N.E.R.D.
RoboCupJunior 2021 Competition

Alex Hu
AlexHu00374@gmail.com

Andy Cheng
andysoccerdude333@gmail.com

Dhruva Chakravarthi
dhruvachakravarthi@gmail.com

Peter Lin
230pelin@gmail.com

Jeremey Desmond
jdesmond38@gmail.com

I - Abstract

II - Introduction

III - Robot Design

IV - Sensors

V - Motor Control

VI - Serial Communication

VII - Maze Algorithm

- **A - Project Structure**
- **B - Storage of Maze**
- **C - IO Options**
- **D - Maze Algorithm / BFS**
- **E - Benchmarks**

VIII - Victim Detection

- **A - Planning**
- **B - Implementation**
- **C - Integration**

IX - Rescue Kit Dropper

X - Future Interest

XI - Conclusion

I - Abstract

This article is about our software and hardware development process for RoboCupJunior Rescue Maze. It also includes the planning done throughout the whole process. Our end goal was to be able to traverse the maze efficiently while giving rescue kits to the victims who needed them.

In order to accomplish this, we had to design and build a robot suited for traversing the maze environment, which included choosing the appropriate sensors to get the job done. The most important part of the design phase was sensor placement, as we needed to have the most precise values in order to detect victims and walls.

Next was software design, which made up a majority of time spent on the project. Our robot ran on two boards, so we had to have two separate processes running on each. The Raspberry Pi had our cameras connected to it, so it acted as the brain of the robot, whereas the MegaPi had the motors connected to it, so it served to help with movement.

Our functional algorithm is more sophisticated, but the basic understanding of communication between these two boards was the Raspberry Pi sending the MegaPi instructions, then the MegaPi following those instructions to go through the maze.

II - Introduction

1. Hardware
    a. RaspberryPi - One of the two circuit boards used in our robot, controls the camera.
    b. MegaPi - The second circuit board we use, which controls the sensors, servo motors and DC motors.
    c. TOF (Time of Flight) sensor - Distance sensor that utilizes lazers.
    d. Servo Motor - Small motor used to turn 180 degrees.
    e. DC Motor - Large motor, used to move the treads. These motors have encoders.
2. Software
    a. Python - Programing language used to control the RaspberryPi side
    b. cv2 - Library used to process images in python

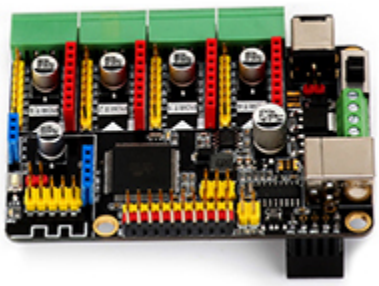
c. NumPy - Library used to create advanced multi-dimensional arrays & matrices
d. Serial Communication - Process to send data over a wire, used to transmit data and messages from and to RaspberryPi and MegaPi
e. Contours - A line representing the shape of an object, used in victim detection

III - Robot Design

---

1. We wanted our robot to be compact enough to move around comfortably in the maze but still have enough space to fit the sensors, boards, and other mechanisms on it.
2. The base was made out of pieces from the OSEPP Tank kit (we did not follow any premade plans), which includes a base for the boards and wiring to rest on, as well as metal beams that hold the motors and base together.
3. We decided to use treads because they were good at traversing over the maze and getting over speed bumps and debris smoothly.
4. We used the Raspberry Pi and the MegaPi as the main controllers for our robot because the Raspberry Pi can use cameras for victim detection and the MegaPi is good for motor control and can have multiple sensors attached to it. The MegaPi is a type of Arduino, so we use C, while in the Raspberry Pi, we programmed in Python.

MegaPi (Arduino)



Raspberry Pi

IV - Sensors

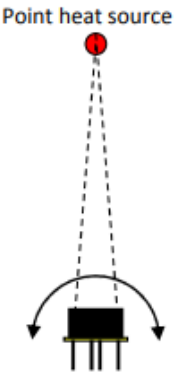
Introduction

In order to have a successful run, we would need to detect a variety of victims using sensors. Because of this, the choosing, testing, and implementing of specific sensors is very important to the overall function of our robot. The following section includes how we planned what sensors to use and the hardware and software components of our sensors.

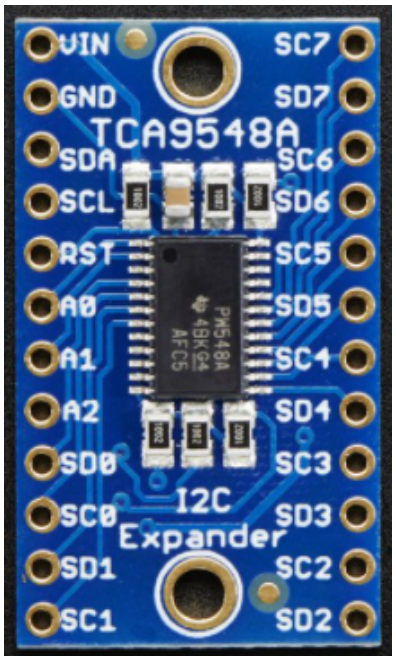
Project Planning and Development Cycle

1. We needed sensors to detect heat victims.
    a. We decided to use a contactless heat sensor called the mlx90614 because it provides accurate readings of the temperature of objects from a distance.
2. We needed sensors to detect black and silver tiles.
    a. We decided to use the TCS34725 light sensor because it could detect RGB and clear light values, which would effectively detect black and silver tiles.
3. We needed sensors to find the distance between the robot and walls.
    a. We decided to use the VL53L0X Time of Flight for accurate readings at a pretty long range, which would allow us to accurately detect walls.
    b. We decided to use 5 time of flight sensors so that we could have 2 sensors on the left and right of the robot to align with the walls, and one time of flight sensor at the front to detect walls in the front of the robot
4. We needed to use an I2C Multiplexer in order to use all sensors with the same I2C address, which we used the TCA9548A for

Hardware

1. PiCam
2. ArduCam
3. Time of Flight Sensors
    a. For each of the five time of flight sensors we used, we connected the clock and data pins to a port on the I2C multiplexer.
    b. We put one of the time of flight sensors at the front of our robot, and on both the left and right of our robot, we put one time of flight sensor near the front and one near the back
4. Heat Sensors
    a. Our heat sensors could detect heat at a narrow beam
    b. 

    c. To attach the heat sensors to our robot, we soldered the pins of the heat sensor onto wires
    d. We connected the clock and data pins of the heat sensors to the I2C multiplexer because they had the same I2C address
5. Light Sensor
    a. Our light sensor could only detect light a very short distance to the ground

6. I2C Multiplexer
    a. The I2C multiplexer makes it possible to use sensors with the same I2C address
    b. The I2C multiplexer has 8 ports - one sensor could be plugged into each port
    c. 

    d. One sensor would connect into each pair of SD and SC on the multiplexer, and the SDA and SCL on the multiplexer would be connected to the SDA and SCL on our robot

Software

1. Both cameras were used to detect victims, which will be explained more in depth in the Victim Detection section.
2. Time of Flight Sensors
    a. We used the RangeMilliMeter function from the Vl53L0X to get readings
    b. We divided the readings by 10 to get the centimeter values of the sensors
    c. We subtracted 4 to 6 from each sensor to compensate for their individual errors
3. Heat Sensors
    a. We read in the temperature from the heat sensor in degrees celsius
    b. If the temperature the heat sensor read was between 28 and 40 degrees celsius, we determined that a victim had been detected.
4. Light Sensor
    a. We used the Clear Light detection of the light sensor to detect black and silver tiles because it had the most distinct values for the two colors.
5. Multiplexer
    a. We used a tcaselect function to select buses in our I2C multiplexer
    b. The tcaselect function would be called with a port number as a parameter to select the bus every time before we use each sensor with each port corresponding to a specific sensor

V - Motor Control

Introduction

Traversing the maze field necessitates precise movement of the robot, as such, motor control must be as exact and reliable as possible. In this section, we will go over how we run the motors and the different functions we use to make them turn and go forward accurately.

Project Planning and Development Cycle

1. We needed to implement a timing system to ensure our motors would follow the encoders accurately.
    a. To solve this, we used sample code to create a class for motor control with encoders.
2. We also needed certain functions to make the robot turn and go forward.
    a. We decided on three main functions, one that does turning, one that goes forward, and one that goes forward for one tile.
3. We had to ensure the robot would correct itself if a turn messed up in the middle, so we had to find a way to align.
    a. Our solution was to use our time of flight sensors to align our robot against the nearest wall.

Software

1. We used a class which implemented interrupts in order to guarantee proper motor running.
    a. The class variables include:
        i. “backwards” - This is a boolean flag that represents whether a motor is backwards or not. In our case, our right motor was placed opposite to our left, so we used this to make sure they both ran in the same direction.
        ii. “port” - The motor port that is connected to the MegaPi
        iii. “intPin” - The interrupt pin used with the motor
        iv. “encPin” - The encoder pin used with the motor
        v. “count” - The encoder count
    b. The only function we have inside the class is “setMotorSpeed.” It sets the speed based on whether the motor is set to “backwards” or not, in which it reverses the speed.
2. The functions we implemented to control motors are: doTurn, goForward, goForwardTiles, alignLeft, alignRight, alignFront, and alignRobot.
    a. The doTurn function takes in a character and an integer value. The character can be either “L” or “R”, representing a left or right turn. In order to turn the correct amount, we compare our motor’s encoder value to the wheelbase divided by the diameter of the robot’s treads multiplied by the amount it should turn. We also use the absolute value of the encoder so the value can stay positive when comparing it to the target value.
    b. The goForward and goForwardTiles functions take in integer values, representing how many centimeters and how many tiles to go forward respectively. We compare the target encoder value to 360 divided by tread diameter multiplied by pi, which is all multiplied by the distance it should go forward. If it is going forward a number of tiles, the previous number would be multiplied by 30 since each tile is 30 centimeters long.
    c. The alignLeft and alignRight functions take values from the two time of flight sensors on each side (left and right) of the robot to determine how far the robot should turn to align. If one sensor has higher readings than another, the robot will turn such that the difference in distance the two

sensors are seeing decreases. This allows the robot to align with the wall to the left or right side of the wall.

d. The alignFront function uses the single time of flight sensor in the front of the robot to align using the wall in front of the robot. If the sensor in front is too close to the wall, the robot will back up until it is 5 cm away from the wall. On the other hand, is the sensor in front is too far from the wall, the robot will move forward until it is 5 cm away from the wall.
e. The alignRobot function combines the three other alignment functions by determining which wall the robot should align to and then aligning it accordingly. First it checks if there is a wall to the left side using the time of flight sensors. If the distance is greater than 30 centimeters (meaning there is no wall), it will then check if there is a wall to the right side and repeat the process. If there are no walls to both the left and right sides, the robot will not align to any of those sides. If there are walls on both sides, the robot will align to the left by default. The final check is if there is a wall in front of the robot. If there is a wall within 30 centimeters, the robot will align to the front.

Hardware

1. MegaPi Encoder Driver

2. MegaPi Encoder Motor

VI - Serial Communication

Introduction

In order to allow for communication between the two boards we are using, the Raspberry Pi and the MegaPi, we use Serial communication. Serial communication sends encoded messages through the Serial port connecting the two boards. Messages sent from one board are received on the other and can be decoded using Serial.read(). This reading function not only reads the message, but it also pushes the just-read message out and moves the Serial buffer up one. This means values can be easily jumbled up if not organized correctly. For each side, Raspberry Pi and MegaPi, messages must be sent at certain times and must not interfere with other messages.

Project Planning and Development Cycle

1. We needed to ensure timing between the two boards was precise enough that messages would not be mixed up with others.

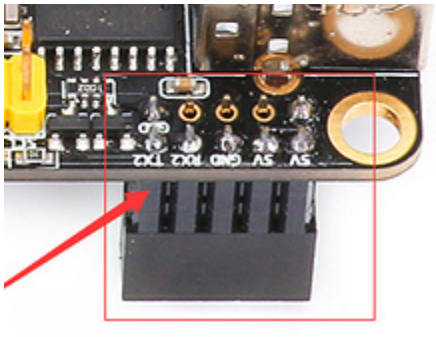
a. We accomplished this by sending various types of messages over Serial. Some are confirmation messages, which are used to make sure timing is correct and a message is received, others are instructions, which are values that should be read in and used in the code itself.

Software

1. An example of Serial communication from the Raspberry Pi to the MegaPi and back is when we determine whether there are walls around the robot. The Raspberry Pi sends a letter "g" to the MegaPi, signaling that it requires wall values from the MegaPi side. Once the MegaPi receives these values, it will get the wall values using the time of flight sensors, then send back an array of wall values, for example "111", which means there are walls on the left, right, and front sides of the robot.
    a. The function "requestData" sends a message, the letter "g" from the Raspberry Pi side to the MegaPi. It then waits until the MegaPi sends the values (which sides the walls are on) back to the Raspberry Pi.
    b. The function "sendWallValues" takes in three inputs, the distance from the left, right, and front sensors. It takes these values and determines whether there is a wall on any of the sides, then sends them through Serial communication as an array of three characters.
2. Another example of Serial communication in our code is in motor control. We use several functions to implement this type of communication. First, on the Raspberry Pi side, the Pi sends a letter "m" for motor control, then uses the function "sendData" to send over the motor values. We encode these values by separating them with semicolons and abbreviating the movement type. Each message is ended with a "$" so the ending spot can be determined. We use "L", "R", "F", and "FT" along with the desired movement amount to send to the MegaPi. A sample message would look something like this: "L90;FT1;R90;FT1;$". This translates to: turn left 90 degrees, go forward one tile, turn right 90 degrees, and go forward one tile.
    a. The function "sendData" basically just takes the values calculated from the search algorithm and sends the corresponding motor values.
    b. The function used to receive these messages is "motorControl", which takes in each message sent by "sendData" and decodes it. The decoded messages are then put into the motor functions and the robot runs accordingly.
3. The third time we use Serial communication is when receiving values for victim detection. Once the Raspberry Pi detects a victim from the camera feed, it sends a message, the letter "v." As soon as the MegaPi receives this message, it knows that it should execute code to drop rescue kits. The messages sent to the MegaPi include the side the victim is on and how many rescue kits it needs. For example: L3 means there is a victim on the left side who needs three rescue kits.

Hardware

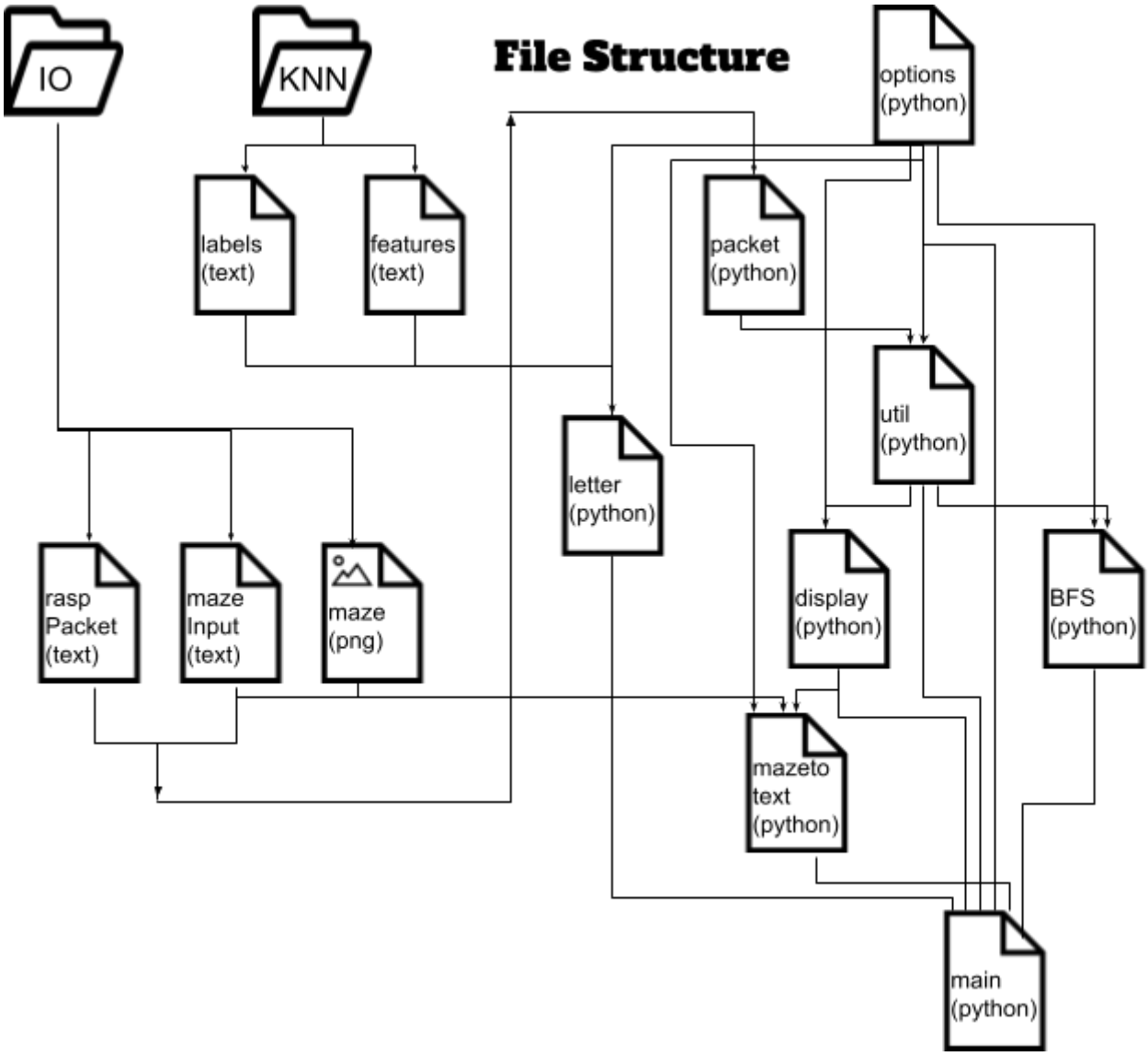
1. Serial port connecting Raspberry Pi and MegaPi



VII - Maze Algorithm

Design & Structure:

A. Project Structure (Raspberry Pi Side)



a. We have a total of 8 python files, four text files, and a maze png. Various files are split up due to a different purpose, and having them all as one would be messy. The project is intentionally made modular so files can be included and excluded at will.

b. Rundown of Python files

   i. options.py

      1. options.py has multiple settings and variables to change the program for each environment and debug easily.

2. These include settings such as settings the maze side length, input type, serial ports, a debug mode which toggles print statements, and easy changing of file paths and display rates.
3. Every python file in the project includes it.

ii. packet.py
1. This file is focused on input, output and serial communication.
2. It has setup functions that for each type of input/output, which are run based on a mode in options.py
3. util.py, main.py include it, as they both deal with input, output and serial.

iii. util.py
1. Util, or utility, contains utility functions and variables
2. These variables include the maze, current tile and all other variables used by BFS
3. Functions in this file include ones for retrieving input from packet, adjusting directions, and sending output requests from packet.
4. BFS.py, display.py and main.py include it, as they use variables and functions located here.

iv. letter.py (aka. everythingDetect.py)
1. Python file that handles everything to do with victims, and our process for doing victims will be explained later.
2. main.py is the only file which includes this file, as main is the only file that handles victim detection.

v. display.py
1. The purpose of this file is to display a given maze using cv2.
2. The walls of a maze are black lines, and the current position is marked as a green square. The target position is also marked as a red square, and any areas in between the two are displayed as a yellow tile, or path tile.
3. This file only has three functions, one for setup, one for creating the cv2 image, and one for showing it.
4. The includes of this file are util.py, for displaying based on the maze, target, and path variables located there, and options, for access of displayRate, displaySize & displayMode.

vi. BFS.py
1. This file contains the maze algorithm that is used, BFS.
2. The functions include a setup, one to calculate the next position, one to calculate the path to the next position, and one to send movement and turning instructions to util.py.

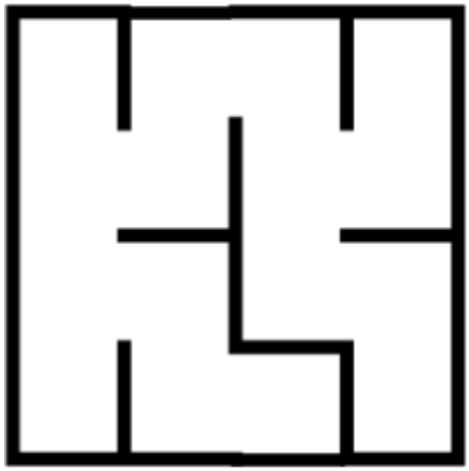
vii. mazeToText.py
1. For testing, takes an input maze from IO, ‘maze,png’, and writes the maze values to ‘mazeInput.txt’
2. A maze such as maze.png can be generated and downloaded from mazegenerator.net

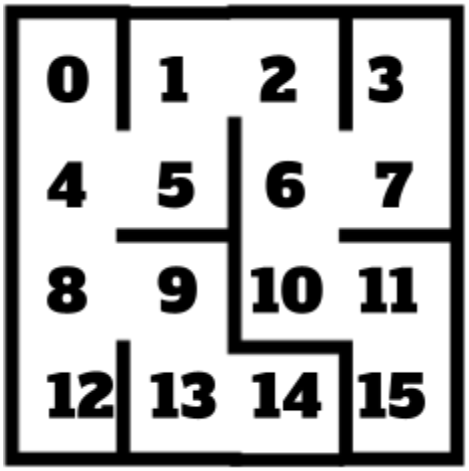
viii. main.py
1. main.py is the file that is actually run by the Raspberry Pi.
2. It includes all python files in the project, and is an integration of all of them.

3. It runs setup, gets input, calculates a path, outputs path, checks for letters, and repeats.

c. Rundown of IO and KNN folders
   i. IO
      1. raspPacket
         a. This file is an output file where an output path can be written. This is another output instead of writing to serial or writing to the console.
      2. mazeInput
         a. As the name describes, this file is for a maze input.
         b. mazeToText.py takes an image, and writes the maze to this file.
         c. The program then takes the input in this file and uses it to solve a maze. This is purely for testing and benchmarking the algorithm.
      3. maze.png
         a. The maze image as described above.
   ii. KNN
      1. These files hold the test data for the KNN letter detection. One holds data values and the other holds the corresponding labels.
      2. This can be updated whenever to improve letter detection.

B. Storage of Maze





a. In order to store the maze, such as the one above, tiles can be held as an element in an array. This array originally was a python list, but was changed to a NumPy array. The index for each of the tiles is listed above.
b. To store attributes about a tile, such as where walls are located, if the tile has been visited, or if there is a victim, each of these tiles is an array.
c. This means our maze is a two dimensional array, with one dimension being the length of the maze, and the other dimension being a length of 5.
   i. The length of 5 is due to the first four values being storage for walls, and the last element for checking if visited and if victim.
   ii. For wall locations, 1 indicates a wall, and 0 indicates no wall
   iii. For whether visited or victims, 0 is unvisited, 1 is visited, 2 is visited + victim.
   iv. Assuming there is a victim on tile 0 in the image above, the tile array for tile 0 would be:

1. Index 0: Wall North, set to 1
2. Index 1: Wall East, set to 1
3. Index 2: Wall South, set to 0
4. Index 3: Wall West, set to 1
5. Index 4: Visited/Victim, set to 2

d. In this case, the maze side length would be four, resulting in an array that is (sidelength * sidelength) x 5, or a [16][5] NumPy array.
   i. The way that our program is setup only allows the maze side length to be even.
   ii. Due to the nature of not knowing where the robot is located in the maze at the start, we assume that we start in a middle tile, which in the case above would be tile 10.
   iii. In our program we set the actual side length to 80, as anything above that value causes memory issues when running the search.
   iv. If a smaller maze, such as the one above, is run in a maze with a side length of 80, it will run fine, even with a higher side length than in the actual maze.

e. In order to get tiles surrounding a given tile, we have some simple equations. Let's take tile 9 as an example.
   i. To get the tile north of 9, we do (tile - sidelength), or 9 - 4, giving us tile 5
   ii. To get the tile east of 9, we do (tile + 1), or 9 + 1, giving us tile 10
   iii. To get the tile south of 9, we do (tile + sidelength), or 9 + 4, giving us tile 13
   iv. To get the tile west of 9, we do (tile - 1), or 9 - 1, giving us tile 8

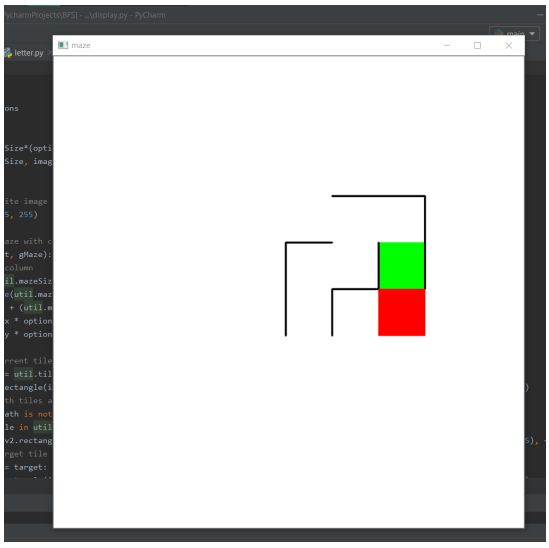
f. In our utility file, we have functions to do the equations above, as well as definitions of N, E, S, W using variables. They can be accessed using util.N, util.E, etc.
   i. N = 0
   ii. E = 1
   iii. S = 2
   iv. W = 3

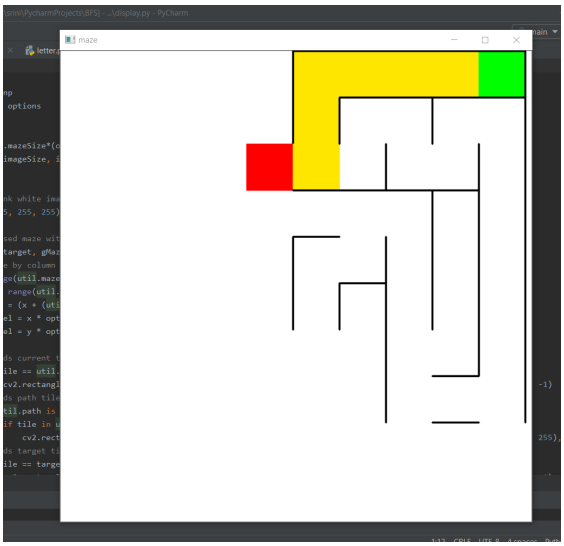
C. IO Options
   a. Evolution/Types
      i. IO through shell/print statement
         1. In the first couple versions of our maze algorithm, we would take wall values in from shell, in a format such as 1010, which would indicate
            a. Wall north
            b. No wall east
            c. Wall South
            d. No wall west
         2. For output, we would print out the target tile number, and the program would ask for input, and so on.
            a. This type of input was extremely tedious, as if we messed up we would have to reset, and visualizing what would happen in the maze was difficult, based on just tiles being printed out as output.
      ii. File IO
         1. Instead of the input being through shell, we decided to opt for getting input from a file, and writing out the movement to another file.

a. At this point we were also planning out serial communication, and changed the input to 101
    i. First digit would represent a wall on the left
    ii. Second digit would represent a wall on the right
    iii. Third digit would represent a wall in the front
    iv. There is no need for a back sensor, as there will always be no wall in the back, as the robot would have come from that way.
        1. The only exception to this would be the starting tile, as we could start with a wall in the back
b. The input file had the word "INPUT" at the beginning of the file
c. The output was also changed to match serial. Let's take going from tile 10 to tile 7 in the diagram above, and facing north as an example. The output would be:
    i. "FT1;R90;FT1$"
        1. FT__ means forward __ tiles
        2. R__ means right __ degrees
        3. "$\n" at the end of one instruction.

iii. Display, PNG Maze / display.py & mazeToText.py

1. To help visualize the maze more, we decided to display the maze step-by-step using cv2, a program called display.py (Example below)
    a. The current tile is green
    b. The path tile(s) are yellow
    c. The target tile is red





    d. Many settings for debugging, such as how big the display is, whether or not to display, what rate to display it at, and whether to only display the next tile only after a key press were implemented in options.py.
2. To get input easier than through a file or through the shell, we were debating whether or not to create a maze generation algorithm or to use on online
    a. In the end, we chose to use one online, specifically mazegenerator.net

b. In order to change the downloadable maze png into an actual maze, we made a program, mazeToText.py, which takes the image, finds where the walls are, and writes those locations to a file.
c. The top of the file says “GENERATED”, so the program can distinguish between the two different types of file input, manual and generated.

3. Both of these combined allowed us to test very quickly and even automate testing to check for errors/stalls. They are one of our best assets for creating a robust maze algorithm.

iv. Serial communication

1. The last input/output system was through serial communication, and testing with an actual robot.
   a. The serial is between the RaspberryPi, where our code is located, and the MegaPi, which controls motors and sensors.
2. The values received from the MegaPi already followed an input we had tested with file input, so implementation for motor and sensor values was quite straightforward.
3. The output is the same as mentioned above in example 1 of section iii of file IO, so sending it through serial instead of writing to a file was the only difference.

b. Keeping all of these different IO options, and being able to toggle them quickly with only one variable change in options.py, allowed us to test very efficiently and easily.

i. For example, if there was a maze issue with our robot using serial, we could easily reproduce the same maze with shell input in order to find out if it was an algorithm issue, a serial issue, or a sensor issue.

D. Maze Algorithm / BFS

a. Our goal is to find all of the victims in the maze, and in order to do this, we must solve the entire maze by visiting every part.

i. Possible solutions

1. After doing research on many maze algorithms, and found three possible algorithms that could help
2. In our case, we would want to visit unvisited tiles, making them our goal node.
   a. A*, which relies on a weight map to find goal nodes
   b. DFS, which finds the farthest goal tile
   c. BFS, which finds the nearest goal tile

ii. We found that the most intuitive way to solve the maze would be the BFS, or Breadth First Search, as finding the nearest unvisited tile would reduce the amount the robot needed to move to get to the next tile.

b. High Level BFS Process

i. Set wall locations of the new tile using util.setWalls()

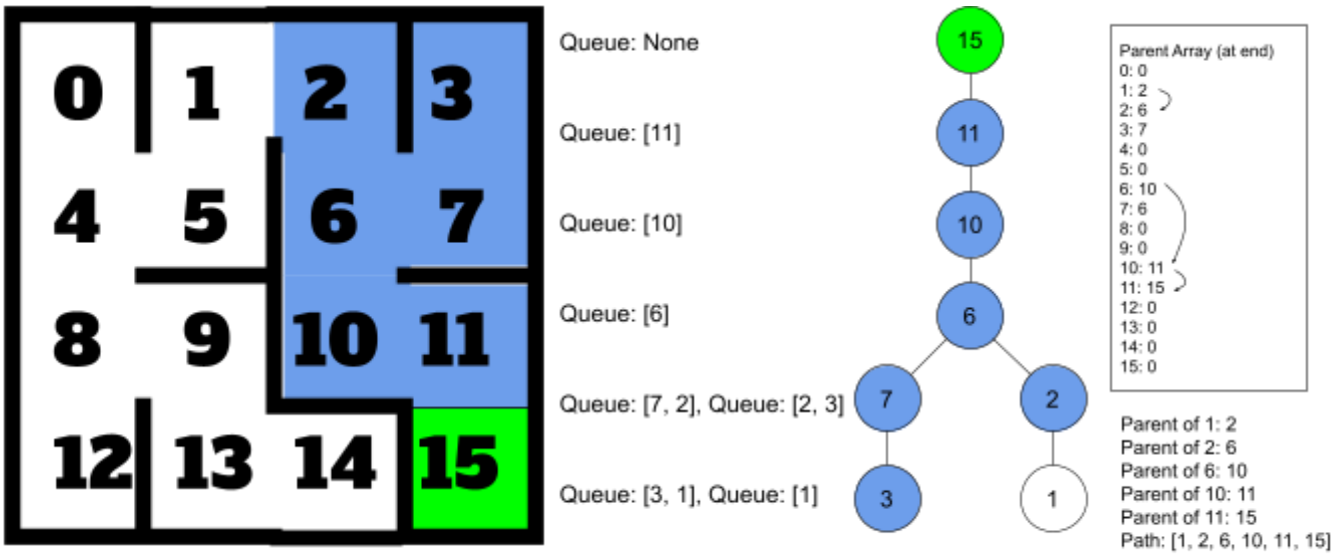
1. Sets walls based on input mode and gets input from either file, console, or serial port.

ii. Calculate target tile with BFS.nextTile()

1. nextTile(tile) is a recursive function which takes an input tile, and returns back the nearest unvisited tile, or target tile. (Explained in depth below)

iii. Find the path from the current tile to the target tile using BFS.pathToTile()

iv. Generate instructions for movement with BFS.turnToTile and util.forwardTile()
    1. This is done in a while loop, with the condition being: while the path generated by pathToTile has at least one element.
    2. The last element of the path stack is popped off during the while loop.
v. Mark the new tile as visited, and reset the parent array
vi. Repeat until all tiles visited



c. Low Level BFS Process (Evaluation/Analysis)
    i. nextTile = BFS.nextTile(util.tile)
        1. This recursive function takes an input tile, and using the maze & some other variables, it returns back the next tile to move towards. Variables used by this BFS include:
            a. Maze, current maze
            b. Tile, tile where the robot is currently located
            c. Direction, which is stored as 0 - 3 for N - W
            d. A queue, for queuing tiles to check (python list)
            e. A stack, for storing the path (python list)
            f. Parent array, which stores relationships of tiles, such as which tiles are the children of others
        2. Process
            a. Take the passed tile, check if it is unvisited
                i. If unvisited, clear queue and return the tile, BFS is over
            b. Create blank 4 element array for storing possible tiles
            c. Check if wall north
                i. If there is no wall north and the tile north isn't located in the parent array, add the tile north to the parent array and queue. Set the 0th index of possible tiles to 1.
                    1. The check for if in the parent array prevents having the same tile repeat over and over.

d. Check if wall east
    i. If there is no wall east and the tile east isn't located in the parent array, add the tile east to the parent array and queue. Set the 1st index of possible tiles to 1.
e. Check if wall south
    i. If there is no wall south and the tile south isn't located in the parent array, add the tile south to the parent array and queue. Set the 2nd index of possible tiles to 1.
f. Check if wall west
    i. If there is no wall west and the tile west isn't located in the parent array, add the tile west to the parent array and queue. Set the 3rd index of possible tiles to 1.
g. Loop through possible tiles
    i. If possible tiles of the index is 1:
        1. If the queue is empty, BFS is over and the entire maze has been visited
        2. Else, run nextTile() on the first in the queue, and pop it. This makes the function recursive, as it is calling itself for the surrounding tiles.

d. Example with image above
    i. In the image above, the function is currently on the green tile, tile 15. We will assume that the blue tiles have already been visited, and we have never been to the white tiles.
        1. In this situation, the nearest unvisited (white) tile would be tile 1
    ii. The tree above can model the entire BFS process
    iii. In a recursion depth of 1 the function is passed 15, and tile 11 is added to the queue and parent array, as it is next to tile 15.
        1. Tile 11 is passed and removed from queue.
    iv. In a recursion depth of 2, the function is passed 11, and tile 10 is added to the queue and parent array, as it is next to tile 11.
        1. Tile 10 is passed and removed from queue.
    v. In a recursion depth of 3, the function is passed 10, and tile 6 is added to the queue and parent array, as it is next to tile 10.
        1. Tile 6 is passed and removed from queue.
    vi. In a recursion depth of 4, the function is passed 6, and tiles 7 and 2 are added to the queue and parent array, as they are both next to tile 6.
        1. Tile 7 is passed and removed from queue, while 2 remains in the queue.
    vii. In a recursion depth of 5, the function is passed 7, and tile 3 is added to the queue and parent array, as is it next to tile 7. Tile 2 is passed and removed from queue, while tile 3 remains in the queue.
    viii. In a recursion depth of 5, the function is passed 2, and tile 1 is added to the queue and parent array, as it is next to tile 2. Tile 3 is passed and removed from queue, while tile 1 remains in the queue.
    ix. In a recursion depth of 6, the function is passed 3, and no tiles are added to the queue or parent array, as it has no neighboring tiles that aren't in the parent array. Tile 1 is passed and removed from queue.

x. In a recursion depth of 7, the function is passed tile 1, which is unvisited. BFS returns tile 1.

e. Retrieving Path
   i. In order to retrieve path from the the BFS, we use the parent array
      1. As shown in the diagram above, we first take the target tile, and add it to the stack.
         a. Stack: [1]
      2. Then we take the parent of the target tile, 1, which is 2
         a. Stack: [1, 2]
      3. Then we take the parent of 2, which is 6
         a. Stack: [1, 2, 6]
      4. This processes is repeated in a while loop until the tile added to stack is the current tile
         a. Stack: [1, 2, 6, 10, 11, 15]
      5. Since the stack is LIFO, the path is popped in the order
         15, 11, 10, 6, 2, 1
      6. In order to check which direction to go in, to reach the next tile in the stack, we use the if statements mentioned above in section V of "storage of maze"

E. Benchmarks
   a. Average speed [100 Maze Trials, (Low / High / Avg)]
      i. Maze size of 10 x 10, entire maze:
         1. Creates path for all tiles in (~7ms / ~36ms / ~24 ms)
      ii. Maze size of 80 x 80, entire maze:
         1. Creates path for all tiles in (~778ms / ~2201ms / ~1624ms)

VIII - Victim Detection

Introduction

This article is about the design process of our letter and color detection algorithms. It goes over the planning, implementation, and integration phases, along with other important steps.

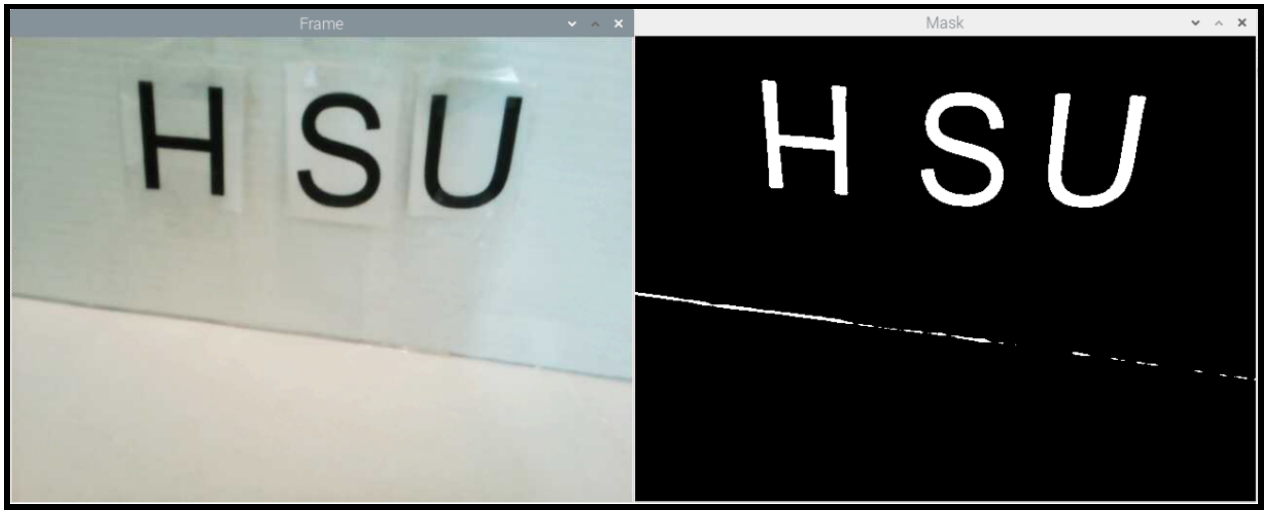
Project Planning and Development Cycle

1. We needed to perform letter detection in order to recognize the letters, H, S, and V.
   a. Possible options that we considered were KNN (k-nearest neighbors) or using various contours to determine unique aspects of each letter
   b. We decided to use KNN as it would allow us to use a wide variety of sample data to determine a variety of letter types, positions, and rotations. The contour plan seemed much harder to implement also as we had to use contours on specific corners and edges to differentiate letters.
2. We needed to perform color detection in order to recognize the colors, red, yellow, and green.

    a. For this program, we immediately decided on using an inRange combined with HSV values. We have experience using these OpenCV functions and it is easy to implement as well as being incredibly effective. Other ideas that we could have done was using a color sensor, but it made sense to use the camera we had for letter detection.
3. We needed to position the camera in a way where it did not see any parts of our robot and could see the letter fully from any position.
    a. We had several issues with this positioning
        i. We had a large tread movement system which took up a lot of space and blocked a good amount of camera vision in the bottom portion. This would especially affect it if the robot was closer to the wall.
        ii. The camera had to be fairly close to the rescue dropper so we would be in the rescue kit boundary and correctly detect victims
    b. We had our camera positioning high and tilted downward. This solved our issue with our large treads as it was able to see over it and downard. We also put it near the center of our robot. This made sure that our rescue kits got to the victim no matter where we put the dropper. Lastly, the angle allowed it to see a good portion of the tile.
4. We needed to be able to see left and right of the robot
    a. Possible options we considered were using a RaspberryPi camera and a USB camera, using two RaspberryPi cameras with the help of a compute model, or using one RaspberryPi camera with a mirror
    b. We decided to use a RaspberryPi camera and a USB camera as it was the simplest option. The compute module idea took up more space and the mirror idea was too difficult to implement. In the end, they were both more complicated than simply plugging in a USB and putting two cameras on each side of our robot.

Implementation

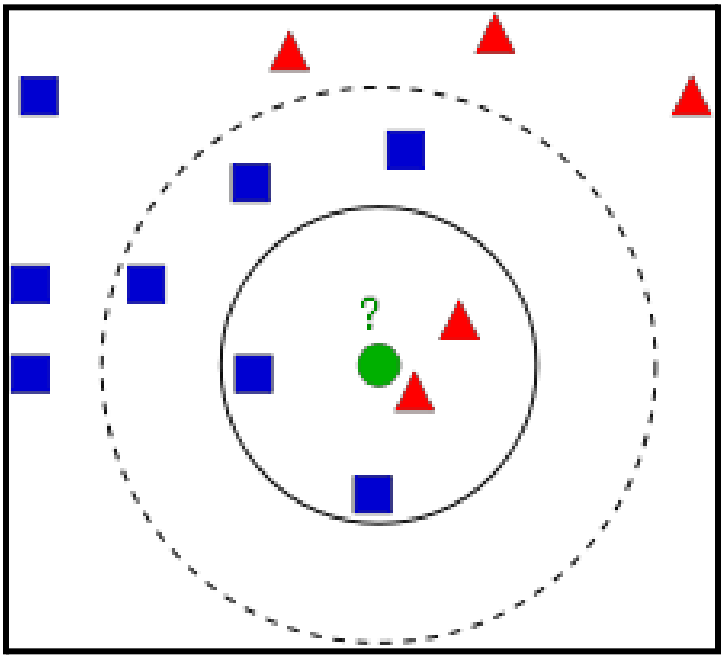
1. Letter Detection
    a. Seeing the Letter
        i. We used the Python provided OpenCV functions and numpy to do video capturing and frame/contour manipulation.
        ii. In order to actually distinguish the letter from its white background, we converted the frame to grayscale, blurred it, and used adaptive gaussian thresholding to highlight black and ignore other colors.
        iii. We changed the width and height of the frame from (640, 480) to (320,240) in order to increase program efficiency



b. Isolating the Letter
   i. After being able to see the letter, we used cv2.findContours to return all portions highlighted in the mask. We then sorted through the contours we received and returned the one with the largest area, which was normally always the letter.
   ii. We used minAreaRect to find the smallest rectangle that would go around the letter and extract it.
   iii. We altered the size of the letter to ensure that it would always be 30 by 30. We also record the minimum rectangle's corner points for future functions.

   iv. To further improve accuracy of letter detection, we used cv2.getPerspectiveTransform and cv2.warpPerspective to ensure that skewed letters would arrive straight and accurate.

c. What is KNN?
   i. KNN is an algorithm that is used for classification and identification. It does so by assuming that similar data values are close to each other. When trying to identify a new item, it uses a distance equation to determine what its closest points are, and identifies it based on its surroundings.



The green circle will be identified as a red triangle, due to its close proximity with other red triangles that have similar data values,

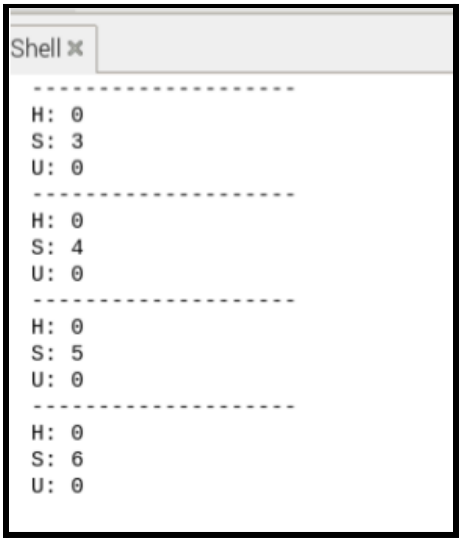
   ii. To use KNN, you need to have pre-existing data samples with labels, in order to properly compare and identify a new item.

d. Getting Sample Data
   i. To get sample data, we repeated the steps for seeing and isolating the letter. We created a program that would label each letter we saw properly and put it into a text file.

ii. To start off, we opened two files with the option “a+”. This file opening mode allowed us to append items to the file, meaning we could get sample data as many times as we wanted to improve our detection. The two files were for the actual data values and the corresponding labels.

```
features = np.loadtxt("/home/pi/Documents/KNN/trainingData/featuresFinal.txt", dtype = np.float32)
labels = np.loadtxt("/home/pi/Documents/KNN/trainingData/labelsFinal.txt",dtype = np.float32)
```

iii. To actually append the data values and labels, we used cv2.waitKey. This would allow us to type in either H, S, or U to identify the current contour as one of the letters.



iv. After we had gathered all the data samples we wanted, we would then alter the shape of the array they were stored in. The KNN functions require that data values and labels be stored in a one line array. We would then store the text file to save everything and conclude the program.

e. Detecting the Letters
   i. After seeing the letter and gathering sample data, it was time to truly detect it using cv2’s KNN functions.
   ii. The first step was to train our program using the sample data we gathered.
   iii. After, we would then repeat the steps for seeing and isolating the letter, in order to compare a consistent item with the data values previously collected.
   iv. We would then pass the item into cv2.findNearest which would classify the item based on its closest points. The amount of items it compared the item could be adjusted based on the functions parameter. The function returned 4 objects, though we only used the distance and result portion.

```
_,result,neighbors,distance = knn.findNearest(knnInput,5)
```

v. We found the smallest distance from a data value to the item and also returned the closest label of the data value using simple expressions. If the shortest distance was too large, we

would return no result. Otherwise, we would return the closest label which would identify the letter.

```
lowestDist = distance[:,0]

result = chr(result[0][0])


if lowestDist > 10000000:
    print("None of the Letters")

else:
    print(result)
```

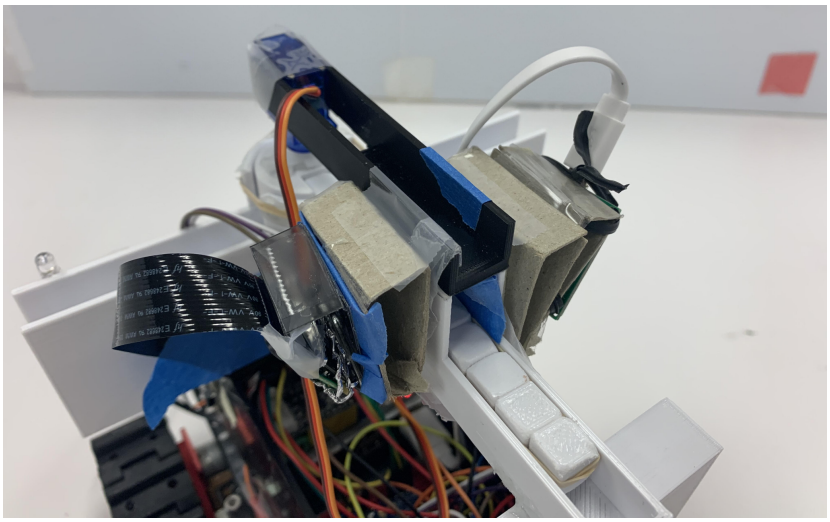
2. Color Detection
    a. Seeing the Colors
        i. To see the colors, we used a variety of HSV ranges combined with cv2.inRange. This returned various masks which highlighted the colors and ignored other colors
        ii. HSV stands for Hue, Saturation, and Value
        iii. The HSV ranges were grouped for each color in lists and the colors themselves were grouped in a list too. This allowed us to use a for loop to find each color instead of having to make a program for each.
    b. Detecting the Colors
        i. After masking the frames and highlighting the colors, we used cv2.findContours to actually separate them from the background.
        ii. We then found the largest contour out of the returned contours and if the area were big enough, we knew we had detected a color.
        iii. Depending on what position it was in the for loop, we could figure out which color we had actually detected

Integration

3. Combining Letter and Color Detection

a. Combining the letter and color detection code took a simple copy and paste.
b. We put the two programs together in a function one after the other. We made an assumption that there could not be more than one victim on a wall per tile. Based on this, we placed more importance on letter victims, guessing that color detection would be more likely to make a mistake.
c. We added code which returned True/False depending if it saw a victim or not and also returned the number of rescue kits the robot needed to give

4. Adding a Second Camera
    a. We added a USB Arducam to our robot to see walls on both sides.
    b. To add it in our program, we had two VideoCaptures with different indexes. When we restarted the robot, the indexes would sometimes change though it was an easy fix.
    c. We doubled the actions done in our letter and color detecting function. To do this, we had to make sure we had no repeating variables, and that we returned True/False and packages for both cameras.
    d. We had to flip the frame for the RaspberryPi cam but not the Arducam.



5. Sending the Information
    a. Because this program was done on Python, we had to send the returned values to the MegaPi (Arduino).
    b. Over serial, if we saw a victim, we sent information on what side the victim was on and how many packages were required (example: L3).

IX - Rescue Kit Dropper

Introduction

The following is the design process of our rescue kit dropper. The design process was how we planned and developed the rescue kit and implemented the software and hardware components of it. This includes the planning of it, the making of a specific design, The making of a CAD model, finalizing the model, and implementing the model.

Project Planning and Development Cycle

1. We needed to find a way to dispense the rescue kits to the left and right.
    a. Possible options that we considered were a continuous servo controlling a circle with 4 slots for rescue kits, a servo controlling a flap that would lean right or left to release the rescue kits, or a circle with 1 slot and a 180 degree servo motor rotating it left and right to drop out rescue kits.
    b. We decided to use a servo controlling a circle with one slot because it was the most efficient and accurate way to drop rescue kits out of the possible options we considered.
2. We needed to find a way to move the rescue kits towards a dispensary system.
    a. We considered using a rubber band system, a spring system, or a gravity system to dispense the rescue kits.
    b. We decided on using a rubber band system to dispense our rescue kits because it would be the most reliable system.
        i. A spring system would require a strong spring, which would coil up and take up a lot of space. It would also be hard to attach to the rescue kit dropper.
        ii. A gravity system would require us to stand a thin but tall shaft on top of our robot, which would not be practical for our robot.
3. We needed to find where and how to attach our rescue kit dropper to our robot and move the rescue kits further than our wheels.
    a. Possible design options we considered were using a flat roof and putting our rescue kit dropper on top of it, using a slanted roof with a flat top for our rescue kit to fit into, and using a large ramp and bars of support
        i. A flat roof would go on top of the shields of our robot, providing ample room for the dropper.
        ii. A curved roof would go on top of the shields of the robot, allowing the dropper to use the slanted surfaces as slides for the rescue kit.
        iii. A ramp would go in the front of the rescue kit dropper along with the dispense system, and a support bar would go in the back to make sure the dropper is stable.
    b. We chose to use a ramp with a support bar because it would save space for our robot and allow the cameras and wires to be connected overtop of the shields on our robot.

Software

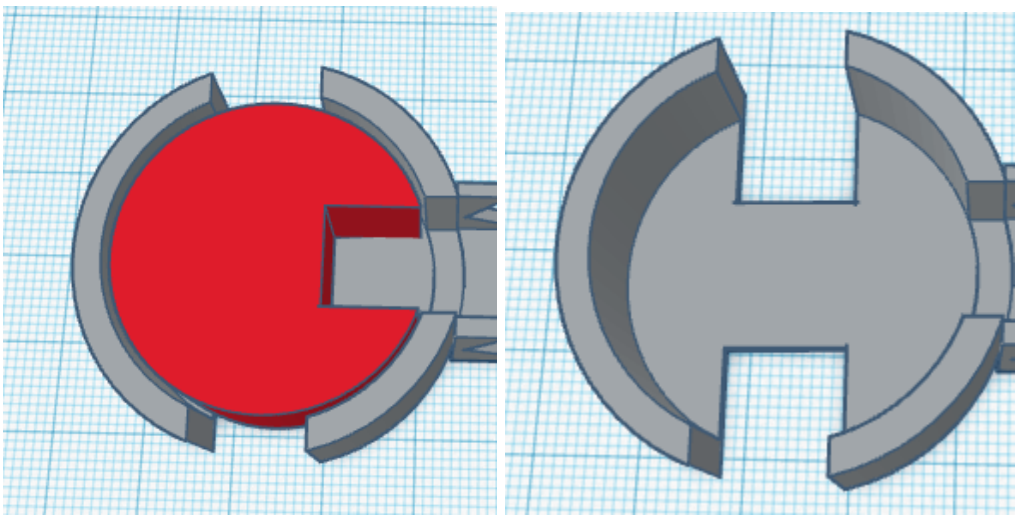
1. We had 5 main functions to control our servo motor: turnLeft, turnRight, midPos, stuckTest, and wiggle

c. stuckTest takes in a target value and reads the current position of the servo motor. If the current position is not within 1 of the target value, then the program will return true. Otherwise, it would return false.
d. The wiggle function takes in a target value and wiggles a little to the left and right in order to make sure rescue kits go into or out of the circular rotating mechanism. The amount of times wiggles is put in as a parameter to the function, and the amount it wiggles depends on the number of times it wiggles. At the end of the wiggle method, if stuckTest returns true, the boolean variable shouldRun is set to false. This is used to stop the rescue kit dropper from running if and only if the circular mechanism is stuck and cannot get free.
e. The turnLeft and turnRight methods turn the circular mechanism to the left or right 90 degrees to drop rescue kits out of holes on the outer shell. They both call the wiggle function to make sure the kits don't get stuck and move out well.
f. The midPos function turns the servo motor to face the shaft in order to allow rescue kits to fit into the slot on the circular mechanism. This calls the wiggle method in order to ensure that the motor doesn't get stuck. The amount of times it wiggles scales with how many rescue kits are left in the dropper because less droppers would mean less tension in the rubber band, making it harder for the kits to move forward.

4. In the setup of the code, we had to use an overridden version of the attach function of the Servo library to set the minimum and maximum pulse widths.
    a. At first, with the default pulse width minimum of 544 and maximum of 2400, our servo motor could only move a total of 150 degrees.
    b. We changed the minimum pulse width to 490 to allow the robot to turn the full 180 degrees.
    c. We adjusted the angle values - we used Servo.write(60) to turn 90 degrees due to the inaccuracy of our servo, Servo.write(173) to turn 180 degrees, and Servo.write(0) for 0 degrees.

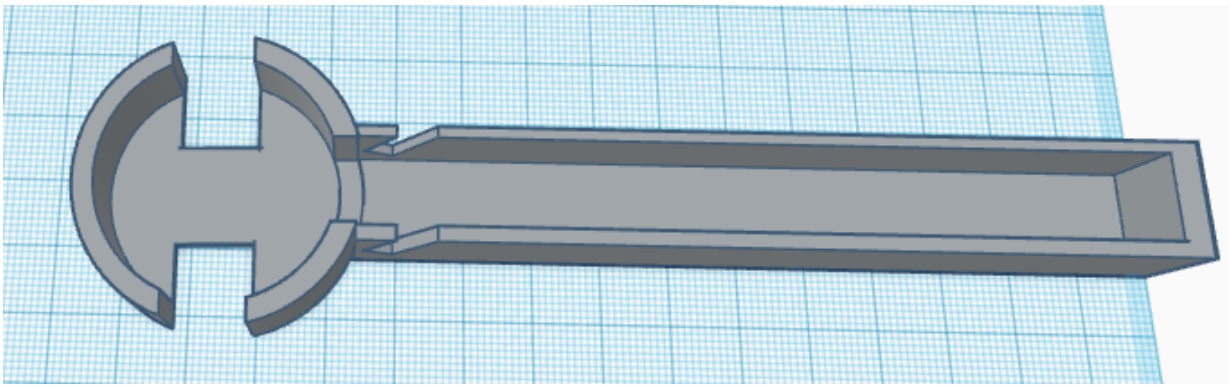
Hardware

Mechanical design and manufacturing

1. Design
    a. Dispenser system
        i. 
        ii. On our dispenser system, we would have an outer layer with 2 holes in the bottom. These holes would allow rescue kits to drop down onto a ramp.
        iii. We would have a circular object that would fit into the outer shell with a slot big enough to fit a cubic rescue kit. This would be where our rescue kits would be pushed into.

iv. We would have a 180 degree servo motor rotating the inner circular object, allowing rescue kits to drop down the holes on the outer shell.

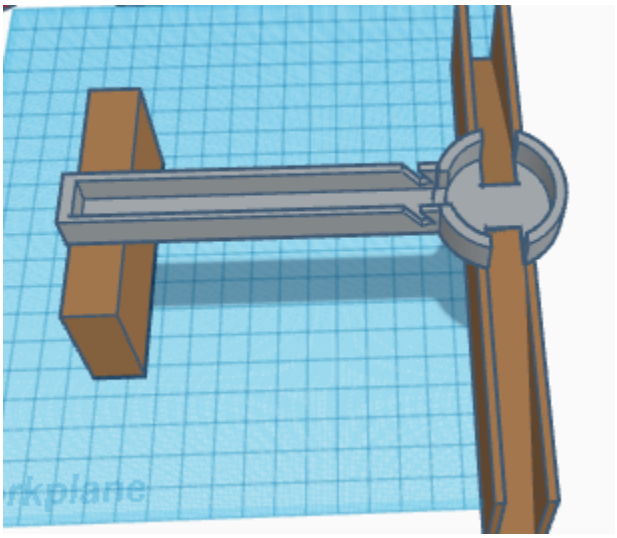
b. Rescue kit pushing system

i. 

ii. In the above system, a rubber band would be connected from the front of the rescue kit dropper around the circular part of the dropper into the slot on the shaft around to the back of it.

iii. Rescue kits would fit into the long shaft.

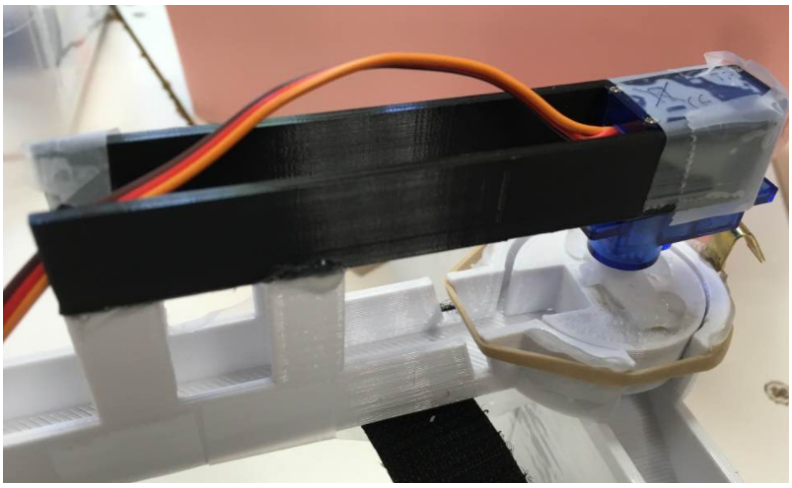
c. Ramp and Support system

i. 

ii. A 2 sided ramp would be at the front of the dropper, which would slide the rescue kits either left or right past the wheels of our robot.

iii. A block the same height as the ramp would be at the back of the dropper to provide support

d. Support and holding pieces in place

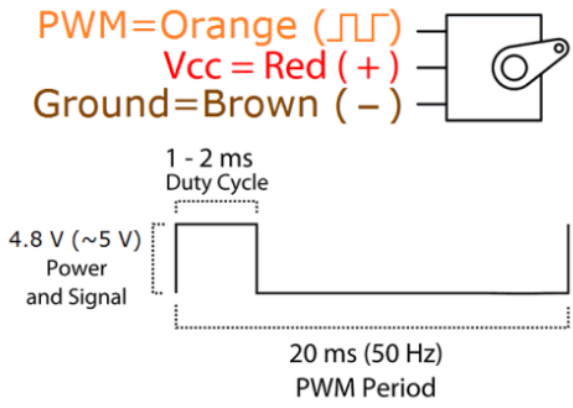
i. 

ii. We put bars going up in order to hold the servo motor in place, secured by tape so it doesn't go left and right when the servo rotates.

iii. Using hot glue, we made a ledge on the outer circular shell so that the rubber band could fit in it securely.

iv. We added velcro under the slot on the shaft so that after we load in the rescue kits, we could use the velcro to secure the rubber band, making sure it doesn't come out of the slot.

2. Electronic Design and Manufacturing
    a. We needed to use a motor to control our rotating mechanism left and right, and were considering using either a 180 servo motor, a continuous servo motor, or a metal gear shaft motor
        i. We decided to use a 180 degree servo motor due to its accuracy

b. 

c. The 180 degree servo motor turns either left or right a certain pulse length in microseconds, which is translated to a certain amount of degrees defined in the servo.write function.

d. The duty cycle of the servo motor is between 1 and 2 milliseconds depending on how much the motor turns, but the minimum and maximum pulse lengths could be redefined in the servo.attach function.

X - Future Interests

If We Had More Time

As of now, our robot is not completely optimized for completing every aspect of the maze, but we could have implemented many more features to resolve those challenges. We plan on competing in RCJ Maze next year as well, so next on our agenda would be to improve many parts of our robot and design.

1. We would make a smaller and more compact base for our robot. Our current base is just barely able to turn in tight spaces, leaving too little room for error. A smaller bot would not only increase our success rate, but also make the robot more stable as its center of mass would be more towards the middle.
2. We would implement colored tile detection so we can see black and silver tiles and run accordingly. Since we had such a time crunch this year, we were not able to completely integrate all of our code together.
3. We would also make an improved version of the rescue kit dropper, one that utilizes gravity instead of relying on a rubber band. We found it very tedious and time-consuming to continuously reload the rubber band, so having a system that can be reset easier would be very helpful in future runs.

XI - Conclusion

Summary

During the RoboCupJunior Rescue Maze competition, our team went through many challenges and had to come up with innovative solutions to help solve these problems. Overall, we learned to coordinate and manage our plans efficiently and were able to work together as a team to overcome many challenges. One of the biggest takeaways we got from this learning experience was the utmost importance of team communication and integration. We experienced firsthand how difficult it can be to merge two differing code segments in order to create a functional product. Our experiences have molded the path to a more successful and efficient future, in which we will improve our coordination, organization, and management.