Robutt’s Journey

Extended version

# Team Members

Riley Snook – Coder

Ben Tang – Coder

Jack Maxwell - Builder

Jamie Snow – Builder

All from All Saints’ College

# Problem definition

For open maze, we are required to design and code a robot that can autonomously navigate a maze to drop rescue packages, and to detect alive/dead victims. For this we needed a way to detect color, distance, temperature, a dropper system and a way to move around.

# Planning

For color we are using NXTCam5, for distance we are using a Time of Flight, and for temperature we are using infrared. Our dropper system is just a simple conveyor belt designed with tracks and triple pegs using skittles for the rescue package. As we wanted the ability to move laterally without needing to rotate the entire robot, we decided to use mecanum wheels even if it meant we needed a MultiMotor for the dropper system. It is slightly more complicated in terms of coding for the motors to get the robot to move in certain directions but once we got it, it would be more efficient and effective compared to tracks or just wheels. Riley and I(Ben) felt confident in our coding abilities to do code them effectively, so we decided to go with mecanum wheels.

## Roles of team

Riley would oversee coding the I2C and creating code that output values of the sensors. This would be due to the fact that he has greater capability and understanding of I2C and more experience of coding than Ben.

Ben would oversee coding the logic behind the robot. From the values from Riley’s code, they would be taken and inputted into the logic code for the correct actions to run.

Jack took the robot home along with parts and constructed the entire robot along with further smaller design changes to the building. This included building the robot at home and during the robotic sessions at school.

Jamie created concepts and designs for the robot, as well as soldering, and wiring the parts together. He was also who we went to first when we wanted something 3D printed, however in our final design no 3D printing was used.

# Building design 1

## Robutt 1.0

Last year we participated in the robotics maze competition, and it was a disaster. We were unprepared, unorganized, and extremely rushed. Our robot barely made it off the first tile, with its wires blocking the sensors and the tracks turning randomly. We took this defeat as inspiration for this year, hopefully to improve and build on our knowledge.

As soon as states ended last year we began to work on a new and improved maze robot. As we examined what had gone wrong a countless number of problems became clear to us.

## Problem 1 The sensors were not placed in The right spots

The rotating sensor was placed too high and too far forward. This made it very difficult to detect victims if they were not placed directly in the center of a tile. As such we overcame this through our coding. Instead we just stopped the robot slightly behind the center to check for victims both alive or dead, before moving into the center.

## Problem 2 The sensors were not suited to their job

We were using an ultrasonic Ev3 sensor last year but upgraded to a mindsensors NXT-v5. This improved what we could detect and was vital for finding the colour on the victims. Although it would be much harder to code for and for us to wire into the robot, as it was an I2C sensor, when we got it work, it was infinitely better as it allowed for color detection for multiple colors and for different light levels on the day.

## Problem 3 The wire of the heat sensor kept getting caught

This made it impossible for the rotating sensor to fully rotate, thereby limiting it’s field of view, this development caused a multitude of issues as it could not fully detect the intricacies of the maze. It meant that the sensor block wouldn’t rotate the entire way for one direction, but we overcame this by just coding for it to rotate with higher force.

## Problem 4 ineffective robot tracks

Our initial design used tracks to traverse the maze, this was useful to overcome certain obstacles but ‘track killers’ proved immensely difficult for this system. This is why we chose to use mecanum wheels that were more durable and could move sideways, these wheels also were able to deal with the track killers. We also, from previous knowledge from Line Rescue, that track does not work on slopes but mecanum wheels would.

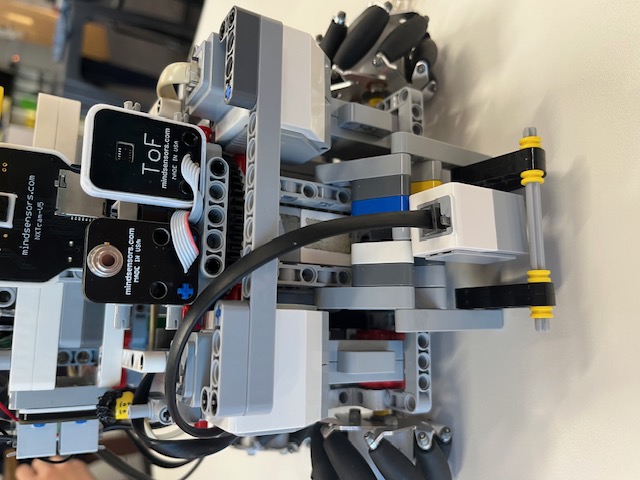
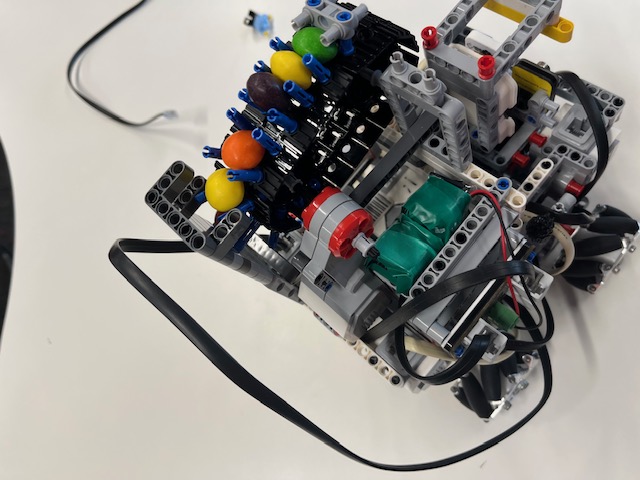
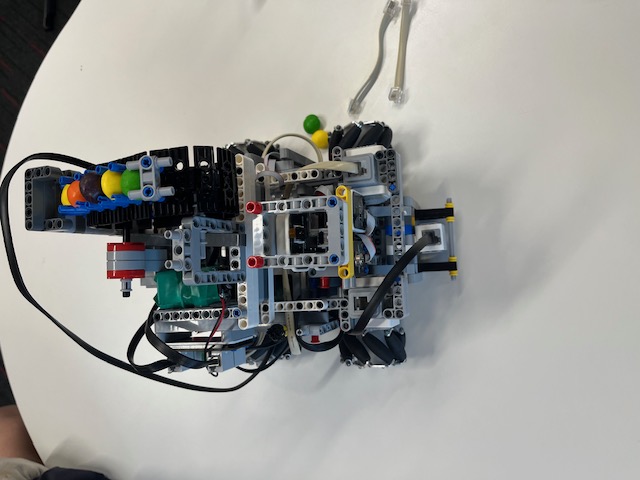
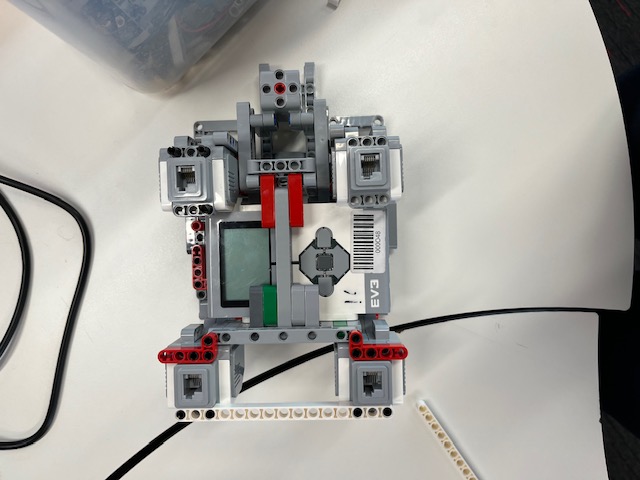
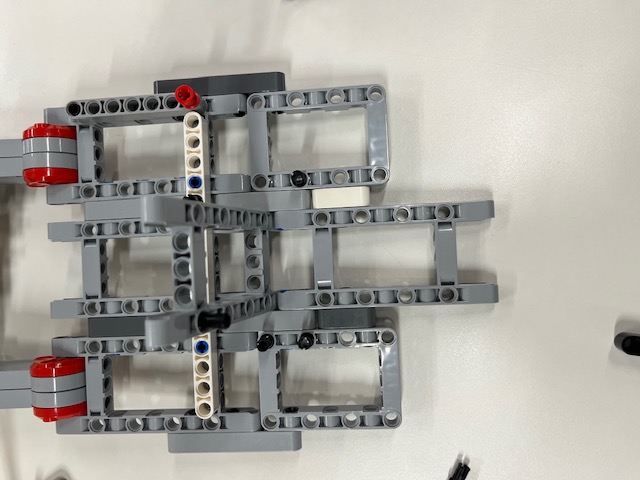
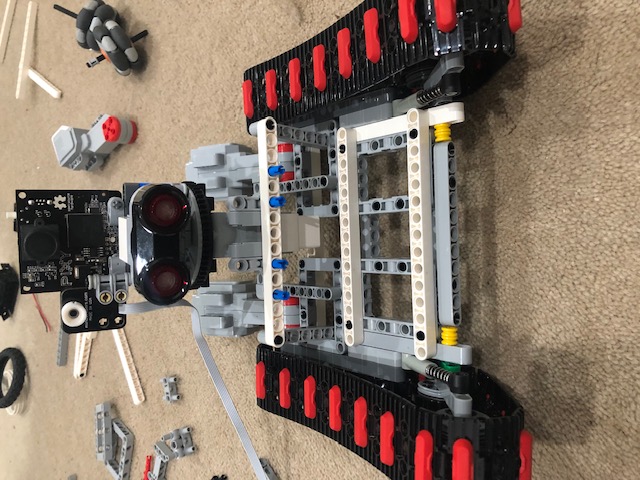
## Problem 4.5 Fitting 4 motors

Trying to fit 4 motors onto the robot while allowing space for it to turn and move around in the maze was a massive difficulty as there was not much space for us to fit 4 motors as well as all the other sensors and brick into a compact design. There was a large issue with this as they spent several weeks trying and testing for the best design. However, Jamie and Jack managed to get it to work, making the robot as compact as they could.

## problem 5 Dropper system

Initially we had a 3D model for a dropper system printed out by Jamie for us to use, however we soon realized that it was not very reliable and at times would get stuck and not properly dispense the rescue packages. Instead, through a conveyer belt of track and pegs, we made a dropper system that although looks rather rudimentary and ineffective, it works really well and rather reliable as each time, we would just need to move the motor a set distance to drop the rescue packages.

# Timeline of the Build



# Coding The robot <(x\_x)>

We did not have much code for the robot last year, so we started from scratch. The code can be broken into 3 different sections of code. The dropper, movement and the sensor block. The code that run continuously in the loop is mainly, pretty much all of it, full of subs with logic. This is in order to keep the running code as short as possible, and it also allows Riley and Ben to collaborate more effectively and more easily with each other.

## Dropper

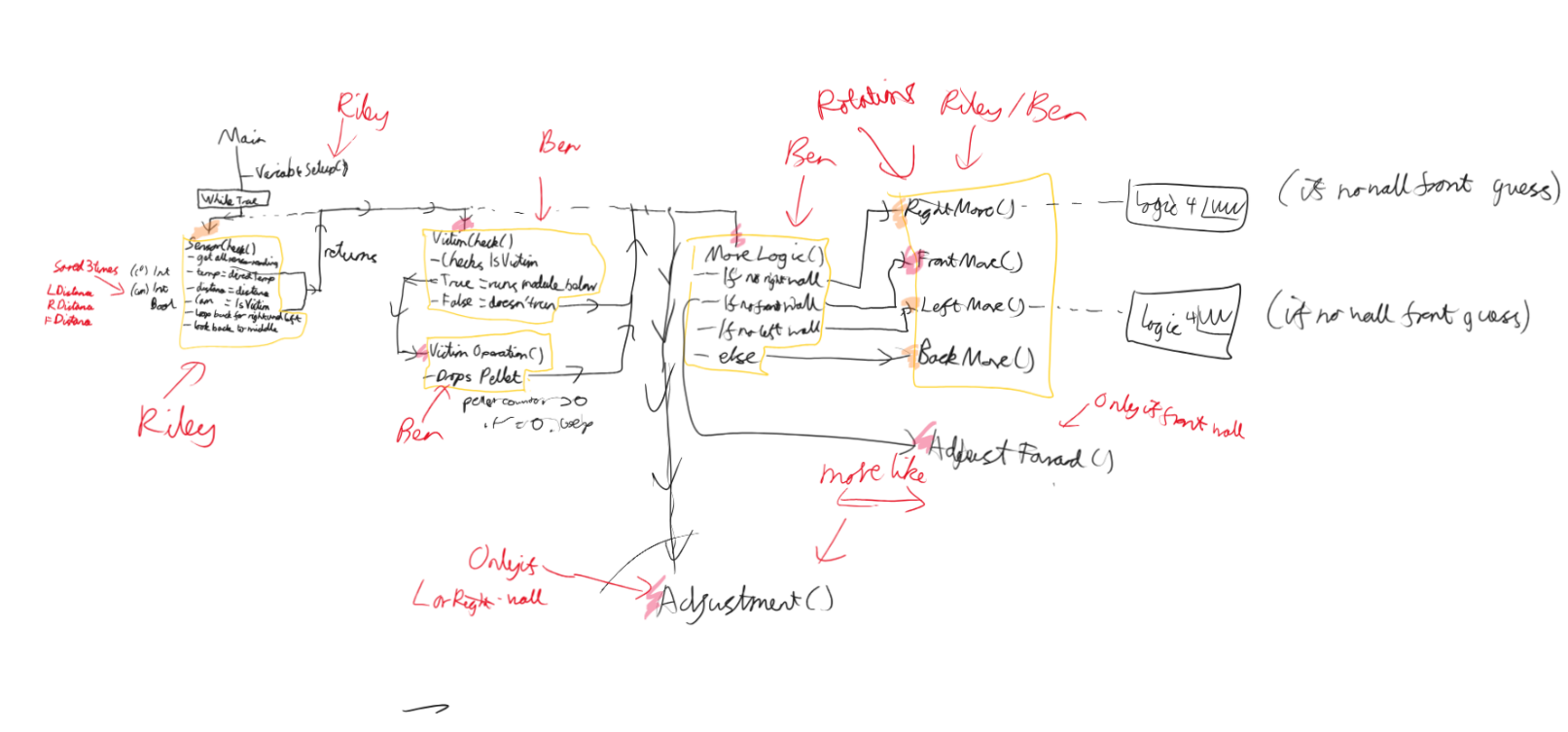
Out of the three, dropper was the simplest and most basic to code. All there was to moving the dropper was a single line, given the right conditions and logic, to move the motor a set distance for a set time to drop 1 rescue package.

## Sensor Block

The movement of the sensor block at first was so difficult to code as the MultiMotor would not work. This was due to a wiring issue, but Jamie managed to solder the wires on properly with each other. The movement of the sensor block after it worked was “relatively” simple, sending code through I2C communication. We ran values through certain registers to get movement of the sensor block. As we needed sensor readings of the left, front, and right walls around the robot, all we needed to do was rotate the sensor to the left, right and center to get the values of each tile. Communicating with the I2C sensors, was a great difficulty as we weren’t sure what values to run into the I2C slaves, and we couldn’t find the correct addresses of each sensor for them to work. The main issue with the all the I2C sensors and the MultiMotor was the initial struggle of trying to establish communication with the sensors and motors. However, when we did finally figure it out after so much trouble, it was a lot easier, and everything began to fall into place. From the start we knew that the struggle would be at the beginning, trying to establish communication with the new sensors and the new MultiMotor we added. From that point onwards, it was just a matter of debugging and testing to iron out the values of our turning, speed, time and other variables.

## Movement

From a mentor/older student, we learnt that the use of mecanum wheels was essentially vectors. To move certain directions, we learnt that it was just the combination of how each mecanum wheel was moving, the direction and how fast they moved in comparison to each other. However, for our robot, it was a little different as the back two motors were attached in the opposite direction to the two motors at the front to allow for the robot to be more compact. For example, to go forwards, all wheels need to forward. But when coding for this, we coded for the front two wheels to go forwards, and the back to turn backwards as they were the opposite way around compared to the two at the front. The coding was rather simple as the four motors that contributed to the movement of the robot were directly connected to the brick so we didn’t need any I2C communication. We just used arrays for the motors so that it allowed us to start all four motors at the same time. This allows us to be able to have all four motors moving at the same time so that we wouldn’t drift out of a straight line. If we didn’t start all four of them at the same time, it would cause the robot to drift slightly as some motors would be starting earlier than others.

Our plan for the code along with the delegation of what Ben would code and what Riley would code:

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquementOur list of variables while planning out the code for before states:

How it turned out in the end:

'Fixed Values

motorArray[1] = "A" 'Front Left

motorArray[2] = "B" 'Front Right

motorArray[3] = "C" 'Back Left

motorArray[4] = "D" 'Back Right

moveSpeed = 30 'Moving Speed

sensorSpeed = 100 'Sensor Block Moving Speed

dropperSpeed = 20 'Dropper Moving Speed

dropperTime = 0.12 ' How Long The Dropper Moves For

fixedTurn = 435 'Turning distance, get past first wall

radius = 30 ' radius of wheel

distanceFudge = 0 'Fudge factor for moving foward

circumference = radius\*2\*Math.Pi ' Circumference of wheel

degreeDistance = circumference/360 ' Distance of one degree

fixedMove = (300/degreeDistance)+distanceFudge ' Degrees to move one tile

maxDist1 = 250 'Max Distance The Wall Should Be If It Is 1 Wall Away (Directly In Front)

tempVariance = 2.5

baseSpeed = moveSpeed

Done = "False"

'Readings in Code

distSide = 0 ' Distace of side wall for adjustments

distFront = 0 ' Distance of front wall for adjustments

averageDirectTemp = 0 'Average Temp Of Non Body Walls

leftDirectTemp = 0 'Temp Of Left Wall

rightDirectTemp = 0 'Temp Of Right Wall

frontDirectTemp = 0 'Temp of Front Wall

wallCounter = 0 ' counter of walls seen, for averag temp

pelletCounter = 10 'How Many Pellets Left

isVictim = "False" 'Does Camera See Victim

tempVariance = 10 'Range For Dead Or Alive Victim In Temp

lDist = 0 'Dist To Left Wall

rDist = 0 'Dist To Right Wall

fDist = 0 'Dist To Front Wall

moveDist = 0 'Distance Needed for Adjustment

lowestDist = 0 ' lowest distance from the wall

knownDistSide = 152 'How Far From Wall The Robot Should Be From Side

knownDistFront = 116 'How Far From Wall The Robot Should Be From Front

lWall1 = "True" 'Is There A Left Wall

rWall1 = "True" 'Is There A Right Wall

fWall1 = "True" 'Is There A Front Wall

targetDist = 0

targetTemp = 0

numObjects = 0

neg = 1

'\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

' I2C  INIT

'InfraRed

irPort = 1

irAddr = 21

irReg = 68

irWriteByte = 1 + 0

irWriteData = Vector.Init(irWriteByte, irReg)

irReadByte = 2

irReadData = Vector.Init(irReadByte, 0)

'Time of flight

tofPort = 2

tofAddr = 1

tofReg = 66

tofWriteByte = 1 + 0

tofWriteData = Vector.Init(tofWriteByte, tofReg)

tofReadByte = 2

tofReadData = Vector.Init(tofReadByte, 0)

'Camera

camPort = 3

camAddr = 1

camReg = 66

camWriteByte = 1 + 0

camWriteData = Vector.Init(camWriteByte, camReg)

camReadByte = 1

camReadData = Vector.Init(camReadByte, 0)

'MultiMotor Init

I2CMotPort = 4

I2CMotAddr = 3

I2CMotReg = 0

I2CMotWriteByte = 0

I2CMotWriteData = Vector.Init(2, 0)

I2CMotReadByte = 0

I2CMotReadData = Vector.Init(2, 0)

M1Rotations = 0

M2Rotations = 0

MVoltage = 0

MCommand = 0

M1Rotate = 0

M2Rotate = 0

M1Speed = sensorSpeed

M2Speed = dropperSpeed

M1Seconds = 0

M2Seconds = 0.12

M1Brake = 1 '  1 = Brake, 0 = Float

M2Brake = 1 '  1 = Brake, 0 = Float

M1Ramp = 1 ' 1 = Ramp, 0 = No Ramp

M2Ramp = 1 ' 1 = Ramp, 0 = No Ramp

M1Wait = 0 ' 1 = Wait For Completion

M2Wait = 0 ' 1 = Wait For Completion

M1Status = 0

M2Status = 0

M1StatusBits = Vector.Init(8, 0)

M2StatusBits = Vector.Init(8, 0)

M1StatusString = "SRPCBOTL"

M2StatusString = "SRPCBOTL"

MRotate = 0

MStatus = 0

bits = 0

# States

* Came 1st

## Issues that we learnt about our robot and its coding from states

* Dodgy sensor block movement.
* The turns were not as precise as we liked them to be and we would drift.
* We didn’t have a way to detect if we were tilted and stuck on obstacles.
* No mapping.
* Dropper system was too bulky and inefficient.

## We broke it down into 3 different sections to prepare for nationals

* Mapping implementation
* Adjustments with our movement
* New building

## Mapping implementation

For mapping we create a 25 by 75 array with each “tile” being represented by a 1 by 3 part of the array. The 1st value is its wall orientation, 2nd if there is a victim, and 3rd for the frequency that we have visited that tile. We combine the gyro and the time of flight(tof) with a calculation that produces a hex value that represents where the walls on that tile are. We assign certain values for certain orientations of the wall. The 2nd was the simplest, and it was just to ensure that we were not dropping care packages over and over again for the same victim. The 3rd was to be used for our preferential follow, so that we can explore more of the maze and not loop infinitely with a right wall follow. However, this wasn’t a really major part of running the course so if it ended up no working, we would just exclude this from the code on the day

## Adjustments with our movement

We noticed that the sensor block wasn’t moving to its full rang of motion, and its wires were getting caught. As such, we redesigned the front of the robot, ensuring that it wouldn’t happen again, and that it could move more precisely and more accurately. Some movements just needed to be adjusted more thoroughly, and that was just simply changing some fixed values of the code.

We also added in a gyro, compass and accelerometer (3-in-1) however, the accelerometer gave us the values in a really weird way and we couldn’t understand which bit or byte gave what information, leading us to not use it after many hours working on it. The compass we decided not to use as the magnetic field can be altered as per the rules. The gyro we are mainly using to determine if we are tilted backwards or forwards so that we know if we are on a ramp or not to not mess up the mapping. We tried using it for turning, however there was too much drift and we were turning more reliably before anyways with our old method of using the tof so we decided to not use the gyro for turning.

## New building

We dropper system did in fact work, however after redesigning the front, it kept on getting caught with the sensor block which led us to change it. We moved to a concept which we disregarded at the start as we thought that it wouldn’t be a viable option as the care packages kept on getting caught in the dropper system. However, we 3D printed the care packages this time around, ensuring that they would work as we wanted them to. We also made it more compact and less clunky, and hopefully that will help with the reliability of the dropper.