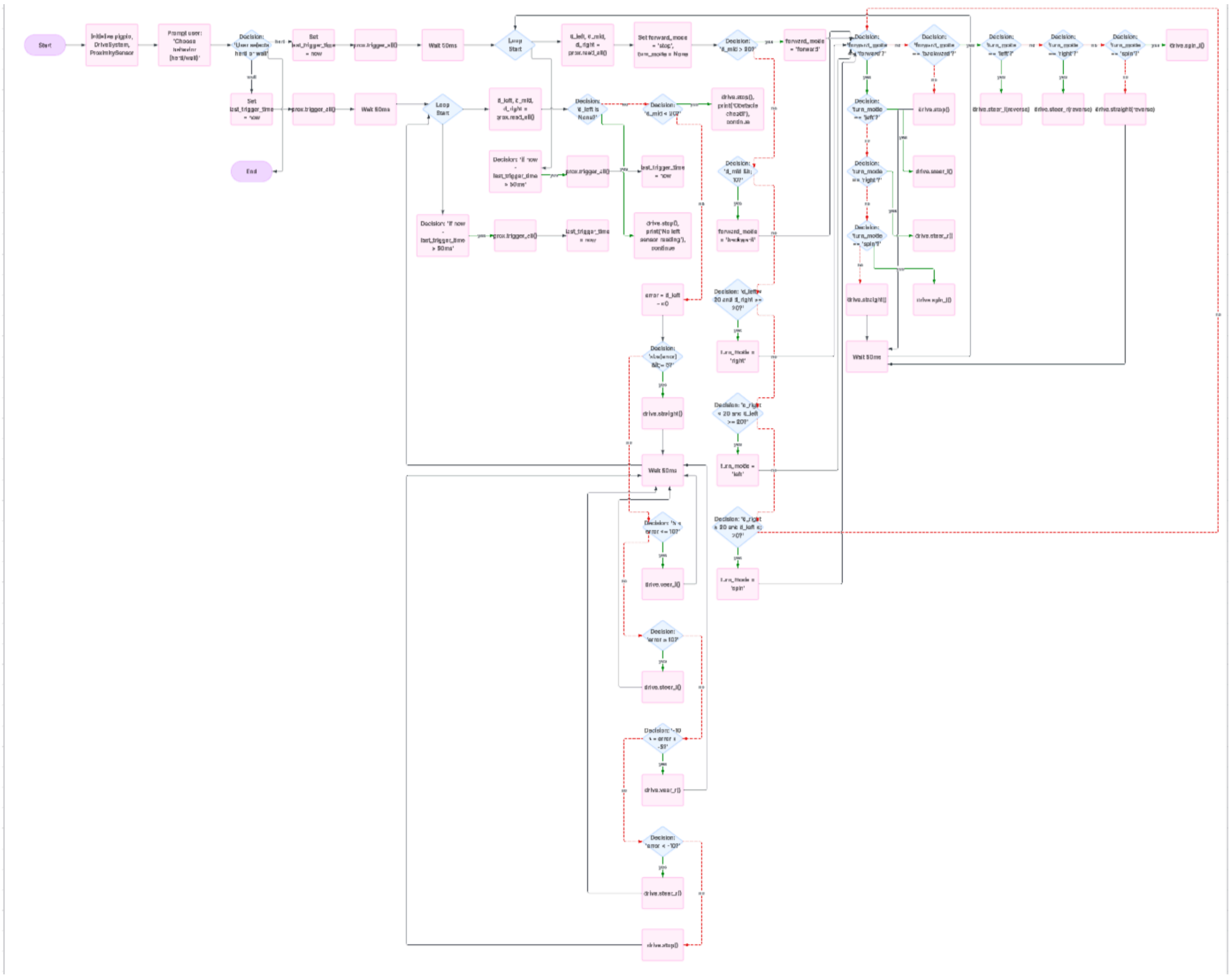
class Ultrasound(io, pintrig, pinecho)

- __init__(self, io, pintrig, pinecho):
    - Io: pigpio instance
    - Pintrig: GPIO pin for trigger
    - Pinecho: GPIO pin for echo
- trigger():
    - Triggers the sensor to send an ultrasonic pulse.
- rising(pin, level, ticks)
    - Callback for the rising edge of the echo pin. Records the timestamp when the echo starts.
    - Usage: Called automatically by pigpio.
- falling(pin, level, ticks)
    - Callback for the falling edge of the echo pin. Calculates the time-of-flight and computes the distance using your measured speed of sound.
- read() -> float or None:
    - Returns the last measured distance (in cm), or None if no measurement.
- read_delta_t() -> int or None:
    - Returns the last measured time-of-flight (in microseconds), or None if no measurement.

class ProximitySensor(io)

- __init__(self,)
    - Initialzies all the pins for each ultrasonic sensor
- trigger_all()
    - Triggers left, middle, and right sensor
- read_all() -> tuple
    - Returns a tuple of the last measured distances from all sensors.
- read_all_delta_t() -> tuple

- Returns a tuple of the last measured time-of-flights from all sensors.


Start
Initialise pigpio, DriveSystem, ProximitySensor
Prompt user: 'Choose behavior (herd/wall)'
Decision: 'User selects herd or wall'
herd
wall
Set last_trigger_time = now
prox.trigger_all()
Wait 50ms
Loop Start
d_left, d_mid, d_right = prox.read_all()
End
Decision: 'If now - last_trigger_time > 50ms'
yes
Decision: 'd_left is None?'
Set forward_mode = 'stop', turn_mode = None
Decision: 'd_mid > 20?'
forward_mode = 'forward'
Decision: 'forward_mode == 'forward'?'
Decision: 'forward_mode == 'backward'?'
Decision: 'turn_mode == 'left'?'
Decision: 'turn_mode == 'right'?'
Decision: 'turn_mode == 'spin'?'
drive.spin_l()
drive.stop()
drive.steer_l(reverse)
drive.steer_r(reverse)
drive.straight(reverse)
Decision: 'd_mid < 20?'
drive.stop(), print('Obstacle ahead!'), continue
Decision: 'd_mid ≤ 10?'
forward_mode = 'backward'
drive.stop(), print('No left sensor reading'), continue
last_trigger_time = now
drive.steer_l()
drive.steer_r()
drive.straight()
drive.spin_l()
error = d_left - 40
Decision: 'abs(error) ≤ 5?'
drive.straight()
Wait 50ms
Decision: 'd_left < 20 and d_right >= 20?'
turn_mode = 'right'
Decision: 'd_right < 20 and d_left >= 20?'
turn_mode = 'left'
Decision: 'd_right < 20 and d_left < 20?'
turn_mode = 'spin'
Decision: '5 < error <= 10?'
drive.veer_l()
Decision: 'error > 10?'
drive.steer_l()
Decision: '-10 <= error < -5?'
drive.veer_r()
Decision: 'error < -10?'
drive.steer_r()
drive.stop()
yes
no

(iii) the data/graphs from the ultrasound testing (Section 5).

5.1 Speed of Sound

Fixed Distance (cm)	time of flight (ms)		
10	590		
30	1745		
50	2905		
100	5785		
200	11670		

Fixed Distance (multiplied by 2) (cm)	time of flight (s)	speed cm/s	speed m/s
20	0.00059	33898.30508	338.9830508
60	0.001745	34383.95415	343.8395415
100	0.002905	34423.40792	344.2340792
200	0.005785	34572.1694	345.721694
400	0.01167	34275.92117	342.7592117
	average	343.1075155	m/s
	linear regression	342.9871065	m/s

5.2 Measuring Angle

(a) Sliding to the Right of the Robot:
   (i) arccos(51/66) = 39.4
   (ii) arccos(51/67) = 40.43

(b) How far from "perfectly-aligned" can an obstacle be?
   (i) 90 - 52 = 38 degrees

The two angles approximately agree

5.3 Minimum and Maximum

(a) Max: 306.8 cm
(b) Min: 2.29 cm

(iv) the bands/actions/feedback constants used in the wall following (Section 7).

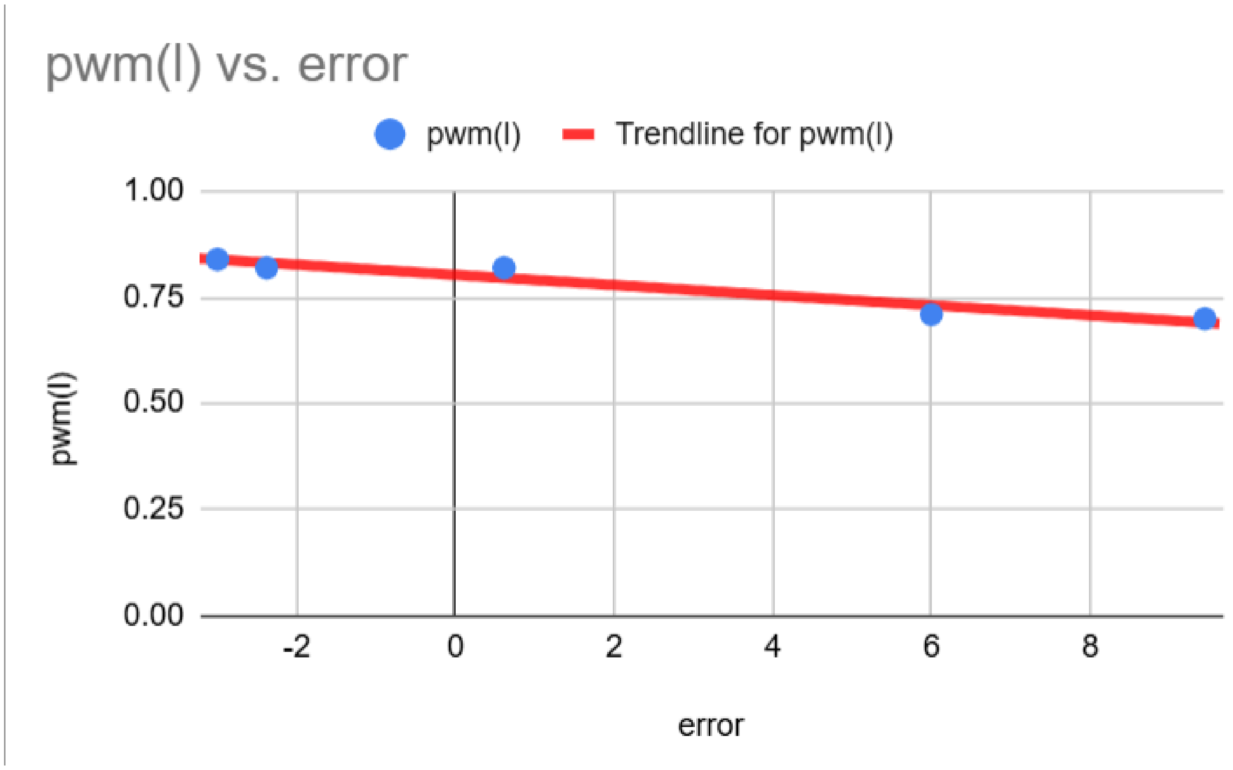
7.1 Discrete Error Bands

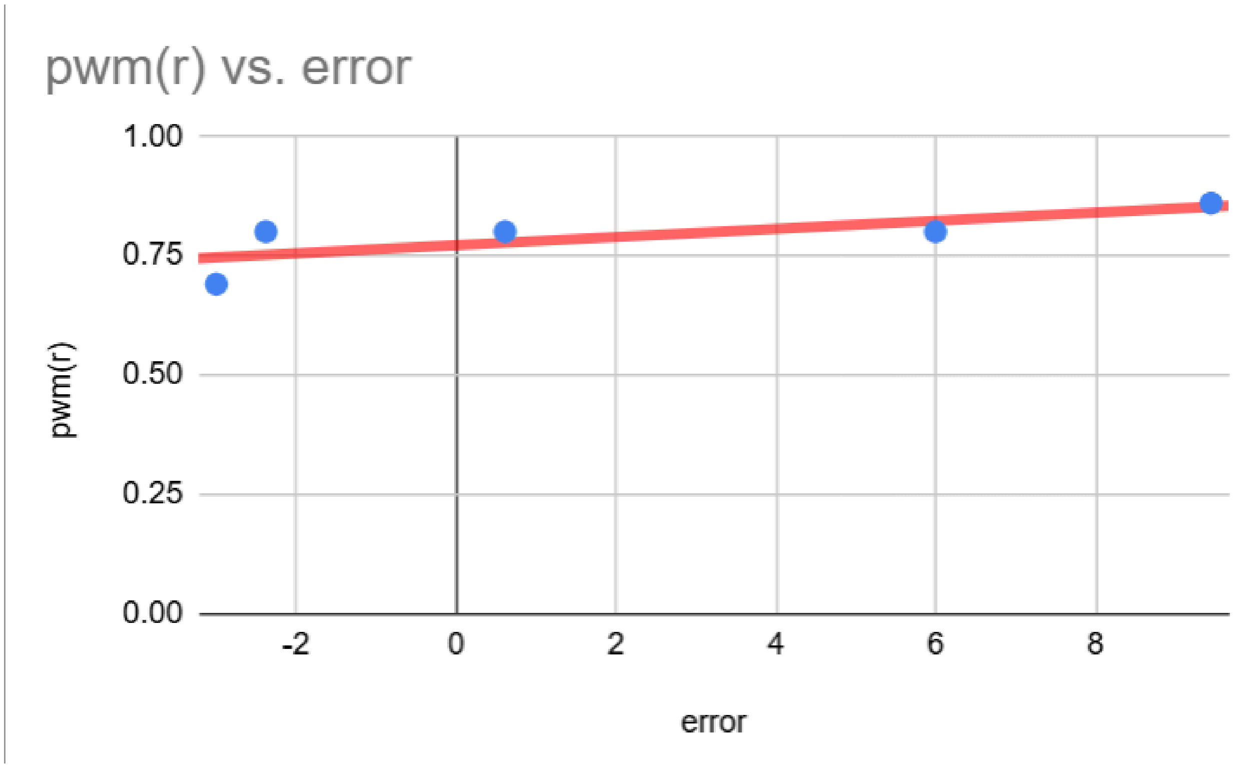
Error Band (cm)	Action	Description
-10 < e < -7	veer_L	Veer left, too close to wall
-7 ≤ e ≤ -3	steer_L	Steer left, much too close
-3 ≤ e ≤ 3	straight	Go straight, on track
3 < e ≤ 7	veer_R	Veer right, too far from wall
7 < e < 10	steer_R	Steer right, much too far

7.2 Proportional Error Feedback

error	pwm(l)	pwm(r)
-2.38	0.82	0.8
0.611	0.82	0.8
5.996496827	0.71	0.8
9.443517247	0.7	0.86
-3	0.84	0.69
pwm (L)		
lin regression	y intercept	
-0.01189577199	0.8035347173	
pwm (R)		
lin regression	y intercept	
0.008546060135	0.7717609744	

pwm_L = 0.8035347173 - 0.01189577199e
pwm_R = **0.**7717609744 + 0.008546060135





(v) a summary of code cleanup effort.

There's this part of the code that we wrote last week in the brain that was removed, it acted as a realignment function, but we removed it.

```
    if choice in ("left", "right"):
        turn_amt = behaviors.turning_behavior(choice)
        if choice == "left":
            drive.drive("spin_r")
        elif choice == "right":
            drive.drive("spin_l")

        while True:
            (L,M,R) = sensor.read()
            if (L,M,R) == (0,1,0):
                drive.stop()
                break

        map.calcturn(turn_amt)
        map.markturn(turn_amt)
```

Logic like this could easily be structured as a function in the behaviors class that could be called later onto the brain for a more clean and concise work flow of code.

Another big thing we did to clean up is the MapBuilding file which includes redundancy, and awkwardly placed functions in the class.

```
# funciton for loading map
def prompt_and_load_map():
    filename = input("Enter filename to load (e.g. mymap.pickle): ").strip()
    try:
        with open(filename, 'rb') as file:
            map = pickle.load(file)
        print(f"Map loaded from {filename}.")
        x = int(input("Enter current x position of robot: "))
        y = int(input("Enter current y position of robot: "))
        h = int(input("Enter current heading (0-7): "))
        map.x, map.y, map.heading = x, y, h
        return map
    except Exception as e:
        print(f"Failed to load map: {e}")
        print("Starting with a blank map instead.")
        return Map()
```

In the map building file as well, we defined the heading_to_delta, heading_vectors and color_map as class variables instead of recalling it within the function. Another thing to clean up the code that we didn't get around to do is automatically recalibrate the magnetometer values. We have a main function that recalculates values for the magnetometer but we have to manually change the inputs in the AngleSensor Class. Instead, to make it easier, we hope to implement it automatically.