Robo-AR

Anmol Parande, Ankit Agrawal, Jay Monga, Eashaan Katiyar

I. Objective

The project objective was to program a robot (Romi) that can autonomously follow a user-defined trajectory by placing virtual waypoint markers in an augmented reality (AR) iPhone app. The robot travels to each of these waypoints and circumnavigates any obstacles it encounters. The primary features which we implemented are:

1) BLE Communication between the iPhone and the Romi.
2) Closed-Loop P Control on the Romi for driving to waypoints.
3) Automatic path correction using feedback from AR.
4) Robot localization with AR and robot sensor data.
5) Autonomous obstacle circumnavigation.
6) In-scene and Out-of-Scene robot navigation.

With these features, we envision our project could grow into delivery, autonomous driving or warehouse navigation related areas for short to mid-range tasks. For example, with the current social distancing guidelines, people could theoretically transfer documents between each other in an office by attaching a basket to our robot.

II. System Design Overview

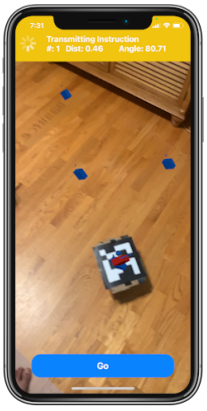

Fig. 1. User App View

From a technical point of view, our project consisted of three major components. We implemented an autonomous driving model as a Hierarchical Finite State Machine in C, an iOS application in Swift (shown in fig. 1) to perform vision, and a Bluetooth Low Energy (BLE) protocol to transfer information between the two devices.

A. Hardware Architecture

Our hardware was largely unmodified from the original Romi architecture. The only modification made was to attach a cardboard fixture with an AR marker on top of it to simplify the computer vision required for localizing the Romi in space. We opted to not put a camera on the Romi itself and instead use an iPhone as the external camera since this reduced the complexity of the vision processing required and allowed us to focus on implementing the embedded systems concepts. For more details about the specific sensors, actuators, and communication protocols used, see fig. 2.

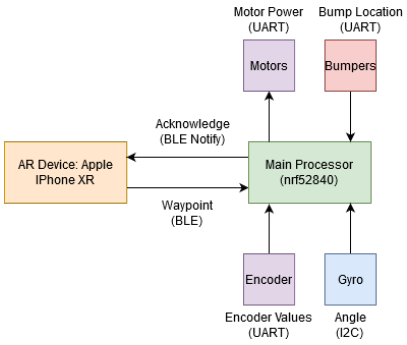

Fig. 2. Architecture Diagram

B. Romi Model (Hierarchical FSM)

The code controlling our Romi was designed as a hierarchical finite state machine and directly implemented in code. The Romi is in charge of two major tasks: communication with the iPhone and obstacle avoidance through driving.

1) Code Structure: To implement our FSM, we developed an abstraction to separate it from the sensors and actuators. The FSM states are defined by an enumeration, and all the associated state variables are stored in a struct. We have functions `get_inputs` and `do_outputs` that are called every time-step to provide the input signals and emit the output signals. At every time step, the FSM will `transition`. The functions: `transition_in` and `transition_out` handle startup and cleanup behavior for any given state. This allowed us to modularize our code and made it easier to keep track of what variables were being set. The refinement had a similar code structure.

2) State Machine: The main state machine manages the communication with the iPhone. It consists of five states: `OFF`, `WAITING`, `TURNING`, `DRIVING` and `END TURNING`. Our system starts in the `OFF` state, and upon establishing a BLE connection with the iPhone, moves to the `WAITING` state where it is ready to receive instructions. When an instruction is received, the robot enters `TURNING` and uses the gyroscope to turn the θ degrees specified by the instruction in order to face its destination. From here, the robot goes

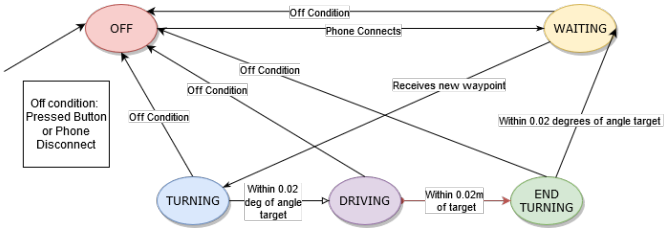

Fig. 3. Overall FSM

to DRIVING where it drives a distance d forwards while circumnavigating any obstacles it encounters. Once at the waypoint, the robot enters END TURNING where it undoes any orientation changes caused by obstacle avoidance. If at any point, the phone disconnects or the robot's button is pressed, the robot moves to OFF. A diagram of this machine with the guards and states is shown in fig. 3.

3) Driving Refinement: Obstacle avoidance is a refinement of the DRIVING state since it requires no communication with the phone. The refinement for obstacle avoidance has 5 states: FORWARD, AVOIDANCE, STOPPED, ROTATION, and BACKWARD. The name of each state describes how it is moving in that state. The variable *avoidance distance* (denoted as α) controls how far forward after encountering an obstacle the robot will drive. Our algorithm is described in algorithm 1, and the refinement is depicted in fig. 4.

Algorithm 1: Obstacle Avoidance Algorithm

```
1  α = 0;
2  (x, y) = 0;
3  while distance to destination not within threshold do
4      robot drives straight for d meters;
5      if robot encounters an obstacle then
6          α = α + 0.2;
7          stop;
8          move backwards a distance of 0.3m;
9          turn 45 degrees away from the obstacle;
10         drive straight for α meters again;
11         if robot encounters an obstacle then
12             goto line 5;
13     turn towards destination;
14     compute Δx and Δy from sensors;
15     (x, y) = (x, y) + (Δx, Δy);
```

C. Robot Controls

We designed two P-controllers to ensure the Romi can both drive straight and turn precisely. Since our system did not noticeably overshoot its target nor did it have steady state error, we decided against adding integral and derivative terms to the control. To stop our motor inputs from getting too small, we defined a minimum motor input for each controller. Once the Romi reaches its target value, we set both wheels speeds to 0. Other than minimum input non-linearity, our P-controllers were implemented in a standard linear feedback control system

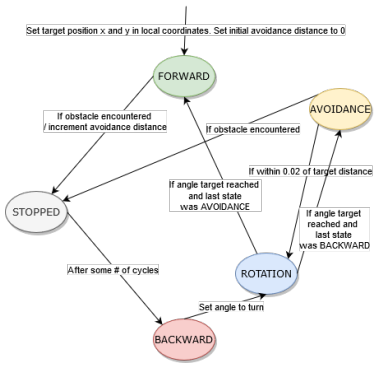

Fig. 4. Driving Refinement

as shown in fig. 5. $r(t)$ is a general reference value, from which we subtract the current target value to generate an error term that is scaled by the appropriate gain K_p.

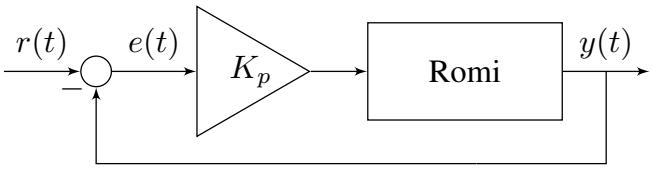

Fig. 5. Turning Controller Block Diagram

For turning, we minimize the difference between the angle measured by the gyroscope and the target angle θ. In other words, our $r(t) = \theta u(t)$ where $u(t)$ is the unit step. We used $K_p = .8$ as our proportional gain and 50 as our minimum wheel input. Since the $|\theta| \leq 180$, our motor inputs will always be in the range of $[50, 144]$, and we would reach the minimum speed once we are within $62.5°$ of the target value. After testing a range of minimum speeds and gains, we learned that a relatively low minimum speed and gain was desirable because turning too fast could create inaccuracy in our angle measurement and increase the chances of overshoot.

For driving straight, we try to minimize both the difference between the distance traveled and the target distance (for driving the correct distance) as well as the difference between the distance traveled by each wheel (for driving straight). If d is the target distance, d_r is the distance traveled by the right wheel, and d_l is distance traveled by the left wheel, then $r(t) = \left(d + \frac{250}{210}(d_r - d_l)\right) u(t)$ where $u(t)$ is the unit step. We chose $K_p = 210$ as the proportional gain for distance to travel and 80 as our minimum wheel input. If the target is sufficiently far away, this would saturate our motor inputs, but it doesn't seem to cause any issues. We would roughly reach the minimum speed once the Romi is within .38 m of the target value. We chose these values because, unlike the turning loop, testing a range of minimum speeds and gains revealed that higher motor inputs made the relative wheel velocities much less mismatched. This is shown in fig. 6 which compares the distance traveled by each wheel for motor inputs at 60 vs 140.

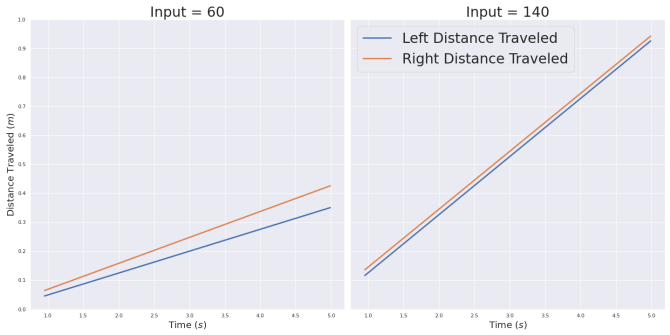

Fig. 6. Distance traveled by both wheels against different motor inputs

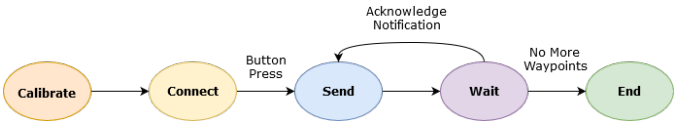

Fig. 7. BLE Protocol

D. BLE Protocol

In order to communicate, the iPhone and the Romi use a BLE service with two characteristics as a peripheral. The two characteristics are:

1) **Acknowledge**: A Read/Notify Boolean characteristic that is set to 1 when the Romi receives an instruction and set to 0 when the Romi has completed an instruction.
2) **Waypoint**: A Read/Write characteristic contains an 8-byte array containing two float values. The first 4 bytes describe the instruction distance (m) and the next 4 bytes describe the instruction angle in degrees.

We designed our BLE protocol to give us the maximum flexibility. Treating waypoints as a series of distance-angle instructions means that the Romi does not have to keep track of its position over long periods of time nor does it have to know where other waypoints are. Instead, all of that logic can be compartmentalized to the phone which can use AR to localize both the Romi and the waypoints. This makes it easier to focus on implementing good control algorithms for the Romi and allows code on the Romi and the iPhone app to be developed simultaneously. This was incredibly helpful for testing the Romi controls because we could use a Python script to test arbitrary distances and angles without being constrained by the iPhone applications.

The general flow of the BLE communication is shown in fig. 7. When transmitting an instruction, the iPhone computes the turning angle and travel distance required of the robot using algorithm 2, which it packs into the 32-bit waypoint characteristic and sends to the robot to fulfill. The angle is computed using algorithm 3. When the Romi receives the waypoint, it sets the acknowledge characteristic to 1 (true), blocking the iPhone from sending additional waypoints. Upon turning through the appropriate angle and driving the distance specified by the instruction, the Romi will reset the acknowledge characteristic to 0 (false), notifying the iPhone that it is ready for the next waypoint.

E. Phone Algorithms

Our algorithms for computing distances and angles relied on measurements from Apple's RealityKit framework.

1) Computing Instructions: In order to allow the phone to accurately send instructions to the robot even when it cannot localize the robot, we introduce the abstraction of the "checkpoint". A checkpoint contains a reference to a position in AR as well as an orientation. The very first checkpoint is set to be the robot when it is initially detected by the phone. The AR world origin is also set to the robots initial position and orientation For this reason, the robot needs to be localized at least once before the user can place any checkpoints. When an instruction is sent, the checkpoint is updated to be the waypoint the Romi is currently driving to as well as the orientation (relative to the Romi's initial orientation) that the Romi will arrive at the checkpoint with. As a result, if the Romi suddenly goes out of the view of the app in the middle of traveling to the next waypoint, the app can assume it was successful and use the checkpoint orientation as well as the angle between the checkpoint and target waypoint to compute the distance and angle for the next waypoint. This allows us to use algorithm 2 for both when the Romi is being tracked and when it is not. All distance computations are done by Apple's ARKit framework.

Algorithm 2: Computing instructions from robot location

```
if robot in scene and close to target then
    waypoint = next waypoint;
else
    waypoint = current waypoint;
end
if robot in scene then
    trackedObj = robot;
else
    trackedObj = checkpoint;
end
angle = trackedObj.angleTo(waypoint);
distance = trackedObj.distanceTo(waypoint);
instruction = {distance, angle};
orientation = checkpoint.orientation + angle;
checkpoint = {waypoint, orientation};
transmit(instruction);
```

2) Computing Angles: To compute angles for the instructions, we use basic vector math. First, we assume the all waypoints and the robot are in a horizontal plane (x and z axes). We then compute the dot product of the vector from the current position to the destination with the forward direction and use the cross product to determine if we need to turn a negative or positive angle. This algorithm is described in algorithm 3.

III. Relation to Course Concepts

Our project utilized an extensive range of concepts taught in this course. The three primary concepts which formed the

Algorithm 3: Computing instructions from robot location

forward = $[0, 0, 1]$;
direction = destination - position;
direction.y = 0;
angle = $\arccos\left(\frac{\langle direction, forward\rangle}{\|direction\|}\right)$;
if $(direction \times forward).y < 0$ **then**
  angle = $-1*$ angle;
end
return angle;

foundation of our project were *Wireless Networks and BLE*, *Hierarchical State Machines* and *Sensors and Actuators*

1) Wireless Networks: We utilized BLE for communication between the iPhone and the Romi. The Romi was our peripheral and the iPhone was the central. They communicated via the GATT protocol. We had to set up our own service and characteristics on the Romi as well as configure them for our use case. Our full BLE protocol is discussed in section II-D.

2) Hierarchical State Machines: We modeled and implemented the Romi code as a Hierarchical State Machine. We built reset transitions and preemptive transitions in our code. Further details about the hierarchical state machine model and implementation are included in the section II-B.

3) Sensors and Actuators: The Romi sensors which we relied upon were the on-board gyroscope, motor encoders, and the bump sensors. We can also think of the iPhone camera as a sensor because it provided us an accurate measurement of where the robot was localized in space relative to some origin point (it's initial starting location). The only actuators involved were the Romi motors. The sensors and actuators as well as the communication protocols used to send and read data from them are depicted in fig. 2.

IV. Evaluation and Results

TABLE I
System Accuracy

True Distance - AR Distance	True Angle - AR Angle
$-0.0175\,\mathrm{m}$	$-1.12°$
AR Distance - Robot Distance	**AR Angle - Robot Angle**
$-0.0725\,\mathrm{m}$	$2.2°$

There are two sources of error in our system.

1) Error between true distance/angle and measured distance/angle in AR.
2) Error between AR measured distance/angle and Romi distance/angle.

AR measurements on the iPhone XS come from Apple's computer vision algorithms which operate on the camera feed. This means our accuracy between distance/angle measured by AR vs the distance/angle measured with a meter stick is limited by the algorithms provided by Apple. We tried to limit this error by providing a reference image (the AR marker) with a known width, but other than this, the only way to remove this error would be to upgrade our phone to the latest model which uses LIDAR for AR. The difference between how much distance/angle the Romi is instructed to move and how much it actually moves is due to sensor noise. We limited the impacts of this error by having the iPhone visually confirm the Romi reached the target waypoint (within a margin of $10\,\mathrm{cm}$) and send a correction instruction if it was not. To quantify these errors, we did a series of trials where placed an AR waypoint at the end of a meter stick at a known angle (measured with a protractor). For each trial, we recorded the distance measured by AR, the angle measured by AR, the distance traveled by the Romi, and the angle turned by the Romi. The average AR and Romi errors are shown in table I.

It is worth noting that we cannot quantify exact error because the "ground truth" position of waypoints is what is in AR. As a result, the best way to evaluate the system is visually, where it performs quite well (see video).

V. Conclusion

A. Limitations and Future Goals

While our system is capable of both obstacle avoidance and basic navigation, we believe there is more that can be done. In particular, our design does not use any other sensors other than our gyroscope and encoders for navigation. This can lead to some inaccuracy, so in the future, it could be useful to incorporate information from AR into the feedback control loop. This would allow the robot to travel along user-specified curves in addition to reducing error.

Another goal would be to have more robust obstacle avoidance, perhaps using a robot-mounted camera or ultrasonic sensors. Currently, the robot has to physically bump into an object, limiting its senses to head-on collisions. A mounted camera could be used to determine how close / far the robot is from a potential obstacle.

B. Successes

Our system exceeds our initial proposal of designing a robot that could work with AR to navigate to different waypoints and be controlled by BLE. We were able to accomplish our stretch goal of obstacle avoidance, and we successfully allowed our system to work when the robot is both seen and unseen by the phone.

We also believe there is a lot of strength to how we designed and implemented our system. We focused on ensuring the individual components independent of each other and implemented well enough to be extensible in the future. We put a lot of our effort into debugging and testing as well, using Python to handle some of the Robot specific test cases.

VI. References

[1] Ahmad, Jawwad. "Core Bluetooth Tutorial for IOS: Heart Rate Monitor." Raywenderlich.com, www.raywenderlich.com/231-core-bluetooth-tutorial-for-ios-heart-rate-monitor.

[2] Chugh, Anupam. "Introduction to RealityKit on IOS-Entities, Gestures, and Ray Casting." Medium, Heartbeat, 21 Jan. 2020, heartbeat.fritz.ai/introduction-to-realitykit-on-ios-entities-gestures-and-ray-casting-8f6633c11877.