Team BSR
RoboCupJunior 2022 Competition

Members:
Allen Gu
n/a
Marco Hu
n/a
Brian Jiang
n/a
Mentor:
Graciela Elia
n/a

I. Abstract

This document is about the design process and implementation of a robot for the Robocup Junior Rescue Maze. It covers the software used for movement, maze exploration, and victim detection, as well as the hardware used to accomplish our goals. It also documents our field testing procedures and future improvements that can be made.

II. Introduction

Natural disasters or severe accidents often result in the damage of property and loss of life. However, conditions may be too dangerous for humans to provide necessary aid to victims. In this case, robots may be used instead. They are able to traverse the terrain without risking lives, and can provide supplies to and/or rescue victims of the accident.

To be successful, the robot must be able to navigate the terrain without human assistance, detect victims and provide aid accordingly, and return back to the starting point. The Robocup Junior Rescue Maze simulates this situation.

III. Planning

I. Research:

To start, we explored hardware options, maze solving and victim detection methods, and designs of past teams. Our research gave us a general idea of the direction we wanted to move in. We narrowed down the hardware that we wanted to use and started discussing maze exploration and letter recognition algorithms.

II. Design:

We started with a circular body design, and edited the design from there. It eventually evolved into an oblong shape. We decided to use two large wheels as well as two caster wheels, and we created a two-layer body structure to allow for all of the hardware to fit. Our model was created in Onshape due to its shareability and ease of use.

III. Software:

When the design was done, we started writing the programs that would allow the robot to run. Explanations of the algorithms we used are written below. One development in letter recognition was the switch from Pytesseract OCR to a contour-counting method.

IV. Assembly and testing - gathering materials, attaching hardware, applying code, troubleshooting using a simulated maze environment

After writing our programs, we started gathering necessary hardware and wiring the robot. We ran parts of our code through the robot, and troubleshooted issues in a test maze that would simulate the actual competition.

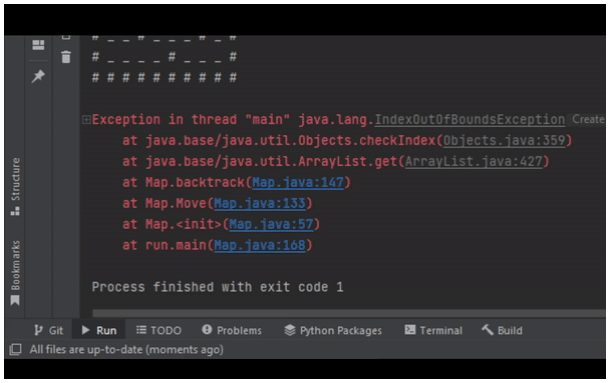
IV. Software and Algorithms

Pathfinding (Maze Exploration)

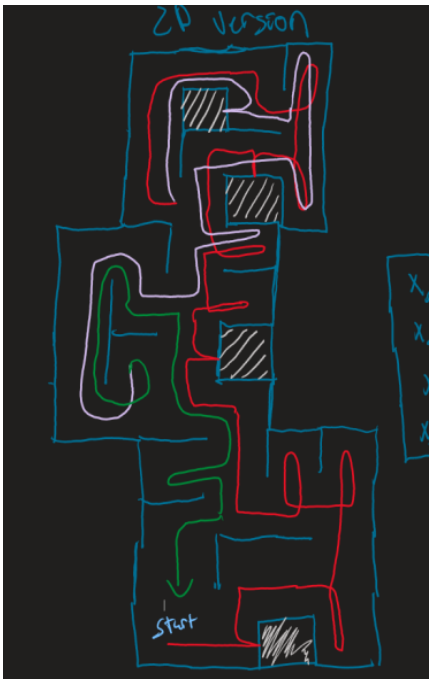
Our strategy for navigating the maze is to have the robot traverse every single tile on the maze.
If every single path from the starting point to every tile on the maze can be represented as a tree, we can use a depth first search to traverse every tile.

This image represents every path the robot takes. To explore every possible path we can fully traverse a path and then backtrack to the last branching node and take the other direction.
The robot can detect whether adjacent tiles are blocked, and can move to 1 of the 4 adjacent tiles. Each tile is either explored or unexplored, and is either decision or not decision. A tile is marked as decision if it has unexplored tiles adjacent to it. If the robot goes over a tile, the tile is marked as explored. The robot will record the path it took, and will only travel from an explored tile to an unexplored tile, unless backtracking. When backtracking, the robot will return to the last decision tile (unexplored adjacent tile). The order of priority for unexplored tiles is right > top > left > bottom.

Due to the fact that Java is easier to develop in than C++, testing and design for the pathfinding algorithm was done in Java. It was later adapted into C++ for the Arduino. The Java Prototype
When the robot has finished exploring the maze in the java prototype, it will return an indexoutofbounds error.
Gif of simulated movement in a maze.



When the robot backtracks, its previous moves will be deleted from the memory, this is true because as each path on the tree is explored, the robot will never need to traverse that path again.



Movement process:

The algorithm outputs absolute directions, as opposed to directions relative to the robot. The orientation of the robot is stored so that the robot can turn to face the absolute direction, then move forward.
The robot travels about 30cm in 1874 steps. Once the robot starts moving forward, it will check for black tiles and victims. If the robot detects a black tile and reverses away, it will indicate that the movement failed. The algorithm will then remove that movement from the history and output a new move.
After a move, error will build up due to variations in tile size and terrain. If there is a wall in front of the robot, we will use the distance from the wall to adjust the robot's position and recenter the robot.

Hole identification:

In order to identify if the robot will traverse a hole, we use a color sensor that compares lux. Black tiles which would be much darker than normal could be detected by the lux sensor. In the process where the robot travels 30 cm forwards, it will also read from the lux sensor to check if it is about to drive over a hole. If it does, it will reverse back to the previous tile.

Visual Victims:

We used OpenCV to process the images from both cameras and detect visual victims.

Color Recognition:

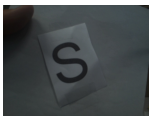
To detect color victims in an image, we defined upper and lower bounds of HSV values for red, green, and yellow, and changed the pixels that fell within the ranges to their respective colors. Next, we used

OpenCV's findContours function to find potential color victims. To eliminate false detections, contours must have 4 sides and meet an area requirement.
If any contours passed the requirements, we cropped the image in a 30x30 square around its center. We checked the BGR values of the remaining pixels. Then we used NumPy's count_nonzero function to find the number of pixels of each color. If 800 of the 900 pixels matched either red, green, or yellow, the computer would detect a victim and return its color.

Letter Recognition:

First, we thresholded the darkest pixels and OpenCV's findContours function to determine potential letter victims. To avoid false detections, contours must be fully contained within the image, meet an area requirement, and have a certain height-to-width ratio.
Since letter victims may be rotated at any angle, we used the minAreaRect and warpAffine functions to rotate the image, so that potential victims that were at an angle would still be correctly registered. To avoid a sideways image, if the width of the rectangle was greater than the height, the image was rotated by an additional 90 degrees.
With the rotated image, we used the center coordinates, width, and height of the minimum area rectangle to crop the image around the letter. To identify the letter, we segmented the image and counted the number of contours within each segment. The image was divided into left and right halves, as well as top, middle, and bottom thirds. The type of victim would be based on the number of contours counted in each section. To avoid detecting false contours, only contours that were at least ¼ the area of the largest contour were counted.

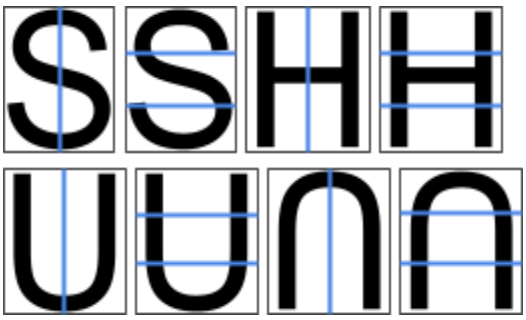
Image Recognition Algorithm:

Image	Grayscale	Threshold	Rotate	Crop	Divide L/R	Divide T/M/B	Count contours, return result
							# of contours: 2L, 2R 1T, 1M, 1B Return 'S'

Number of significant contours in each section

	Left/right	Top/middle/bottom
S	2 / 2	1 / 1 / 1
H	1 / 1	2 / 1 / 2
U	1 / 1	2 / 2 / 1
U (upside down)	1 / 1	1 / 2 / 2

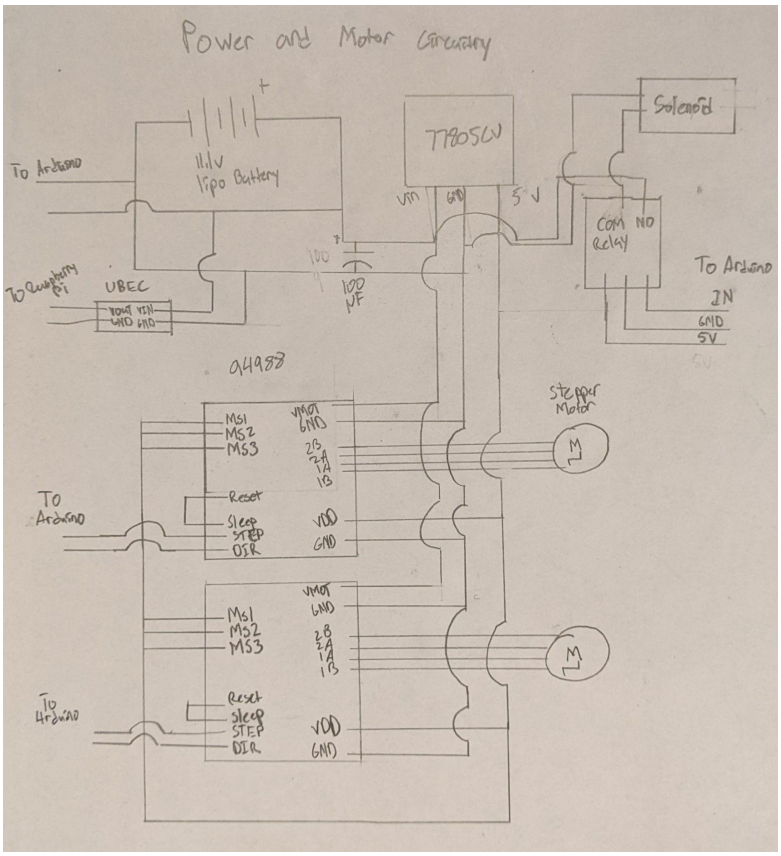
Segmenting the letter contour

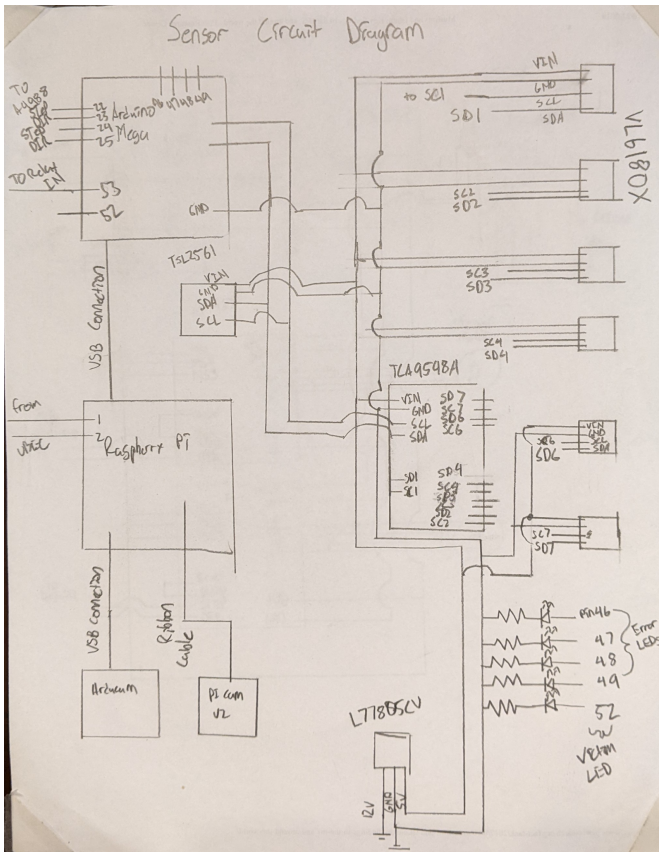


To leverage the full processing power of the Pi, we ran image processing of the cameras in parallel with multithreading.

Communication between the Pi and Mega

We initially tried to use a parallel communication protocol where information was sent through IO pins in





a binary code, but we decided that a serial protocol would be just as effective with a cleaner implementation. Using the Serial libraries for Python and Arduino, we could open a serial port between the two over USB and transfer the data in strings.
Once the Pi has found a victim, it will send over a string consisting of the camera and a letter representing the identified victim. (h s u r y g ; ex. “1u” for camera 1 detects an unharmed victim)
The Arduino will periodically check the serial buffer while it moves, read the string, and drop the appropriate number of medkits.

Error identification:

To help with debugging on the field, we implemented a 4-bit binary system with LEDs. In the event that the Arduino fails to start a sensor, an LED will be lit up, which can then be used to identify the error. We also have an error light on the Pi that will indicate if it fails to open a serial port with the Arduino.

V. Hardware

Circuit diagrams for power and motor

Note: the current wiring of the robot follows the general circuitry of the robot but some connections of the power connections have been changed, while keeping the overall functionality.

Sensors

MLX90614 - Contactless temperature sensor
TSL2561 - lux sensor
VL6180X - Time of flight sensor

We chose a VL6180X time-of-flight sensor to detect walls over other popular alternatives such as an ultrasonic sensor because the algorithm is tile-based and the robot only needs to detect whether an adjacent wall exists. In addition to a larger form factor, ultrasonic sensors may also interfere with each other. This problem could be solved using a time-of-flight sensor, which uses light instead of sound to measure distance. The sensors are much smaller than an ultrasonic sensor and equally reliable.

A large problem that we faced was how to have the robot accurately turn. We worried that without a method of correction, inaccuracies built up through constant turning would eventually send the robot off a predictable course, which the algorithm does not plan for. A proposed solution to this was to measure yaw using an IMU. We started with an MPU6050, a 6 DOF accelerometer and gyroscope. We found that the resulting yaw calculated from integrating rotational data was inaccurate. IMUs are prone to drift (the mpu yaw drifted 0.02 rad/s). Many people have sought to mitigate drift through filters such as the madgwick and kalman filter. Using a 9 DOF IMU (accelerometer, gyro, magnetometer), the LSM9DS0, we thought that error could be corrected with the madgwick filter. Unfortunately, the measurements did not improve. We decided to compromise on error correction and dropped the yaw measurement method. To calculate proper turn angles, we used the center of rotation and distance from wheels to calculate the amount of steps for the robot to rotate 90 degrees. Due to the inherent accuracy of stepper motors, we decided that an IMU would be superfluous.

Controllers

At the start, our team used a Raspberry Pi 3 B+ and an Arduino Nano. However, due to a shortage in pins, we switched to the Arduino Mega for its greater port selection. On top of that, the Arduino Mega had 8 kilobytes of memory, which was 4 times more than the Arduino Nano with 2 kb. The increase in memory was critical for pathfinding due to the way data gathered about the maze was stored.

Cameras

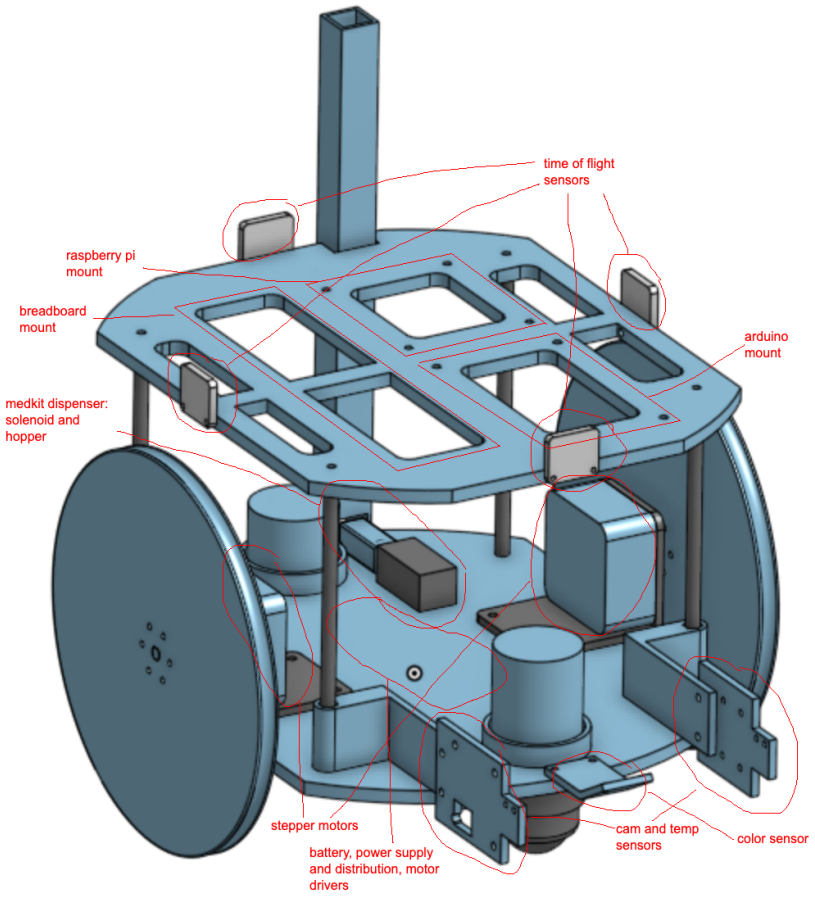
Originally, we planned to use a rotating camera arm connected to a servo motor. We discarded this idea, as the robot would have to stop and wait for the camera to scan the surroundings, and the camera was too close to the walls to effectively recognize victims. We avoided using camera multiplexers due to their poor speed and image quality. Ultimately, we used a USB camera attached to one of the Pi's USB ports, alongside a PiCam.

Dropper

We used a hopper mechanism and a pusher to reliably drop a controllable amount of medkits. For the hopper, we designed a tall tube that could store medkits in a single stack. For the pusher, we required something that could produce linear motion quickly. We considered using a servo motor and a guide to convert rotational motion to linear but settled on a solenoid, which would work better at the cost of a greater power requirement. The solenoid runs on 12 volts of electricity which can consequently be pulled out of the battery directly. To handle power switching, we used a relay controlled by the Mega.

VI. Reliability / Quality Assurance

To test our robot, we simulated a maze environment by using wooden planks as walls. We debugged the movement code by testing the robot in small sections of the maze. In addition, we ensured that the robot was able to drive up slopes, stairs, debris, and various ground materials.
When the robot could explore the maze consistently, we taped color and letter victims to the maze walls. We uploaded images taken by the cameras from the maze to calibrate the values used for victim detection (ex. HSV bounds, contour size).



VII. Constraints and Future Interest

One constraint that we faced was that the robot cannot move sideways. It must turn in place, creating slight inaccuracies that build up over time. In addition, the robot would need to be restarted if it messed up its orientation or location in the maze, which is very time consuming and inefficient. Another constraint was that the cameras have hard-coded HSV bounds, so they may not work in all lighting conditions. This requires the robot to be calibrated before entering the field. Finally, our color recognition algorithm only looks at the center 30x30 pixels of the contour, which can result in errors.

The orientation problem could be fixed in the future by using omniwheels, allowing the robot to drive in all directions accurately. If the robot loses track of its location in the maze, we could have the robot keep track of the last visited silver tile, so that it can be reset and still remember the layout of the maze.
To avoid needing to calibrate the robot, we could use a color correction card to make sure that the image is consistent, regardless of the lighting conditions. To improve the color recognition algorithm, we could take the average color of the entire contour, rather than the color of certain pixels, and compare the average color to pure red, yellow, or green. This would return much more accurate and consistent results.

We look forward to implementing these changes in the future.

VIII. Conclusion

In this paper, we have discussed the software and hardware we used to create a robot for the Robocup Junior Rescue Maze.

To explore the maze, our robot moves from explored to unexplored tiles, and backtracks to the nearest unexplored tile when there are no adjacent unexplored tiles. We used a color sensor to avoid driving over holes in the ground. To detect color victims, we counted the number of colored pixels at the center of qualifying contours. To detect letter victims, we divided qualifying contours into different segments and counted the number of significant contours in each one.

For hardware, we used MLX90614 temperature sensors, a TCS34725 color sensor, and VL6180X time of flight sensors. We also used a Raspberry Pi 3 B+ and an Arduino Mega, with a PiCam and USB camera attached to the Pi. We used a solenoid and a hopper system to drop medkits.