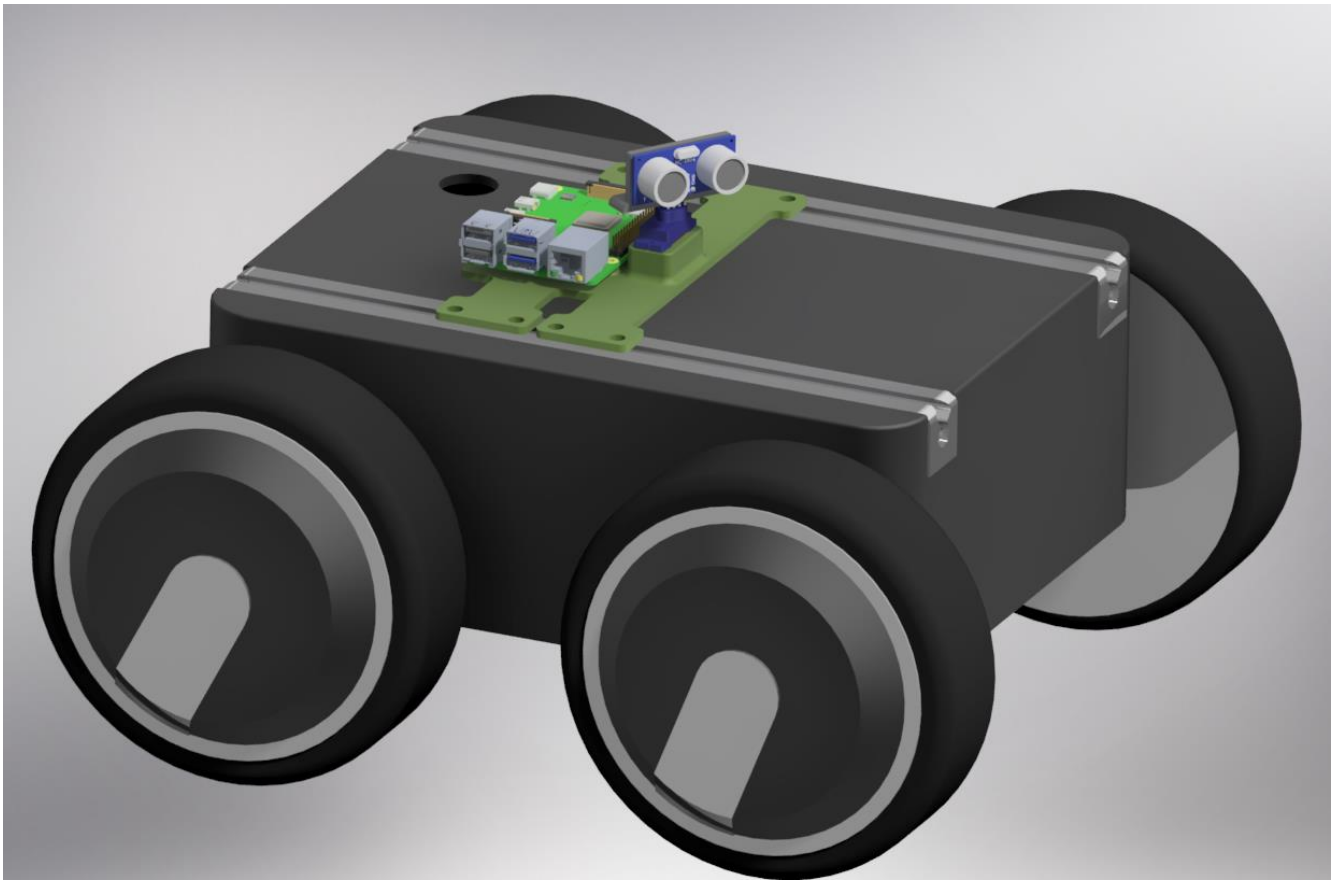
Python-Based Environment-Mapping Program for the MINI Platform using Ultrasonic



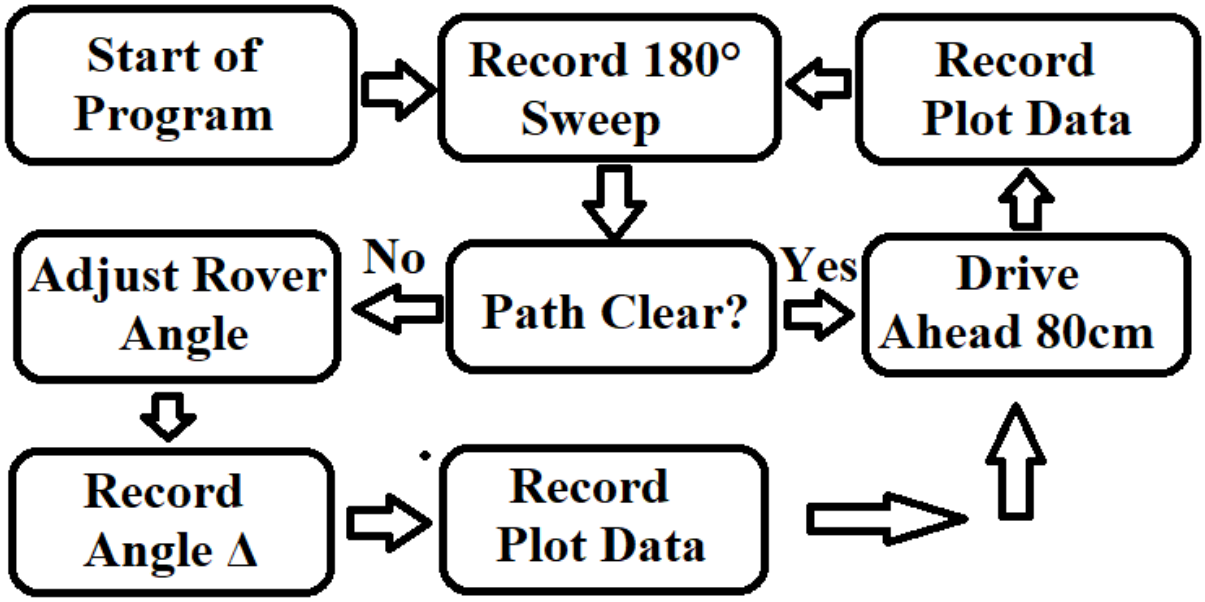
Summary

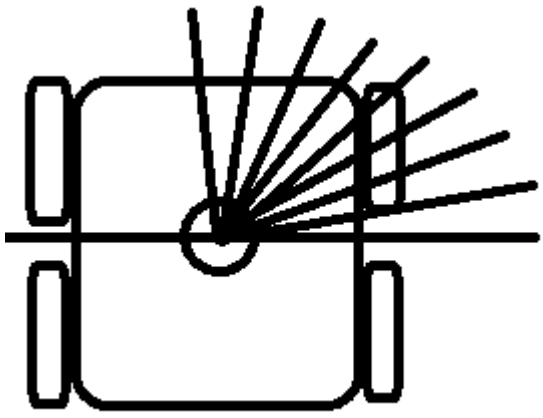
With the use of a small servo and an ultrasonic sensor, it is possible to give the Rover Robotics MINI basic room-mapping capabilities. Using some simple vector math, we can record distance data from the sensor and use it to plot a crude layout of an environment.

Table of Contents

A Logical Approach to Ultrasonic Mapping

Before delving into the code itself, it is important to understand what the software is striving to accomplish.





Upon initialization of the program, the rover spins its ultrasonic sensor 180°, recording a distance value every 10°. To determine if the path is clear, the rover then compares the 90° distance value to a reference length. If it is less than 80cm, the rover will turn; if a turn is necessary, it will check the distance value recorded 45° to the left and right of dead center. If there is an obstruction on one side, it will turn toward the other. If both sides are obstructed, it will turn around.

If no obstruction is recorded, it will continue forward with an advance of +80cm. In any of the aforementioned cases, the rover's movement data is recorded to a master variable.

Vector Math for Computation

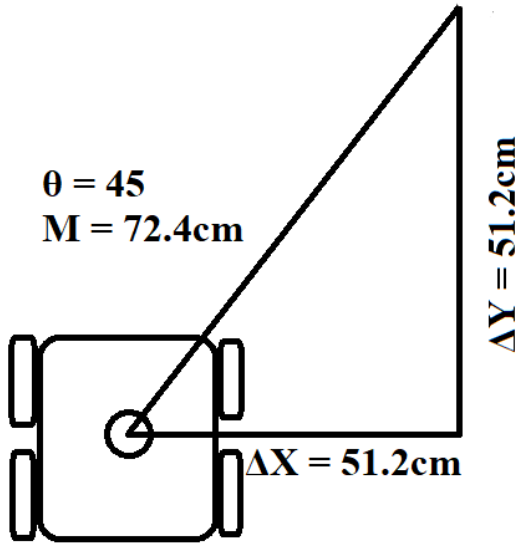
The rover keeps track of position data with the help of vectors. Anytime the rover is moved, its path angle and magnitude are used to keep a record of its current position, relative to its starting location.

When a distance value is collected by the ultrasonic sensor, the rover records the angle at which the measurement was taken, along with the measurement value itself. With some simple formulas, this information can be used to find the position of the plot, relative to the rover's starting position.

$$M = Distance\ Value\ (cm)$$

$$\theta = Sensor\ Angle\ (°)$$

$$X - Component = Mcos(\theta)$$
$$Y - Component = Msin(\theta)$$



Distance values are recorded for 2D plotting. For this reason, the rover's position is accounted for in the x/y component values. Each time the rover is moved, the translation is broken into components using the same equations. When an ultrasonic value is recorded, its components are added to those of the rover. With

this approach, we can record plot data for all points relative to the rover's starting position.

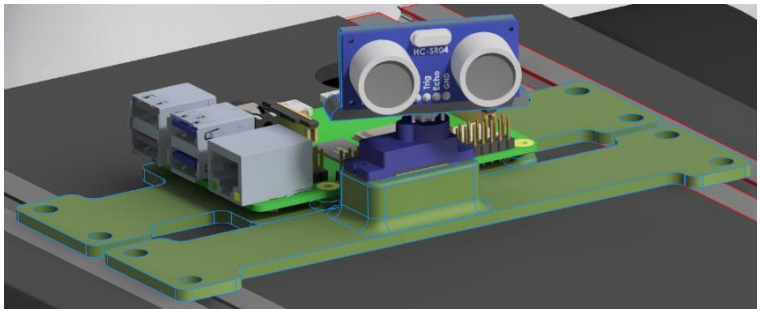
This means that regardless of which direction the rover moves, the points will always remain static.

This program is designed to run the rover indefinitely, meaning that as time progresses, the map will become more and more accurate. However, the program can be easily modified using a finite loop function to apply this technology to case-specific applications.

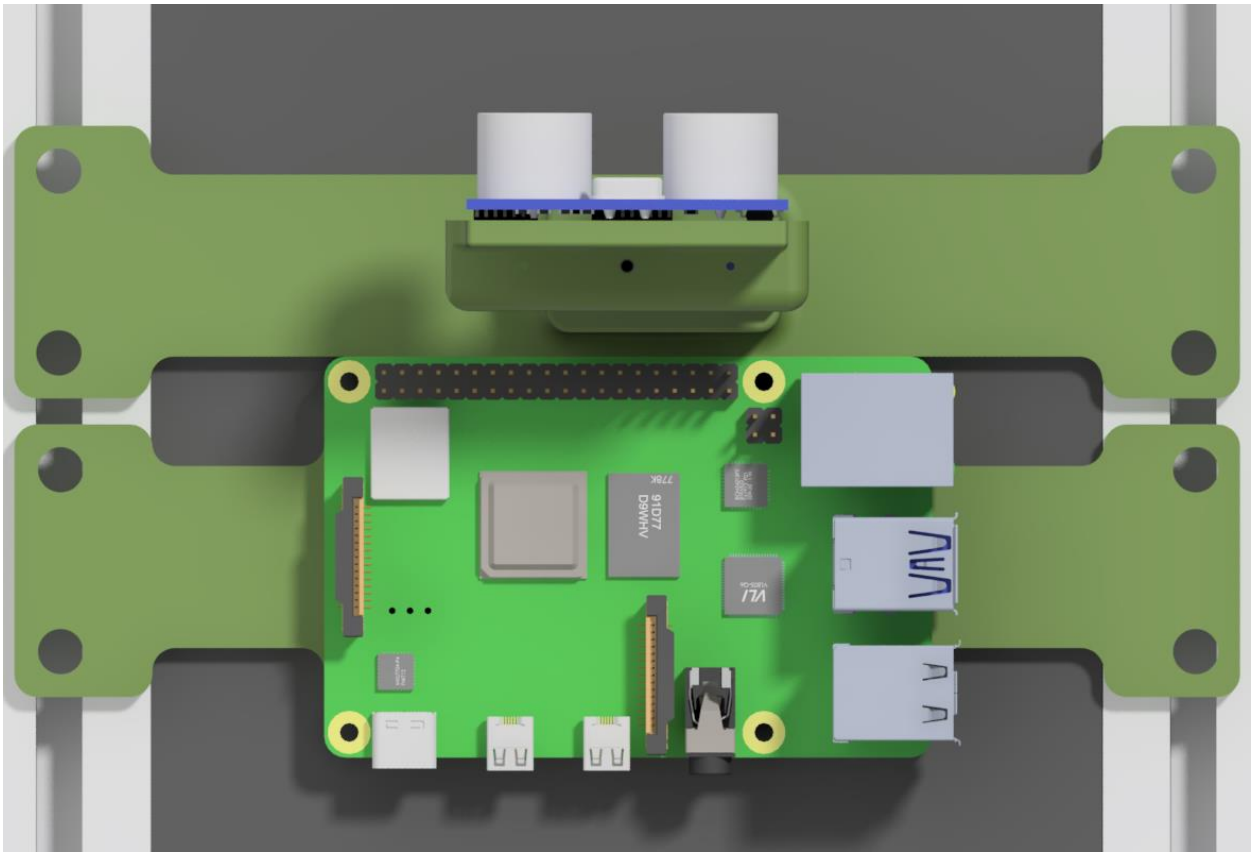
Physical Modifications to the MINI Hardware

While the computation is done onboard the MINI's integrated computer, it is necessary to introduce some additional peripheral equipment. These additional components consist of:

- SG90 Micro Servo
- HC-SR04 Ultrasonic Sensor
- 3D Printed Components
- Assorted Mounting Hardware



All additional mounting plates utilize the MINI's built-in rails and can be installed with standard hardware. It is recommended that the ultrasonic sensor baseplate be installed in the direct center of the rover to ensure accuracy. This can be accomplished with the use of a caliper and square.



The servo and ultrasonic sensor both operate at +5VDC and can both be connected to VCC & GND respectively. The GPIO pinouts for these sensors are defined within the mapping program. Please ensure that the servo arm is installed with the proper 0 ° position.

If the ultrasonic sensor's range of motion interferes with the GPIO pin headers on the Pi, it may be advisable to move the Pi's mounting bracket back to increase the clearance.

A Close Look at the Software

The mapping program is written in Python, and it makes use of a few different libraries. To avoid errors, ensure that all of the appropriate files are installed locally before running the program.

```python
# Room-Mapping Software Package for Rover Robotics Mini 1-23-24

import os                       # For writing shell commands
import time                     # For delay commands
import math                     # For trig functions
import terminalplot as tp       # For Terminal Plotting
import RPi.GPIO as GPIO         # For GPIO communication

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO_TRIGGER = 18 # Ultrasonic Trigger GPIO Pin
GPIO_ECHO = 24    # Ultrasonic Echo GPIO Pin
GPIO_SERVO = 16   # Servo Data GPIO pin
GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
GPIO.setup(GPIO_ECHO, GPIO.IN)
GPIO.setup(GPIO_SERVO, GPIO.OUT)
angles = [1.75,2.167,2.584,3.001,3.418,3.835,4.252,4.669,5.086,5.5,6.0,6.5,7.0,7.5,8.0,8.5,9.0,9.5,10]
p = GPIO.PWM(GPIO_SERVO, 50) # Initialize Servo at 50Hz
p.start(1.75)   # Send servo to starting position
```

The first thing we do within the program is define the GPIO locations for our servo and ultrasonic sensor. We assign variable names to their pins and define them as inputs/outputs. Next, a string “angles” is defined.

This string is used to store the duty cycle values for each of our angular positions as a reference list for the servo. *These values were calculated based on the **SG-90**, so if you use different hardware they may need to be modified.* Then, the servo is initialized and sent to its homed position of 0 degrees.

The duty cycle of the servo can be calculated based on a desired angle using a simple function, but by assigning the duty cycles as a referenceable list, we can conserve computational power.

The first function defined is our distance() command. This is used throughout the program to obtain the current value of our ultrasonic sensor in cm. Please keep in mind that the trigger timing within this function is optimized for the HC-SR04 sensor.

```
21  def distance():     # Find Ultrasonic Distance Value
22      GPIO.output(GPIO_TRIGGER, False) # Initialization Window
23      time.sleep(2)
24      GPIO.output(GPIO_TRIGGER, True)
25      time.sleep(0.00001)
26      GPIO.output(GPIO_TRIGGER, False)
27      while GPIO.input(GPIO_ECHO) == 0:
28          StartTime = time.time()
29      while GPIO.input(GPIO_ECHO) == 1:
30          StopTime = time.time()
31      # time difference between start and arrival
32      TimeElapsed = StopTime - StartTime
33      # multiply with the sonic speed (34300 cm/s)
34      # and divide by 2, because there and back
35      distance = (TimeElapsed * 34300) / 2
36      return distance
37
38  roverxpos = float(0) # x-component of rover position
39  roverypos = float(0) # y-component of rover position
40  recdist = [] # Define list for temporary ultrasonic value storage
```

Below that function, we define float values for the rover's X & Y position. These will be written to each time a positional command is sent to the drivetrain. It is important to track the total displacement of the rover as these changes will impact the location of points on our map. A string "recdist" is declared; it will later act as temporary storage for ultrasonic measurements.

The roverscan() function is called to acquire our 18 distance values. It runs the sensor through all of its 10° increments and records the distances to recdist[].

```
42  def roverscan(): # Collects 18 distances from ultrasonic
43      time.sleep(1)
44      for i in range(18):  # Run 18 Times (For Each Position)
45          p.ChangeDutyCycle(angles[i])  # Pull Angle Reference from List
46          time.sleep(.1)
47          recdist.append(distance())  # Store distance in recdist list
48
49
50  def componentcompute(): # Converts Distance Data into Vector Components
51      angl = 0
52      xcords = []  # List for X-Components
53      ycords = []  # List for Y-Components
54      xcords.append(20) # For Scaling X-Position Graph in Console
55      ycords.append(300) # For Scaling Y-Position Graph in Console
56      for i in range(18):  # Read List & Index Component Values
57          xcord = float(math.cos(math.radians(angl)) * recdist[i]+roverxpos)
58          ycord = float(math.sin(math.radians(angl)) * recdist[i]+roverypos)
59          xcords.append(xcord)
60          ycords.append(ycord)
61          angl = angl + 10
```

The componentcompute() function is called to calculate the X & Y coordinates of each measured point. To do this we keep track of our angle using the “angl” variable. Two more strings are defined to store the component values, and then a loop is ran. Each instance of the loop cycles it through the next set of data. The math library is used here for our trig functions.

```
def currentposition():  # Acquires Current Rover Position
    output_streamx = os.popen('ros2 topic echo --once /odometry/wheels --field pose.pose.position')
    xnew = float((output_streamx.read().partition("x:")[2]).partition("y:")[0])  # Extract Pos Value
    return xnew
```

The currentposition() function obtains the rover’s current encoder-based x-position via a ros2 /odometry/wheels echo command executed in a subshell. The output of this command is saved as a string; the position value is then extracted and assigned to variable “xnew”. This function will be used later for error analysis.

The roverposition() function handles both the movement logic and physical translation of the rover. To accomplish this, it runs through a series of conditions and makes a movement based on four scenarios:

1. The are no obstructions in front of the rover
2. The rover is blocked in front and on the left
3. The rover is blocked in front and on the right
4. The rover is blocked in front and on both sides

To make determinations about its surroundings, the rover indexes values from its most recent ultrasonic scan. We can see in the code below that it first checks value [9] of the string, which corresponds to the 90° position directly in front of the rover. All obstruction conditions are considered

after this first check, meaning that if the rover's path is clear, it will skip all if statements and simply advance forward.

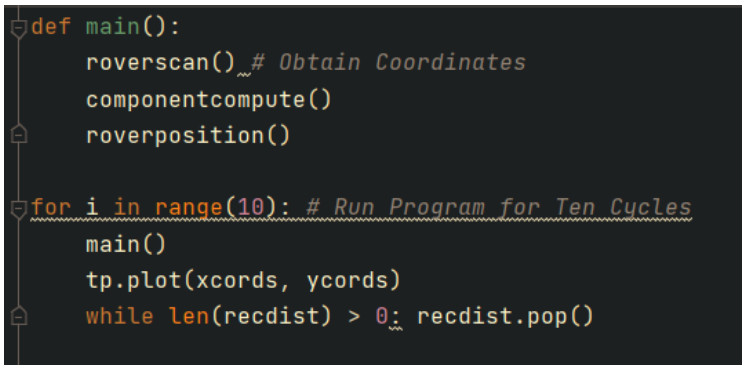
```
def roverposition():
    global leftwall, rightwall, roverxpos, roverypos, rotationvalue
    if recdist[9] < 110:  # If the rover is close to a wall
        if recdist[4] < 50:  # Check for wall on right side
            rightwall = bool(1)
        elif recdist[14] < 50:  # Check for wall on left side
            leftwall = bool(1)
        if rightwall == 1 and leftwall == 1:  # If there are walls on both sides, turn around
            os.system('ros2 topic pub -t 195 /cmd_vel geometry_msgs/Twist "{linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.1}}" -r 30') #turn around
            time.sleep(0.5)
            desiredDist = float(0.228144)  # 80cm Encoder Value
            increment = int(math.ceil(desiredDist / rotationvalue))  # Determine # of increments to move
            DesiredPos = currentposition() + desiredDist
            os.system('ros2 topic pub -t %s /cmd_vel geometry_msgs/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}" -r 30' % (increment))
            time.sleep(0.5)
            offset = currentposition() - DesiredPos  # Inaccuracy of Movement, Post to Stored Pos Value
            roverxpos = roverxpos - 80 - offset*0.0028518 # Rover Position -80cm
            return
        if rightwall == 1 and leftwall == 0:  # If left direction is open, turn left
            os.system('ros2 topic pub -t 85 /cmd_vel geometry_msgs/Twist "{linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.1}}" -r 30') # Turn to the
            time.sleep(0.5)
            desiredDist = float(0.0570063) # 20cm Encoder Value
            increment = int(math.ceil(desiredDist / rotationvalue))  # Determine # of increments to move
            DesiredPos = currentposition() + desiredDist
            os.system('ros2 topic pub -t %s /cmd_vel geometry_msgs/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}" -r 30' % (increment))
            time.sleep(0.5)
            offset = currentposition() - DesiredPos  # Inaccuracy of Movement, Post to Stored Pos Value
            roverxpos = roverxpos + math.cos(.707)*(20 + offset*0.0028518) # Rover X-Position +14.142cm
            roverypos = roverypos + math.sin(.707)*(20 + offset*0.0028518) # Rover Y-+Position +14.142cm
            return
```

```
    if righwall == 0 and leftwall == 1: # If right direction is open, turn right
        os.system('ros2 topic pub -t 85 /cmd_vel geometry_msgs/Twist "{linear: {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -0.1}}" -r 30') # Turn to the
        time.sleep(0.5)
        desiredDist = float(0.0570063)  # 20cm Encoder Value
        increment = int(math.ceil(desiredDist / rotationvalue))  # Determine # of increments to move
        DesiredPos = currentposition() + desiredDist
        os.system('ros2 topic pub -t %s /cmd_vel geometry_msgs/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}" -r 30' % (increment))
        time.sleep(0.5)
        offset = currentposition() - DesiredPos  # Inaccuracy of Movement, Post to Stored Pos Value
        roverxpos = roverxpos + math.cos(-.707)*(20 + offset*0.0028518) # Rover X-Position +14.124cm
        roverypos = roverypos + math.sin(-.707)*(20 + offset*0.0028518) # Rover Y-Position -14.124cm
        return
else: # If no Obstruction Detected, Advance Forward 80cm
    desiredDist = float(0.228144)  # 80cm Encoder Value
    increment = int(math.ceil(desiredDist / rotationvalue))  # Determine # of increments to move
    DesiredPos = currentposition() + desiredDist
    os.system('ros2 topic pub -t %s /cmd_vel geometry_msgs/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}" -r 30' % (increment)) # Ro
    offset = currentposition() - DesiredPos  # Inaccuracy of Movement, Post to Stored Pos Value
    roverypos = roverypos + 80 + offset*0.0028518 # Rover Position +80cm
    return
```

If the rover detects an obstruction in front of its path, it will then run two subsidiary statements to check its left and right sides. Remember that the scan has already been completed so these "checks" are just the program recalling its value from the string. By default, it considers its path to be obstructed if it has less than 90cm in front and 20cm to each side at 45°.

Once the rover has made a determination as to its desired path, it will execute movement commands within the subshell. For more information regarding the translation commands, refer to the Position Driver documentation. After a movement is completed, the rover will update its position values (including calculated error from the movement) and begin another scan.

Lastly, we define our main() function. This function is a bit redundant, but it is important for the sake of organization if the program is expanded in the future. This function simply runs through the scan & movement procedures. A for loop is responsible for starting the rover, and by default it runs for ten cycles. This loop can be easily modified using a variable if more cycles are desired. After a cycle is completed, the gathered points are graphed, and the distance list is reset.



```
def main():
    roverscan() # Obtain Coordinates
    componentcompute()
    roverposition()

for i in range(10): # Run Program for Ten Cycles
    main()
    tp.plot(xcords, ycords)
    while len(recdist) > 0: recdist.pop()
```

Conclusion & Recommended Revisions

While this program serves as a reliable demonstration of what can be done with the MINI platform, it is very limited in a technical sense. It is designed to be easily expanded and revised.

Perhaps some improvements could be made to this program to strengthen the rover's pathfinding. The current logic chain makes movement decisions based exclusively on three angles. Giving the rover a larger set of values to use may result in the creation of a more efficient

route. Previously traversed coordinates could also be saved and referenced in future pathfinding to prevent duplicate scanning. From a hardware standpoint, more sensors could be added to improve scan density and/or speed.

As you can see, there are a variety of ways this software can be improved upon. The hope is that this code can be used to demonstrate how this hardware can be utilized for application-specific projects.