SearchAndRescue.ino   
contains all our functions that we call to allow us to move around the map. This includes forward and backward movement, rotating or scanning rooms. It also has several switch statements that it uses to decide which movement to do based on the user input.

The two biggest issues I had with the project was the forward movement due to the motor’s power being out of sync, one was more powerful than the other. This meant that it would curve into a wall and wouldn’t go straight. The second was the turning not being precise, it took me a week to finally come to a realisation that it’s impossible with the hardware we had to get it perfect all the time. In this part it explains the steps I took to get the forward movement working and how I got the rotations working.   
  
P.S – In the READ\_ME I’m only discussing here the different approaches I took, where I found them and the ones I have settled on.   
  
I don’t believe I need to talk about my code here. This can be seen within the actual code itself since the naming of the variables explain what each one is used for and part of the code that don’t have naming have comments explaining what it does. Again, this is only discussing the approaches I took, where I found them and the ones I have settled on.

## Moving forward

I originally used just the motor speeds and delays to go across the map this worked but was inconsistent and was prone to getting stuck on walls or being stuck in the corner of a corridor when used with the line sensors. It was also not very accurate, so I had to find another alternative.

I tried numerous attempts in hardcoding the motor that had less power however again it was unreliable. I came across another method that adjusted the speed according to what angle we are currently, we would get this angle from the gyro. <https://github.com/pvcraven/zumo_32u4_examples/blob/master/HeadingHold/HeadingHold.ino>  
More information on it here. However, it was still slightly curving, and I wanted to be as straight as it can be so I looked again to see what I could find on the web.

I found a method that used the encoders and it was this method that I have used in all of the movement and rotation functions. Using the encoders to work out the correction, depending which one you want to compensate for. Example if it's the right motor. You would multiply the right encoder value with a number between 0.9 to 1.1, this is to adjust for any minor curves, I believe mine is 1.015. You then take this value off the left encoder to give us our correction, you then add this correction to the current speed of the motors. Then add the result to the speed of the right motor. This will apply the correct compensation, either adding or deducting speed away to keep us in a straight line. This has worked very well for me and so I've used it.  
http://www.abrowndesign.com/2017/02/25/zumo-32u4-synchronize-motor/  
This was the handy guide that helped me correct my motor issues. I applied this also to my rotations which works most the time. I believe the reason the rotations aren’t always accurate is due to the gyro.

## Line Sensors

I also had to make sure the Zumo would stop when it hits the wall this was when we used the line sensors to detect if it was over a line or not. Luckily, I didn’t have to much of an issue with this thanks to following the Maze Solver in the zumo32u4 examples. This allowed us to calibrate the line sensors allowing for more precision for the sensors when it goes over a line. I did have to add a delay though once it hit a right or left sensor since the middle sensor is further back, so it needs time to get into position. This allows us to know if we hit a wall straight or if we hit the wall at an angle.

The TurnSensor.cpp and TurnSensor.h was taken from the Maze Solver example. This allows us to calibrate, update and reset are gyro. It also converts it into an angle which we can use to rotate our Zumo. This was necessary for the calibrate line sensors where it had to rotate left and right.

## Rotations

Once I realised that it wasn’t possible to get the turning 100% accurate all the time the Rotations was fairly straight forward I used the Gyro to work out what angle I am at, I needed to make sure we updated the angle each loop so the gyro knows how much it has turned by. You then want to make sure you reset the gyro back to zero once you're finished or your gyro data may become invalid. Only do this on any rotations and not moving forward or reversing etc. it messes up the values.  
<https://github.com/pvcraven/zumo_32u4_examples/blob/master/TurnExample/TurnExample.ino>

^I used this example in my code, I did use the Maze Solver example but found this one better.

I had tested rotating by just using encoders which is commented out in my code somewhere and delays but again the same issue they we're not accurate enough and the Gyro was the one that proved more reliable but still isn't 100%. I applied the encoder speed correction to the Gyro to compensate for the power difference in the motors that had improved the accuracy by a small margin.

## Proximity Sensors

The proximity sensors were probably the simplest to get working, this allows us to know if there is an object that has been detected in a room. I had to play around with the Proximity Sensor Threshold, but I found 5 - 6 was the best. I used the SumoProximitySensors Zumo32u4 example to give me an understanding of how to apply it and was able to get it working fine on the first try.

## BuzzerAndLED.h and BuzzerAndLED.cpp

Contains the functions that allow our buzzer to play a tune and allows us to have turn on or off our LED lights on top of making them flash I was able to understand how to use the LED lights by following the example BlinkLED and I looked at the class reference for the buzzer it is also where I got my tune from. https://pololu.github.io/zumo-32u4-arduino library/class\_zumo32\_u4\_buzzer.html#ab72bde97ceceef8705f1bbaeccb970db

## SerialMessages.h and SerialMessages.cpp

This is where we store the messages for our code. I realised in my code that the messages are repeated a lot, so I condensed them down to save memory. (Dynamic Memory is now down to 54%. it originally 86%, was a good choice doing this). Remember and this is very important, Serial1 is what we use to communicate to the XBEE and Serial is what we use to communicate through the USB cable.

MotorSpeedAndDuration.h and MotorSpeedAndDuration.cpp   
Provides the variable speeds for the motors, sets the time limit for any delays and provides functions that set the motor speeds. I investigated the class reference mo32u4 Zumo motors to understand how it all works and helped by the Maze Solver example. More information in the code but this explains what this class and header does.

## GUI

The most difficult part about the GUI was figuring out how to get the GUI to read incoming and sending messages from the serial. Luckily, I found a very basic tutorial that allowed me to create a very simple version. There was a more advanced version that allowed you to check every COM port to see if there was one in a dropdown and would message you if you were connected but for what we needed I went simple instead of over engineering it.

<https://www.instructables.com/id/C-Serial-Communication-With-Arduino/>  
^ the example I followed.

I used COM6 on my laptop to read incoming messages from the XBEE, I also use windows forms to create my GUI and since I’ve used it before it was straight forward. To read messages I use a stop timer that gets turned on when I connect to the serial port. The timer will read incoming existed messages and is filtered so that’s how I get messages on text box.