Mechanical & Industrial Engineering
UNIVERSITY OF TORONTO

CONTEST 3 REPORT

UNIVERSITY OF TORONTO

MIE443 - MECHATRONICS SYSTEMS: DESIGN AND INTEGRATION

# Finding and Interacting with Emotional People in an Unknown Environment

*Authors:*

Project Group 18

Farhan Wadia — 1003012606

Henry Cueva Barnuevo — 1003585122

Yilin Huang — 1003145232

*Professors:*

Prof. Goldie Nejat

Dr. Meysam Effati

*Head Course TA:*

Pooya Poolad

*Grading TA:*

Cristina Getson

April 17, 2021

## Table of Contents

**1.0 Problem Definition** **1**

**2.0 Functional Requirements** **1**

2.1 Objectives 1

2.2 Constraints 1

**3.0 Competition Strategy and Emotion Responses** **2**

3.1 Detected Emotion - Anger 4

3.2 Detected Emotion - Neutral 5

3.3 Detected Emotion - Fear 5

3.4 Detected Emotion - Sad 5

3.5 Detected Emotion - Happiness 6

3.6 Detected Emotion - Surprise 6

3.7 Detected Emotion - Disgust 6

3.8 Emotion Classification Model 6

**4.0 Sensor Design** **8**

4.1 Microsoft Kinect RGBD Sensor 8

4.2 Odometry 9

4.3 Bumpers 9

4.4 Inertial Measurement Unit (IMU) 9

**5.0 Controller Design** **10**

5.1 High-Level Control Architecture 10

5.1.1 Frontier Exploration 11

5.1.2 Sending move_base commands (inspired by Contest 2) 11

5.1.2.1 One frontier exists and is red 11

5.1.2.2 Multiple frontiers exist and all are red 12

5.1.3 Forced movement and random rotations (inspired by Contest 1) 12

5.1.4 TurtleBot emotion output using robotReaction 12

5.2 Low-Level Control Architecture 13

5.2.1 Description of navigateNearby 13

5.2.2 Emotion classification 13

**6.0 Future Recommendations** **14**

**References** **15**

**Appendices** **16**

Appendix A - Complete C++ Code - contest3.cpp 16

Appendix B - Complete C++ Code - explore.cpp 30

Appendix C - Complete C++ Code - navigation.cpp 36

Appendix D - CNN Training Code 37

Appendix E - Attribution Table 39

# 1.0 Problem Definition

This problem is the third and final in a series of three contests related to the control and navigating capabilities of autonomous robots. For this third problem, a simulated TurtleBot will be required to find 7 victims in an unknown disaster environment by autonomously driving around the map and dynamically map this environment with frontier exploration as it traverses the area. Once the TurtleBot meets a victim, it should be able to identify the emotional state of the victim and react accordingly, while relaying information about a disaster evacuation, which we have chosen to be an evacuation due to a nearby wildfire. For navigation, a dynamic map and frontier-based exploration will be generated and implemented using gmapping, based on sensory information from the Kinect sensor, bumpers, and odometry; this is described in more detail in Section 4.0.

# 2.0 Functional Requirements

The objectives and constraints for this contest are discussed in the subsections below.

## 2.1 Objectives

In accordance with the contest requirements stipulated in Section 2.2, the following objectives have been identified:

1. The TurtleBot should maximize the amount of area it travels over and maps
2. The TurtleBot should be able to identify walls and boundaries visually (ie. using the RGB sensor) as well as physically (i.e. bumpers). Both should aid in the TurtleBot's ability to navigate the environment.
3. The TurtleBot should map the environment in the shortest possible time, minimizing the amount of time exploring the same area.
4. The TurtleBot should be able to accurately identify the emotion of the victim and react accordingly
5. The TurtleBot should be able to use multiple sensors to estimate its state as a redundancy if any single sensor gives faulty readings.

## 2.2 Constraints

The problem itself has several constraints, including:

1. The victims will each have one of 7 emotions, the robot must respond to each victim's unique emotion in a unique way
2. The robot's reactions must be made up of 3 primary emotions, 3 secondary emotions, and one that is either primary or secondary
3. The robot must react with a combination of visual and auditory stimulus, with the message of evacuating relayed to the victim
4. The TurtleBot must both explore and locate victims in the environment in a maximum of 20 mins.

5. It must be autonomous and use ROS gmapping libraries to create a map dynamically
6. The emotion classifier's CNN structure can be modified, but no additional software packages can be installed
7. The robot will be placed at a location of the TA's choosing and must begin navigating from there
8. The team must not access the emotions in "detectedVictim.txt", which holds the ground truth of the victim's emotion.

## 3.0 Competition Strategy and Emotion Responses

As stated in the contest constraints in Section 2.2, there are 7 victims, each in a unique location of the house, as shown in Figure 1. The robot is unaware of where the victims are and must explore this unknown environment autonomously. The TurtleBot relies on frontier exploration, explained in Section 5.1.1, and detects a victim when the TurtleBot gets within a 1 m radius of the victim. The control strategy our team implemented is largely behaviour-based control, which is discussed below, as well as in Section 5.1.

Behaviour-based control is well-suited for this task since it requires the robot to interpret sensory information obtained from its sensors, but it also does not need to sense and immediately act as it would in reactive control since there are no obstacles or moving objects in the environment (the victims are static). Below in Figure 2 is an example of the increasing level of competence of the robot, where our TurtleBot in this contest can accomplish the bottom five tasks (avoid obstacles, explore, generate maps, monitor for change, and identify people).

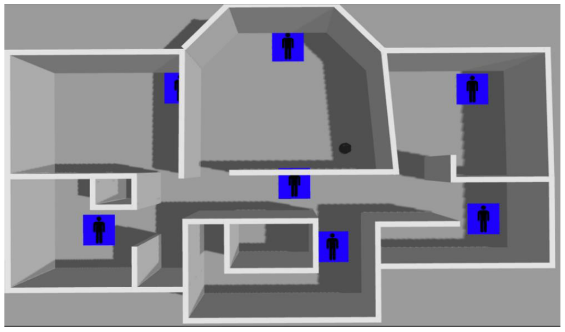


*Figure 1: Sample locations of the victims that the robot must navigate to*

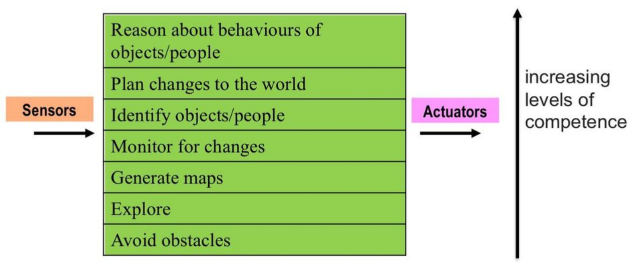

Sensors
Reason about behaviours of objects/people
Plan changes to the world
Identify objects/people
Monitor for changes
Generate maps
Explore
Avoid obstacles
Actuators
increasing levels of competence

*Figure 2: Examples of the increasing level of competence of the robot, adapted from lectures presented in MIE443 by Prof. Goldie Nejat and Dr. Meysam Effati*

As stated in the contest constraints in Section 2.2, there are seven victims, each of which has a unique emotion, as shown in Figure 3 below. The TurtleBot must then react to each of these emotions with one unique emotion (i.e. it can only react with sadness to fear, and cannot react with sadness to any other detected emotion). As stated in constraint 2, the robot must also display 3 primary emotions, 3 secondary emotions and 1 that is either, drawing from a list of 12 total possible emotions, shown in Figure 4. To ensure this requirement is met, we must determine what constitutes a primary and secondary emotion.

A primary emotion is one that arises naturally in the body, there is no control of 'choosing' to feel the emotion, as it arises automatically. Psychologists have classified there to be eight primary emotions which are: anger, sadness, fear, joy, interest, disgust, surprise, and shame (embarrassment) [1] . A secondary emotion is one that arises in response to feeling a primary emotion and often can require some more active thought. Secondary emotions do not occur naturally, and are learned through families, culture, and others. As the robot does not have this cultural influence, it was up to the team to decide what constituted a secondary emotion for the TurtleBot. A summary of the robot's emotion reactions and their classifications are shown in Table 1.

Each of the robot's reactions has two multimodal methods to convey the emotion and message. There is a visual cue from the image the robot displays. The image shows a human reacting with the emotion the robot is feeling, since it has been shown that humans empathize more easily with those that look most similar to them [2]. In our case, the person on the images is a team member, however, this can be adapted to be someone the family is familiar with so the message seems like it is coming from a caring family member or friend. The image also shows a solid background colour that matches the emotion the robot is feeling, which are described in detail in the sections below. Emotion classification is discussed further in Section 3.8.

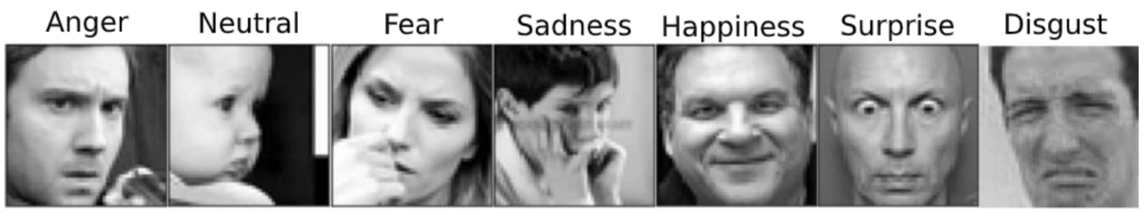
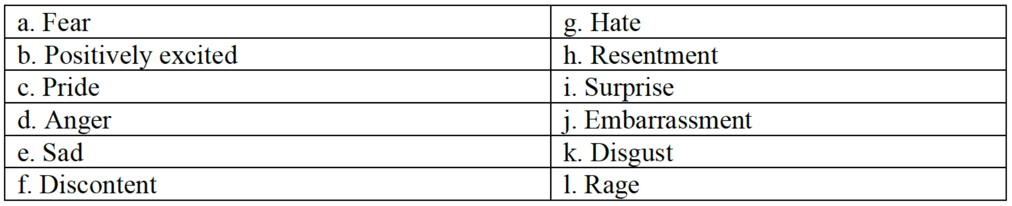


*Figure 3: The seven possible emotions the victims will show, which the TurtleBot can detect and classify*



| | |
|---|---|
| a. Fear | g. Hate |
| b. Positively excited | h. Resentment |
| c. Pride | i. Surprise |
| d. Anger | j. Embarrassment |
| e. Sad | k. Disgust |
| f. Discontent | l. Rage |

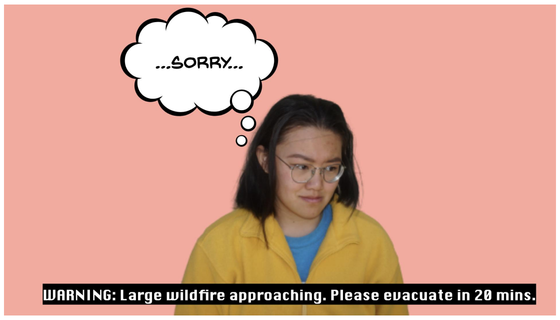
*Figure 4: The table above is a screenshot from the contest report showing the total list of 12 emotions the team could draw the robot's emotions from*

*Table 1: Summary of the victim's reaction and the robot's reaction, the classification of that emotion, and the main colour of the image displayed by the robot*

| **Victim's Emotion** | **Robot's Reaction Emotion** | **Robot Reaction - Emotion Classification** | **Robot Image: Colour Displayed** |
|---|---|---|---|
| Angry | Embarrassment | (1) Primary | Pale pink |
| Neutral | Pride | (1) Secondary | Purple |
| Fear | Sad | (2) Primary | Blue |
| Sadness | Positively excited | (2) Secondary | Yellow |
| Happiness | Resentment | (3) Secondary | Red |
| Surprise | Surprise | (3) Primary | Orange |
| Disgust | Disgust | (4) Primary | Army green |

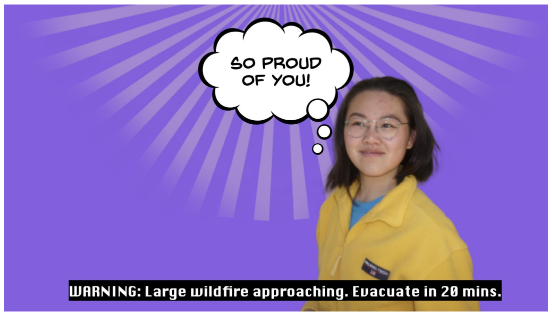
In terms of our team's navigation strategy to reach the victims, it consists primarily of using the provided starter code for frontier exploration, and then cascades down a list to try other strategies that we implemented in contests 1 and 2 for when the frontier exploration fails. However, we do continuously monitor whether the frontier exploration is able to restart, and exit the backup strategies to do so if able. This is discussed further in Section 5.1.1 and is shown as a strategy flowchart in Figure 6.

## 3.1 Detected Emotion - Anger


...SORRY...
WARNING: Large wildfire approaching. Please evacuate in 20 mins.

If the robot detects that the victim is angry, the robot will react with the primary emotion response of **embarrassment (shame)**. This reaction is appropriate because the robot is unsure why the person is angry, and is worried and self-conscious that the person is angry at the robot. The TurtleBot subsequently feels embarrassed that it might've done something wrong and is now getting yelled at. The image the robot displays is shown above, and has a pale pink coloured background, which is meant to diffuse the anger of the victim. The anger of the person must be diffused for the evacuation message to reach the victim, it has been shown that strong emotions such as anger, disgust or resentment as a reaction to someone's anger is not an effective way to diffuse their anger, hence the use of a softer emotion like embarrassment. The audio for this emotion can be heard here, and features a dejected voice that displays shame for having done something wrong, but still kindly asks the victim to evacuate in the next 20 minutes.

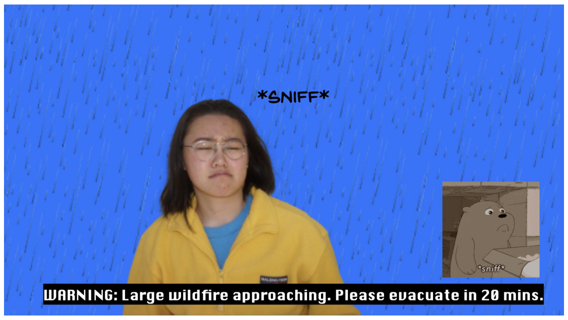
## 3.2 Detected Emotion - Neutral



If the robot detects that the victim is neutral, the robot will react with the secondary emotion response of **pride**. This reaction is appropriate because the robot is proud that the victim is remaining calm in such a time of adversity and change.

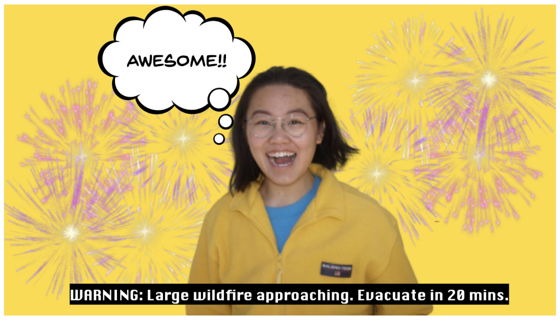
The image the robot displays is shown to the left, and has a purple coloured background, which is a historically symbolic colour of pride and loyalty, indicating the robot's loyalty to the victim's safety. The neutrality and calmness of the victim must be maintained in order for a successful evacuation, so the robot will not show any exceedingly strong emotions such as positive excitement, sadness or resentment. The audio for this emotion can be heard here, and features a cheerful voice that is beaming with pride, stating with positive affirmation how proud they are of the victim for remaining calm.

## 3.3 Detected Emotion - Fear


*SNIFF*
*sniff*
WARNING: Large wildfire approaching. Please evacuate in 20 mins.

If the robot detects that the victim is fearful, the robot will react with the primary emotion response of **sadness**. This reaction is appropriate because the robot is upset that the victim is sad, and wants to try to show the victim that they understand their fear, and that it is valic, but that they must evacuate in the next 20 minutes. Any other emotion such as pride, resentment or disgust is not appropriate in this situation. The audio for this emotion can be heard here, and features a sad voice that sniffs and empathizes with the victim.

## 3.4 Detected Emotion - Sad



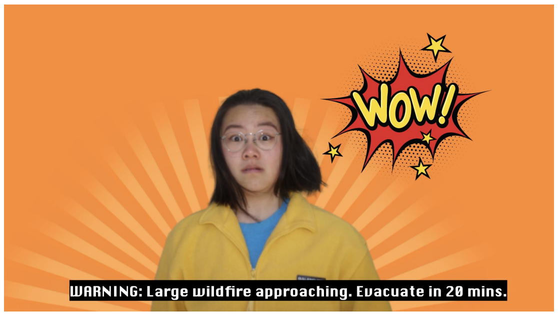
If the robot detects that the victim is sad, the robot will react with the secondary emotion response of **positively excited.** This reaction is a reassurance to the victim that things will be okay. In this case, it is not useful for the robot to empathize or be sad, since the victim will remain sad and might not want to evacuate. The robot must show the victim that life is worth living and to evacuate. The audio for this emotion can be heard here, and features a cheerful and excited voice and music that brings joy to the listener.

## 3.5 Detected Emotion - Happiness

...REALLY??
WARNING: Large wildfire approaching. Evacuate in 20 mins.

If the robot detects that the victim is happy, the robot will react with the secondary emotion response of **resentment**. This reaction is appropriate since the robot is resentful of the fact that the victim is happy during a time of crisis. The robot's response is not a strong primary emotion such as anger or disgust since this could trigger the same emotion in the victim. Rather the robot realizes now is not the time to be happy, as stated in the audio, which can be heard here.

## 3.6 Detected Emotion - Surprise



If the robot detects that the victim is surprised, the robot will react with the primary emotion response of **surprise.** This reaction is appropriate since the surprised person shocks the robot, and they are surprised to see the person. The robot's response is an effort to sympathize with the victim, so that they are more likely to comply with the evacuation instructions the robot gives, and can be heard here.

## 3.7 Detected Emotion - Disgust

...UGH... I KNOW...
WARNING: Large wildfire approaching. Evacuate in 20 mins.

If the robot detects that the victim is disgusted, the robot will react with the primary emotion response of **disgust**. This reaction is appropriate since the robot wants to empathize with the victim, and let them know that they too are disgusted with the current state of evacuations and the emergency itself. This empathy means the victim is more likely to comply with the evacuation instructions the robot gives, which can also be heard here.

## 3.8 Emotion Classification Model

The image classification model was done through a Convolutional Neural Network (CNN). The team was provided with a baseline model which used a 70:30 training and validation split with a dataset of faces. This network has the following structure: 5 convolutional layers and 3 fully connected layers, along with batch normalization, a dropout value of 0.25, and the ReLU activation function. Additionally, it employs the Adam optimizer to adjust the weights, and CrossEntropyLoss to calculate the model's loss.

The strategy to obtain the final classification model can be divided into two sections: architecture and training. For training, the first modification made was the implementation of cross-validation in place of the regular training-validation split. This allowed us to validate our model across the entire dataset by splitting the data into K number of folds (which we chose as K=5), and training the model K times, saving one of the folds for validation and using the remaining ones for training in each iteration. In order to implement this method, the team imported the KFold method from sklearn [3]. Then, the cross-validation accuracy is taken to be the average of all the iterations' validation accuracies. Using this method, the baseline model achieved a cross-validation accuracy of 61.03% after 60 epochs.

In order to improve this, the team made the following modifications to the model's architecture. First, although we kept the same number of layers, the order of each block was altered slightly, moving the batch normalization before the activation. This was based on the original batch normalization paper, which recommended this configuration [4]. Next, the number of output channels was increased from 128 to 256 in the last convolutional layer. Additionally, in the fully connected layer, the values were replaced with numbers proportional to the input features (1152), in order to have a more symmetric progression. Finally, the ReLU activation was replaced by LeakyReLU, which was done to fix some of the "vanishing gradient" problems [5].

Regarding the training optimization, the team decided to implement two main changes. First, the optimization method was changed from Adam to AdamW [6]. AdamW is based on Adam, but it uses weight decay to increase the generalization of the model. In essence, it penalizes large weights in order to reduce the level of codependence between neurons. The second modification was applying a step learning rate decay to multiply the learning rate by a factor of 0.1 after 30 epochs, in order to avoid plateaus in the loss decrease.

Finally, the model was trained for 60 epochs, achieving a cross-validation accuracy of 62.43%.

## 4.0 Sensor Design

The simulated TurtleBot comes equipped with multiple sensors, including several of which are built into the Kobuki base. A layout of where these sensors are located is shown in Figure 5. For this contest, the bumpers, odometry, depth sensor (part of the RGBD sensor), and IMU are used. Each sensor and how it was used to meet the contest requirements are described in more detail in the following subsections.

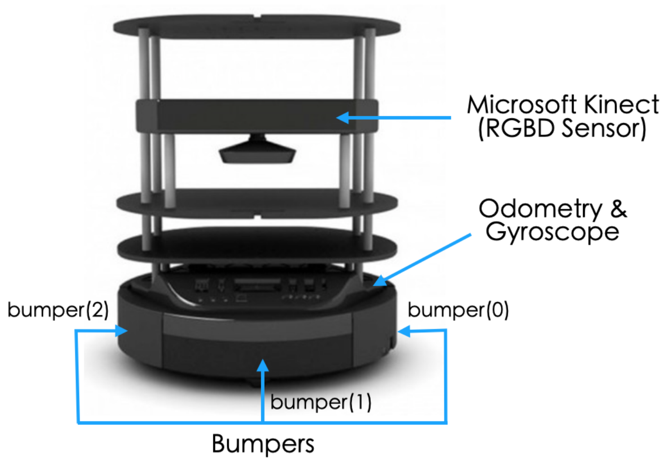


*Figure 5: Placement of sensors on the simulated TurtleBot (Front view).*

### 4.1 Microsoft Kinect RGBD Sensor

The Microsoft Kinect RGBD sensor is composed of two sensors, the RGB sensor and the depth sensor. For this contest, the depth sensor gets utilized to aid with navigation, and the RGB sensor helps to obtain input images which get passed to the CNN classifier. The depth sensor projects a speckle pattern of IR beams onto objects in the environment, which are then reflected back and get detected by the sensor. The distortion of the original pattern from this reflected pattern determines the depth information. This results in a 3D point cloud used to generate the map in RViz, and is an input to the provided frontier exploration starter code, which we rely upon extensively. We found that setting the maximum range of the laser depth range to 1.8 struck a good balance between being able to explore the environment effectively without missing victims.

Additionally, during each iteration of our program's main loop, the laser sensor returns a one-dimensional array of laser distance measurements in a field of view 22.5° to the left and right of the TurtleBot's dead centre heading. We use this array in two ways, but mainly as one of the last backup strategies (as discussed in Section 5.1) for when the provided frontier exploration fails:

1. We define `minLaserDist` as the minimum laser distance within the array, and use it as a metric to determine what the laser distance is at the current heading.
2. We define `LSLaserSum` and `RSLaserSum` as the sum of the last `n/2` and first `n/2` values respectively in the array, where `n` is the size of the array. In essence, `LSLaserSum` and

`RSLaserSum` represent which side of the TurtleBot is likely to have a clearer path. This is used primarily to determine the direction of movement as part of our low-level controller helper functions such as **`chooseAngular`**.

## 4.2 Odometry

The odometry sensor helps to estimate the TurtleBot's position and orientation relative to its starting location by using the wheel encoders of the Kobuki base. From this sensor, we use `posX`, `posY`, and `yaw` to represent the horizontal position, vertical position, and angular orientation of the TurtleBot relative to its starting position.

We use these variables within the implementation of several of our controllers for when the frontier exploration fails, such as to rotate a desired angle or move linearly for a desired distance. In addition, the frontier exploration code itself uses odometry as an input for making navigation commands using the `move_base` ROS node. In the contest, we use a maximum linear speed of 0.7 m/s rather than the 0.5 m/s set in the starter code (in `dwa_local_plannel_params.yaml`) since the Kobuki base can support up to a maximum of 0.7 m/s. However, we left the maximum angular velocity unchanged from the provided starter code (5 rad/s), despite this being significantly higher than the absolute maximum and recommended maximum angular velocities to prevent degradation of the gyroscope sensor of the Kobuki base (180 °/s and 110 °/s respectively) [7].

## 4.3 Bumpers

There are three bumpers on the TurtleBot's Kobuki base located on the left, center and right sides of it, as shown in Figure 2. The bumpers are used when there are obstacles that the RGB sensor cannot see, or in an event where reading the laser scan inputs was not sufficient to help the TurtleBot from hitting an obstacle.. The bumpers are initially set as released, with a state of 0, while a pressed bumper returns a state of 1.

Whenever a bumper is pressed, the program executes **`bumperPressedAction`**, which is a handling function we developed for Contest 1, and is discussed further in that report.

## 4.4 Inertial Measurement Unit (IMU)

The IMU of the Kobuki base measures and reports the TurtleBot's specific force, angular rate, and orientation using a combination of the accelerometer and gyroscope. The IMU sensor was not used by our team for this contest. However the IMU sensors can be used to verify the velocity since they have their own inertial measurement unit. This was used in our work for Contest 1.

# 5.0 Controller Design

Section 5.1 discusses our high level control architecture and Section 5.2 discusses the pertinent parts of the lower level control architecture that supports the operation of the high level control.

## 5.1 High-Level Control Architecture

As discussed in Section 3.0, our control strategy is behaviour based since that form of control lends itself well to meeting the primary contest requirement of interacting with emotional victims appropriately. In this section, we discuss the navigation strategy that acts as the backbone of being able to implement this behaviour based control by navigating autonomously to approach the victims. The navigation strategy is shown as a flowchart in Figure 6, and can be summarized as trying to use the provided frontier exploration strategy, but implementing several backup strategies similar to approaches tried in Contests 2 and 1 for when the provided frontier exploration does not work as expected. The goal of these backup strategies is to create movement of the TurtleBot such that the frontier exploration strategy recovers and is able to be used again. Our tiered backup strategy approach is discussed in the below subsections.

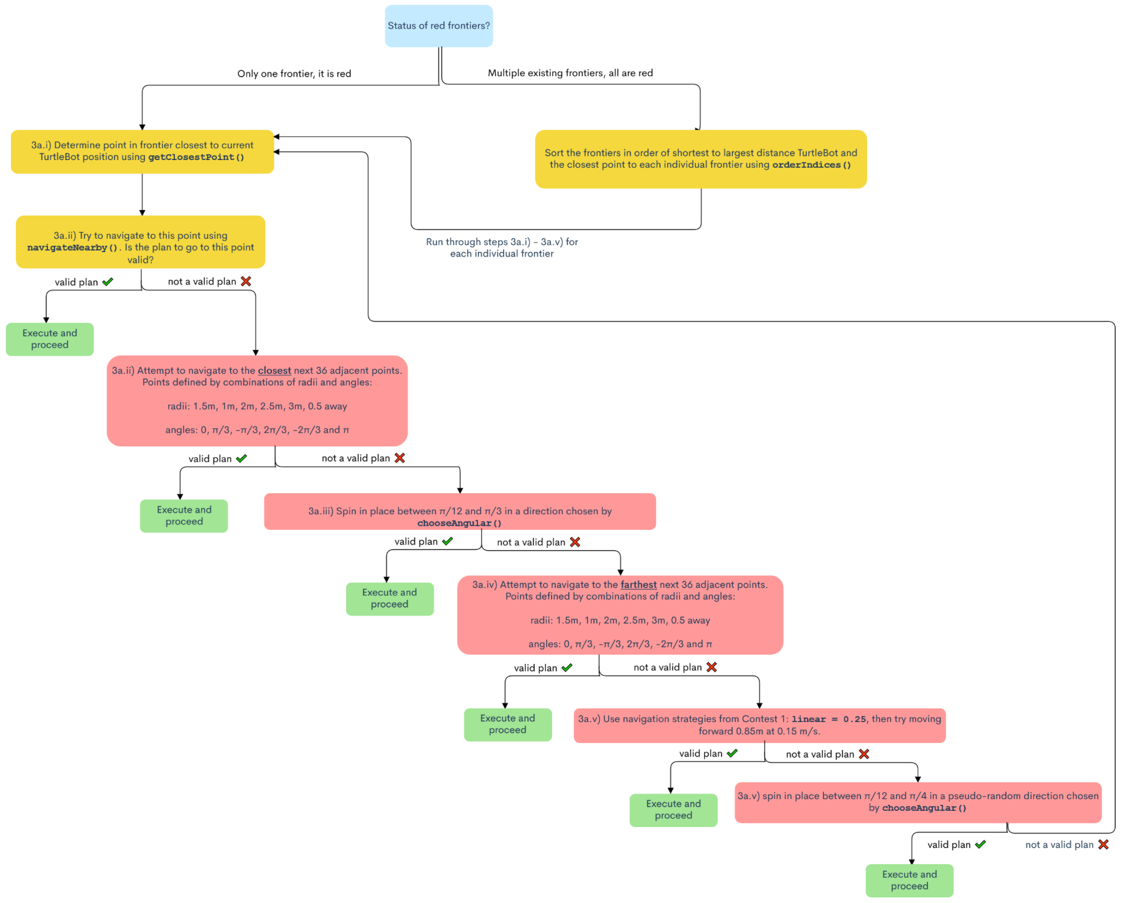

Status of red frontiers?
Only one frontier, it is red
Multiple existing frontiers, all are red
3a.i) Determine point in frontier closest to current TurtleBot position using getClosestPoint()
Sort the frontiers in order of shortest to largest distance TurtleBot and the closest point to each individual frontier using orderIndices()
Run through steps 3a.i) - 3a.v) for each individual frontier
3a.ii) Try to navigate to this point using navigateNearby(). Is the plan to go to this point valid?
valid plan ✓
not a valid plan ✗
Execute and proceed
3a.ii) Attempt to navigate to the closest next 36 adjacent points. Points defined by combinations of radii and angles:
radii: 1.5m, 1m, 2m, 2.5m, 3m, 0.5 away
angles: 0, π/3, -π/3, 2π/3, -2π/3 and π
valid plan ✓
not a valid plan ✗
Execute and proceed
3a.iii) Spin in place between π/12 and π/3 in a direction chosen by chooseAngular()
valid plan ✓
not a valid plan ✗
Execute and proceed
3a.iv) Attempt to navigate to the farthest next 36 adjacent points. Points defined by combinations of radii and angles:
radii: 1.5m, 1m, 2m, 2.5m, 3m, 0.5 away
angles: 0, π/3, -π/3, 2π/3, -2π/3 and π
valid plan ✓
not a valid plan ✗
Execute and proceed
3a.v) Use navigation strategies from Contest 1: linear = 0.25, then try moving forward 0.85m at 0.15 m/s.
valid plan ✓
not a valid plan ✗
Execute and proceed
3a.v) spin in place between π/12 and π/4 in a pseudo-random direction chosen by chooseAngular()
valid plan ✓
not a valid plan ✗
Execute and proceed

*Figure 6: Flowchart of the actions taken to improve the frontier exploration capabilities of the original exploration code. The image is a vector, please zoom in if the text is small.*

### 5.1.1 Frontier Exploration

Frontier exploration is a relatively new navigation strategy for autonomous robots where we maintain a set of points called frontiers which are a boundary between currently known space and unknown space in the environment. For a set of points to be classified as a frontier, it must neighbour at least one region of known open space that does not contain obstacles . The algorithm works by running a breadth-first search to discover frontiers within the entire environment grid at each update step, and then another breadth-first search is run on those points to obtain their centroid locations for attempted navigation [8]. The frontiers are then sorted in increasing order of their cost, which is a function that takes into account the distance to and size of each frontier, and then visited in this order by sending commands through `move_base`. In `contest3.launch`, we weigh minimum distance at 0.003 and frontier size at 1 in calculating each frontier's cost. After arriving at each frontier centroid, the TurtleBot spins 360° to get a better view of the environment [8].

For the most part, this algorithm as provided works fairly well, and is the base of our navigation efforts for this contest. However, there are known issues where it sometimes times out and blacklists frontiers which should be explorable. Our resolutions to this issue are discussed in the subsections that follow.

### 5.1.2 Sending `move_base` commands (inspired by Contest 2)

While the TurtleBot moves, we subscribe to the frontiers published by the frontier exploration algorithm. In the callback function, **`markerCallback`**, we maintain a global vector of vectors called `redFrontiers` that stores the points within each individual frontier so long as that frontier has been blacklisted (i.e. it has been coloured red). We also maintain a global boolean, `allFrontiersRed`, which is true if every single frontier that currently exists is red, and false otherwise. We consider the frontier exploration strategy failed and implement the backup strategies discussed below and in Section 5.1.3 if `allFrontiersRed` returns true. As soon as `allFrontiersRed` returns false, we exit these backup strategies and immediately revert back to using frontier exploration.

If `allFrontiersRed` returns true, we first check whether we have only one frontier and it is red, or if we have multiple frontiers which are all red. Regardless of case, the strategy is essentially the same since the latter just repeats the former in a particular order for all frontiers.

#### 5.1.2.1 One frontier exists and is red

If one frontier exists and is red, we use the helper function **`getClosestPoint`** to try and navigate to the closest point in the frontier from the current TurtleBot position. The function used to do this navigation, **`navigateNearby`**, is described in Section 5.2.1. Essentially, this function uses `move_base/NavfnROS/make_plan` to check whether a valid path exists to a point, and then uses `move_base` to navigate there if the path is valid. If the path is invalid, it also attempts navigation to adjacent points at predefined radii and angles away. This is similar to our navigation approach from Contest 2.

Although completing the above resolves the issue and at least one frontier becomes non-red after doing so, there are some cases where it doesn't work; either no valid path could be found, or a path that was considered valid failed during its execution. If this happens, we repeat the procedure described in the previous paragraph, but instead try navigating to the farthest point in the given frontier, which is determined using the helper function **`getFurthestPoint`**.

#### 5.1.2.2 Multiple frontiers exist and all are red

In this case, we use helper function **`orderIndices`** to sort the frontiers in order of increasing distances between the TurtleBot and the closest point on an individual frontier. We then repeat the procedure described in Section 5.1.2.1 for each frontier in this sorted order.

### 5.1.3 Forced movement and random rotations (inspired by Contest 1)

In the rare event that the approaches described in Section 5.1.2 have not worked, and all the frontiers are still red, then we resort to navigation methods from Contest 1. We set `linear=0.25` and directly publish this velocity. We then try moving forward 0.85m at 0.15m/s using the **`moveThruDistance`** function. Finally, we rotate in place a random amount between 15° and 45° in a pseudo-random direction. The rotation is done by **`rotateThruAngle`**, and the pseudo-random rotation direction is implemented using **`chooseAngular`**.

Since this strategy essentially has the TurtleBot navigating blind, our bumper handling implementation from Contest 1, which can be seen in **`bumperPressedAction`**, has been implemented. Functions **`moveThruDistance`**, **`rotateThruAngle`**, **`chooseAngular`**, and **`bumperPressedAction`** were all developed for Contest 1, and detailed documentation for how these functions work can be found in our team's Contest 1 report

### 5.1.4 TurtleBot emotion output using **`robotReaction`**

Once the TurtleBot assesses the emotion that the victim is displaying from `emotionDetected` it will display a unique and appropriate emotional response as stipulated in the contest constraints in Section 2.2. The function **`robotReaction`** directs the TurtleBot on the actions to take, which include displaying an appropriate image and playing a corresponding sound file. The function first takes the emotion the human is displaying, ie. fear, and finds it in the string `humanEmotions`, which in this case would be an index of 2. The robot will then display the emotion that corresponds to that index, in this case 2, in `robotEmotions`, which in this case would cause the robot to display an emotion of sadness. This is shown in the order the emotions are laid out in the two strings, below:

```
    string humanEmotions[7] = {"angry", "disgust", "fear", "happy", "sad",
"surprise", "neutral"};
    string robotEmotions[7] = {"embarrasment", "disgust", "sadness", "resentment",
"positive excitement", "surprise", "pride"};
```

The system then shows the robot's emotion's corresponding image using the built-in `imread` function through the command `imread(imagePaths[emotionDetected], CV_LOAD_IMAGE_UNCHANGED)`. The parameter `CV_LOAD_IMAGE_UNCHANGED` indicates for the system to show the image as is (other options include B&W, RGB etc.). Sound is then played from the command `sc.playWave(path_to_sounds + soundFiles[emotionDetected])`, which fetches the corresponding wav file from the string `soundFiles`, which have the sounds listed in the same order as `robotEmotions`. All sounds are roughly 15 seconds in length, and the image window is destroyed after the sound finishes playing.

## 5.2 Low-Level Control Architecture

This section describes pertinent supporting functions used to help enable the operation of the high-level navigation control architecture described in Section 5.1.

### 5.2.1 Description of **`navigateNearby`**

**`navigateNearby`** is one of the key functions in our program that helps to navigate to individual points, and typically resolves the frontier exploration issues after having been called.

The function takes in the desired x,y navigation point, `startPoint`; a vector of radii, `radii`; a vector of angles, `angles`; the node handle `n`; and the current x,y, angle position of the TurtleBot, `robotPose`.

To begin, and in order to minimize navigation time, we calculate a desired orientation angle relative to the current TurtleBot's orientation such that if navigating in a straight line, the TurtleBot will not need to rotate in place before the navigation command fully finishes. However, we note that if navigating around a wall, that setting the orientation in such a way is not always guaranteed to be optimal. More formally, if the TurtleBot's starting position is $(x_1, y_1)$ and the desired navigation point is $(x_2, y_2)$, then the desired orientation angle is $arctan(\frac{y_2 - y_1}{x_2 - x_1})$.

Using the boolean **`checkPlan`** function we developed for Contest 2, we check if it is possible to navigate to `startPoint`, and do so if the plan is valid using `move_base`

If the plan was not valid, we iterate through all combinations of elements in `radii` and `angles` to navigate to `startPoint` at an offset of `radii[i]` and angle of `angles[j]` away. The function gets called with `radii = {1.5, 2.0, 1.0, 2.5, 3.0, 0.5}` and `angles = {0.0, M_PI/3, -M_PI/3, 2*M_PI/3, -2*M_PI/3, M_PI-0.01}`, and tries all angles for a given radius before moving on to the next radius. The radii have intentionally not been sorted in order to achieve a balance between trying at near and far radii from the desired point. As soon as a valid plan is found, navigation is executed using `move_base`.

### 5.2.2 Emotion classification

The process of obtaining the emotion is done through the **`emotionCallback`** function. This function subscribes to the **`EmotionDetector`** class. Once the robot stops in front of a victim, this function runs it through the best CNN model and obtains the index corresponding to the emotion detected, which gets saved into global integer variable `emotionDetected`. Afterwards, the **`robotReaction`** function directs the robot to react appropriately depending on the emotion detected.

## 6.0 Future Recommendations

The main recommendations are for improving the accuracy of the emotion detector, which currently has an accuracy of around 63%. The contest constraints stipulated that teams must only use the training dataset provided, and not use any pre-trained models. This is adequate for a contest where teams should be on even footing, but to ensure optimal performance, using pre-trained models would help achieve a higher accuracy. Namely, more datasets of emotions could also be included, in particular those with induced emotions. Most emotion datasets are staged, in that the participant is asked to act “disgusted” or act “sad”. On the other hand, in an induced dataset, participants are filmed while they are shown images/videos that induce the ration, leading to far more natural expressions, which translate far better into the real world [9].

If a suitable dataset cannot be found, more image transformations should be applied so that the CNN learns to generalize; examples of image transformations are shown in Figure 7, and [10] explains how transformations help a CNN generalize.

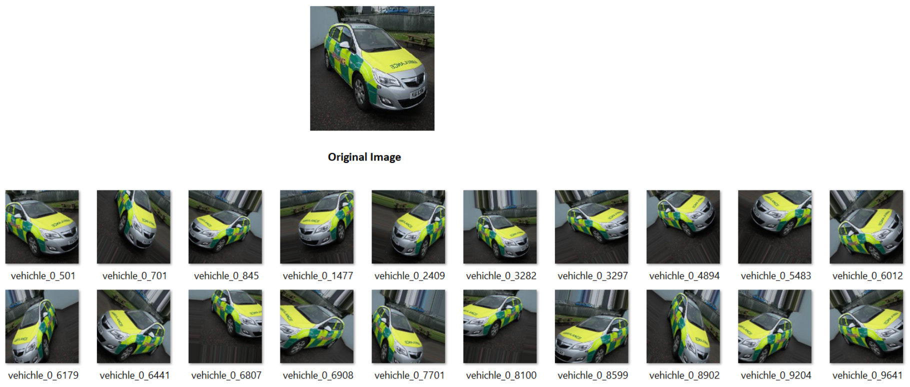


*Figure 7: Examples of image transformations to increase the ability of the CNN to generalize*

Another method of improving the accuracy of the model would be to perform a structured hyperparameter search in terms of the learning rate, momentum, regularization in a structured way to determine which combinations provide the highest accuracy.

## References

[1] W. Marston, "Primary emotions.", Psychological Review, vol. 34, no. 5, pp. 336-363, 1927.

[2] Riess, H, “The science of empathy.”, Journal of patient experience, vol. 4, no. 2, pp. 74-77, 2017.

[3] “sklearn.model_selection.KFold,” scikit. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html. [Accessed: 17-Apr-2021].

[4] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” arXiv.org, 02-Mar-2015. [Online]. Available: https://arxiv.org/abs/1502.03167. [Accessed: 17-Apr-2021].

[5] CS231n Convolutional Neural Networks for Visual Recognition. [Online]. Available: https://cs231n.github.io/neural-networks-1/. [Accessed: 17-Apr-2021].

[6] “AdamW and Super-convergence is now the fastest way to train neural nets,” AdamW and Super-convergence is now the fastest way to train neural nets ·. [Online]. Available: https://www.fast.ai/2018/07/02/adam-weight-decay/. [Accessed: 17-Apr-2021].

[7] "Kobuki - ROS Components", *Kobuki*, 2021. [Online]. Available: https://www.roscomponents.com/en/mobile-robots/97-kobuki.html. [Accessed: 15- Apr- 2021]

[8] A. Topiwala, P. Inani J. Bender, and A. Kathpal, "Frontier Based Exploration for Autonomous Robot", University of Maryland Robotics Center , 2018.

[9] Haamer, R. E., Rusadze, E., Lsi, I., Ahmed, T., Escalera, S., & Anbarjafari, G. “Review on emotion recognition databases”. Hum. Robot Interact. Theor. Appl, 3, 39-63, 2017.

[10] Azulay, A., & Weiss, Y. “Why do deep convolutional networks generalize so poorly to small image transformations?”. arXiv preprint arXiv:1805.12177, 2018.

# Appendices

## Appendix A - Complete C++ Code - contest3.cpp

A copy of the code can be found here.

```
#include <contest3.h>

float dist(float x1, float y1, float x2, float y2){
    //Calculates the Euclidean distance between two points (x1, y1), (x2, y2)
    return sqrt(pow(x2-x1, 2) + pow(y2-y1, 2));
}

bool allFrontiersRed = false; // return true if ALL frontiers are red
std::vector<std::vector<geometry_msgs::Point>> redFrontiers;
void markerCallback(const visualization_msgs::MarkerArray::ConstPtr& msg){
    // Checks if all frontiers are red
    // Also updates a vector of vectors corresponding to current red frontiers
    visualization_msgs::Marker m;
    int numMarkers;
    numMarkers = msg->markers.size();

    // Reset redFrontiers vector
    if(!redFrontiers.empty()){
        redFrontiers.clear();
    }

    allFrontiersRed = true;
    std::cout << numMarkers/2 << " frontiers found \n";
    for(int i = 0; i < numMarkers; i+=2){
        m = msg->markers[i];
        if (m.color.r < 1.0){
            allFrontiersRed = false;
        }
        else{
            // Update redFrontiers with points
            redFrontiers.push_back(m.points);
        }
        //std::cout << m.points.size() << " points in frontier " << i/2 << ". Frontier Red:
" << (m.color.r == 1.0) << "\n";
    }
}

geometry_msgs::Point getCentroid(std::vector<geometry_msgs::Point> points){
    // Calculates the centroid point (average in each dimension) given an array of points
    geometry_msgs::Point centroidPoint;
    float xAvg = 0.0;
    float yAvg = 0.0;

    if (points.size() > 0){
        for(int i = 0; i < points.size(); i++){
            xAvg += points[i].x;
            yAvg += points[i].y;
        }
        xAvg = xAvg / points.size();
        yAvg = yAvg / points.size();
    }
    centroidPoint.x = xAvg;
    centroidPoint.y = yAvg;
    centroidPoint.z = 0.0;
    return centroidPoint;
}

geometry_msgs::Point getClosestPoint(std::vector<geometry_msgs::Point> points, RobotPose
robotPose){
    // Takes in a vector of points corresponding to a frontier and current robotPose
    // Returns the closest point within the frontier
    geometry_msgs::Point closestPoint;
    float distance, minD = std::numeric_limits<float>::infinity();

    closestPoint.x = 0.0;
    closestPoint.y = 0.0;
    closestPoint.z = 0.0;

    if (points.size() > 0){
        for(int i = 0; i < points.size(); i++){
            distance = dist(robotPose.x, robotPose.y, points[i].x, points[i].y);
            if(distance < minD){
                minD = distance;
                closestPoint = points[i];
            }
        }
    }
    return closestPoint;
}

geometry_msgs::Point getFurthestPoint(std::vector<geometry_msgs::Point> points, RobotPose
robotPose){
    // Takes in a vector of points corresponding to a frontier and current robotPose
    // Returns the furthest point within the frontier
    geometry_msgs::Point furthestPoint;
    float distance, maxD = 0;

    furthestPoint.x = 0.0;
    furthestPoint.y = 0.0;
    furthestPoint.z = 0.0;

    if (points.size() > 0){
        for(int i = 0; i < points.size(); i++){
            distance = dist(robotPose.x, robotPose.y, points[i].x, points[i].y);
            if(distance > maxD){
                maxD = distance;
                furthestPoint = points[i];
            }
        }
    }
    return furthestPoint;
}

std::vector<int> orderIndices(std::vector<std::vector<geometry_msgs::Point>> frontiers,
RobotPose robotPose){
    // Takes in a vector of red frontier vectors and current robotPose
    // Returns a vector of indices corresponding to the frontiers sorted from closest to
furthest
    geometry_msgs::Point point;
    float d;
    std::vector<float> distances;
    std::vector<int> indices;
    std::vector<std::pair<float, int>> distances_and_indices;

    for(int i = 0; i < frontiers.size(); i++){
        point = getClosestPoint(frontiers[i], robotPose);
        d = dist(robotPose.x, robotPose.y, point.x, point.y);
        distances_and_indices.push_back(std::make_pair(d, i));
    }

    std::sort(distances_and_indices.begin(), distances_and_indices.end());

    for(int i = 0; i < distances_and_indices.size(); i++){
        indices.push_back(distances_and_indices[i].second);
    }

    return indices;
}

uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED};
const uint8_t LEFT = 0, CENTER = 1, RIGHT = 2;
void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg){
    bumper[msg->bumper] = msg->state;
}

float posX = 0.0, posY  = 0.0, yaw = 0.0, angular = 0.0, linear = 0.0;
void odomCallback(const nav_msgs::Odometry::ConstPtr& msg){
    posX = msg->pose.pose.position.x;
    posY = msg->pose.pose.position.y;
    yaw = tf::getYaw(msg->pose.pose.orientation);
    //ROS_INFO("Odom: (%.2f, %.2f, %.2f)", posX, posY, yaw);
}

float minLaserDist = std::numeric_limits<float>::infinity(), minLSLaserDist =
std::numeric_limits<float>::infinity(), minRSLaserDist =
std::numeric_limits<float>::infinity();
float LSLaserSum = 0, RSLaserSum = 0;
int32_t nLasers=0, desiredNLasers=0, desiredAngle=22.5;
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
        // Calculates minLaserDist overall, per side as minLSLaserDist and minRSLaserDist,
    // and the sum of laser measurements on each side LSLaserSum and RSLaserSum

    minLaserDist = std::numeric_limits<float>::infinity();
    minLSLaserDist = std::numeric_limits<float>::infinity();
    minRSLaserDist = std::numeric_limits<float>::infinity();
    LSLaserSum = 0, RSLaserSum = 0;
    float maxLaserThreshold = 7;
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment;

    int start = 0, end = nLasers;
    if (DEG2RAD(desiredAngle) < msg->angle_max && DEG2RAD(-desiredAngle) > msg->angle_min){
        start = nLasers / 2 - desiredNLasers;
        end = nLasers / 2 + desiredNLasers;
    }
    for (uint32_t laser_idx = start; laser_idx < end; ++laser_idx){
        minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
        if (laser_idx <= nLasers / 2){
            minRSLaserDist = std::min(minRSLaserDist, msg->ranges[laser_idx]);
            if (msg->ranges[laser_idx] < maxLaserThreshold){
                RSLaserSum += msg->ranges[laser_idx];
            }
        }
        else{
            minLSLaserDist = std::min(minLSLaserDist, msg->ranges[laser_idx]);
            if (msg->ranges[laser_idx] < maxLaserThreshold){
                LSLaserSum += msg->ranges[laser_idx];
            }
        }
    }
    //ROS_INFO("Min Laser Distance: %f \n Left: %f \n Right: %f \n LSum: %f \n RSum: %f",
minLaserDist, minLSLaserDist, minRSLaserDist, LSLaserSum, RSLaserSum);
}

int emotionDetected = -1; //Use -1 to indicate exploring is occuring, 0-6 for emotions
void emotionCallback(const std_msgs::Int32::ConstPtr& msg){
    emotionDetected = msg->data;
    std::cout << "Emotion detected in its callback: " << emotionDetected << " \n";
}

void update(geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub, uint64_t* pSecondsElapsed,
            const std::chrono::time_point<std::chrono::system_clock> start){
    // Sets linear and angular velocities, updates main loop timer
    (*pVel).angular.z = angular;
    (*pVel).linear.x = linear;
    (*pVel_pub).publish(*pVel);
    ros::spinOnce();

    // Update the timer.
    *pSecondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start).cou
nt();
    ros::Duration(0.01).sleep();
}

template <class T>
T randBetween(T a, T b){
    // Returns a random number between a and b inclusive. Assumes a<b.
    if (std::is_same<T, int>::value){
        float x = float(rand())/float(RAND_MAX);
        return int(round(float(b-a)*x + float(a)));
    }
    else{
        T x = T(float(rand())/float(RAND_MAX));
        return (b-a)*x + a;
    }
}

bool anyBumperPressed(){
    // Returns true if any bumper is pressed, false otherwise
    bool any_bumper_pressed = false;
    for (uint32_t b_idx = 0; b_idx < N_BUMPER; ++b_idx) {
        any_bumper_pressed |= (bumper[b_idx] == kobuki_msgs::BumperEvent::PRESSED);
    }
    return any_bumper_pressed;
}

void moveThruDistance(float desired_dist, float move_speed, float startX, float startY,
geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub,
                    uint64_t* pSecondsElapsed, const
std::chrono::time_point<std::chrono::system_clock> start){
    // Moves turtlebot desired_dist at move_speed. Negative desired_dist moves backwards.
Only magnitude of move_speed is used.
    int i = 0;
    float current_dist = dist(startX, startY, posX, posY);
    while (current_dist < fabs(desired_dist) && i < 100 && *pSecondsElapsed < 1200){
        ros::spinOnce();
        angular = 0;
        linear = copysign(fabs(move_speed), desired_dist); //move move_speed m/s in
direction of desired_dist
        update(pVel, pVel_pub, pSecondsElapsed, start); // publish linear and angular
        current_dist = dist(startX, startY, posX, posY);

        if (anyBumperPressed()){
            bumperPressedAction(pVel, pVel_pub,pSecondsElapsed, start);
            break;
        }
        i+=1;
    }
}

void rotateThruAngle(float angleRAD, float angleSpeed, float yawStart, float set_linear,
geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub,
                     uint64_t* pSecondsElapsed, const
std::chrono::time_point<std::chrono::system_clock> start){
    // Rotates turtlebot angleRAD rad CW(-) or CCW(+) depending on angleRAD's sign at
angleSpeed rad/s.
    // Make sure angleRAD is between +/- pi
    // Use set_linear = 0 to rotate in place
    int i = 0;
    while (fabs(yaw - yawStart) <= fabs(angleRAD) && i < 500 && *pSecondsElapsed < 1200){
        ros::spinOnce();
        angular = copysign(angleSpeed, angleRAD); //turn angleSpeed rad/s in direction of
angleRAD
        linear = set_linear;
        update(pVel, pVel_pub, pSecondsElapsed, start); // publish linear and angular

        if (anyBumperPressed()){
            ROS_INFO("Breaking out of rotate due to bumper press \n Left: %d \n Center: %d
\n Right: %d \n", bumper[LEFT], bumper[CENTER], bumper[RIGHT]);
            bumperPressedAction(pVel, pVel_pub,pSecondsElapsed, start);
            break;
        }
        i += 1;
    }
}

float chooseAngular(float laserSideSumThreshold, float probSpinToLarger){
    // Chooses the angular velocity and direction based on clearer laser side and
probability inputted
    // Can also call this in second argument of copysign() to only extract a direction (e.g.
for a rotation)
    float prob = randBetween(0.0, 1.0), angular_vel = M_PI/8, maxLaserThreshold = 7;
    ros::spinOnce();
    if(fabs(fabs(LSLaserSum) - fabs(RSLaserSum)) > laserSideSumThreshold){
        //If one side's laser distance > other side by more than laserSideSumThreshold, go
to that side
        if(fabs(LSLaserSum) - fabs(RSLaserSum) > laserSideSumThreshold){
            ROS_INFO("LS >> RS. Spin CCW");
            angular_vel = randBetween(M_PI/16, M_PI/8); //improves gmapping resolution
compared to always using constant value
        }
        else{
            ROS_INFO("RS >> LS. Spin CW");
            angular_vel = -randBetween(M_PI/16, M_PI/8);
        }
    }
    else{
        // Laser distances approx. equal. Go to larger at probSpinToLarger probability
        ROS_INFO("LS ~ RS. Spinning to larger at %.2f chance", probSpinToLarger);
        if ((fabs(LSLaserSum) > fabs(RSLaserSum) || fabs(minLSLaserDist) >
fabs(minRSLaserDist)) && prob < probSpinToLarger){
            angular_vel = randBetween(M_PI/16, M_PI/8);
        }
        else{
            angular_vel = -randBetween(M_PI/16, M_PI/8);
        }
    }
    return angular_vel;
}

void bumperPressedAction(geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub, uint64_t*
pSecondsElapsed,
                         const std::chrono::time_point<std::chrono::system_clock> start){
    // Actions to implement if any bumpers get pressed
    bool any_bumper_pressed = true;
    any_bumper_pressed = anyBumperPressed();

    if (any_bumper_pressed && *pSecondsElapsed < 900){
        ROS_INFO("Bumper pressed \n Left: %d \n Center: %d \n Right: %d \n", bumper[LEFT],
bumper[CENTER], bumper[RIGHT]);

        if (bumper[LEFT]){
            ROS_INFO("Left hit. Move back and spin 90 CW");
            moveThruDistance(-1.2, 0.2, posX, posY, pVel, pVel_pub, pSecondsElapsed, start);
            rotateThruAngle(DEG2RAD(-90), M_PI/4, yaw, 0, pVel, pVel_pub, pSecondsElapsed,
start);
        }
        else if (bumper[RIGHT]){
            ROS_INFO("Right hit. Move back and spin 90 CCW");
            moveThruDistance(-1.2, 0.2, posX, posY, pVel, pVel_pub, pSecondsElapsed, start);
            rotateThruAngle(DEG2RAD(90), M_PI/4, yaw, 0, pVel, pVel_pub, pSecondsElapsed,
start);
        }
        else if (bumper[CENTER]){
            ROS_INFO("Center hit. Move back and spin");
            moveThruDistance(-1.2, 0.2, posX, posY, pVel, pVel_pub, pSecondsElapsed, start);
            rotateThruAngle(copysign(2*M_PI/3, chooseAngular(10, 0.9)), M_PI/4, yaw, 0,
pVel, pVel_pub, pSecondsElapsed, start);
        }
    }
}

bool checkPlan(ros::NodeHandle& nh, float xStart, float yStart, float phiStart, float xGoal,
float yGoal, float phiGoal){
       //Returns true if there is a valid path from (xStart, yStart, phiStart) to (xGoal,
yGoal, phiGoal)
    //Adapted from
```

```
https://answers.ros.org/question/264369/move_base-make_plan-service-is-returning-an-empty-pa
th/

    bool callExecuted, validPlan;

    //Set start position
    geometry_msgs::PoseStamped start;
    geometry_msgs::Quaternion phi1 = tf::createQuaternionMsgFromYaw(phiStart);
    start.header.seq = 0;
    start.header.stamp = ros::Time::now();
    start.header.frame_id = "map";
    start.pose.position.x = xStart;
    start.pose.position.y = yStart;
    start.pose.position.z = 0.0;
    start.pose.orientation.x = 0.0;
    start.pose.orientation.y = 0.0;
    start.pose.orientation.z = phi1.z;
    start.pose.orientation.w = phi1.w;

    //Set goal position
    geometry_msgs::PoseStamped goal;
    geometry_msgs::Quaternion phi2 = tf::createQuaternionMsgFromYaw(phiGoal);
    goal.header.seq = 0;
    goal.header.stamp = ros::Time::now();
    goal.header.frame_id = "map";
    goal.pose.position.x = xGoal;
    goal.pose.position.y = yGoal;
    goal.pose.position.z = 0.0;
    goal.pose.orientation.x = 0.0;
    goal.pose.orientation.y = 0.0;
    goal.pose.orientation.z = phi2.z;
    goal.pose.orientation.w = phi2.w;

    //Set up the service and call it
    ros::ServiceClient check_path =
nh.serviceClient<nav_msgs::GetPlan>("move_base/NavfnROS/make_plan");
    nav_msgs::GetPlan srv;
    srv.request.start = start;
    srv.request.goal = goal;
    srv.request.tolerance = 0.0;
    callExecuted = check_path.call(srv);

    if(callExecuted){
        ROS_INFO("Call to check plan sent");
    }
    else{
        ROS_INFO("Call to check plan NOT sent");
    }

    if(srv.response.plan.poses.size() > 0){
        validPlan = true;
        ROS_INFO("Successful plan of size %ld", srv.response.plan.poses.size());
    }
    else{
        validPlan = false;
        ROS_INFO("Unsuccessful plan");
    }
    return validPlan;
}

bool navigateNearby(geometry_msgs::Point startPoint, std::vector<float> radii,
std::vector<float> angles, ros::NodeHandle& n, RobotPose robotPose){
    // Navigates to startPoint, and if not successful, to adjacent points defined by radii
and angles away
    bool validPlan, navSuccess;
    float xx, yy;

    // Check if valid navigation plan to the startPoint exists
    xx = startPoint.x;
    yy = startPoint.y;
    validPlan = checkPlan(n, robotPose.x, robotPose.y, robotPose.phi, xx, yy, atan2f(yy -
robotPose.y, xx - robotPose.x));

    //Try points at different radii and angles from the centroid
    for(int rCount = 0; rCount < radii.size(); rCount ++){
        for(int aCount = 0; aCount < angles.size(); aCount ++){
            if(!validPlan && allFrontiersRed){
                //ROS_INFO("Testing at offset r=%.2f phi=%.2f", radii[rCount],
angles[aCount]);
                std::cout << "Testing at offset r=" << radii[rCount] << " phi=" <<
angles[aCount] << "\n";
                xx = startPoint.x + radii[rCount]*cosf(angles[aCount]);
                yy = startPoint.y + radii[rCount]*sinf(angles[aCount]);
                validPlan = checkPlan(n, robotPose.x, robotPose.y, robotPose.phi, xx, yy,
atan2f(yy - robotPose.y, xx - robotPose.x));
            }
            else {
                break;
            }
        }
        if(validPlan || !allFrontiersRed){
            break;
        }
    }

    if(validPlan){
        navSuccess = Navigation::moveToGoal(xx, yy, atan2f(yy - robotPose.y, xx -
robotPose.x), 10);
    }
    ros::spinOnce();
    ros::Duration(0.01).sleep();
    if(!allFrontiersRed){
        std::cout << "At least 1 frontier blue \n";
        return false;
    }
    if(!validPlan || !navSuccess){
        std::cout << "Could not navigate to point. \n";
        return false;
    }
    return true;
}

void robotReaction(sound_play::SoundClient& sc){
    // Displays image and plays sound based on emotionDetected
    using namespace cv;
    using namespace std;

    string humanEmotions[7] = {"angry", "disgust", "fear", "happy", "sad", "surprise",
"neutral"};
    string robotEmotions[7] = {"embarrasment", "disgust", "sadness", "resentment", "positive
excitement", "surprise", "pride"};

    string imagePaths[7] =
{"/home/turtlebot/catkin_ws/src/mie443_contest3/images/embarrasment.jpeg",

"/home/turtlebot/catkin_ws/src/mie443_contest3/images/disgust.jpeg",
                            "/home/turtlebot/catkin_ws/src/mie443_contest3/images/sad.jpeg",

"/home/turtlebot/catkin_ws/src/mie443_contest3/images/resentment.jpeg",

"/home/turtlebot/catkin_ws/src/mie443_contest3/images/excited.jpeg",

"/home/turtlebot/catkin_ws/src/mie443_contest3/images/surprise.jpeg",

"/home/turtlebot/catkin_ws/src/mie443_contest3/images/proud.jpeg"};

    string soundFiles[7] = {"embarrasment.wav",
                            "disgust.wav",
                            "sad.wav",
                            "resentment.wav",
                            "happy.wav",
                            "surprise.wav",
                            "proud.wav"};

    if(emotionDetected >= 0 && emotionDetected <= 6){
        cout << "Detected " << humanEmotions[emotionDetected] << ". Responding with " <<
robotEmotions[emotionDetected] << "\n";

        Mat Image = imread(imagePaths[emotionDetected], CV_LOAD_IMAGE_UNCHANGED);
        if (Image.empty()){cout << "Error loading image" << endl;}

        namedWindow("robotEmotion", CV_WINDOW_NORMAL);
        imshow("robotEmotion", Image);
        waitKey(3500);
        //destroyWindow("robotEmotion");

        std::string path_to_sounds = ros::package::getPath("mie443_contest3") + "/sounds/";
        sc.playWave(path_to_sounds + soundFiles[emotionDetected]);
        destroyWindow("robotEmotion");
    }
}

int main(int argc, char** argv) {
    // Setup ROS.
    ros::init(argc, argv, "contest3");
    ros::NodeHandle n;

    // Frontier exploration algorithm.
    explore::Explore explore;

    // Class to handle sounds.
    sound_play::SoundClient sc;
    ros::Duration(0.5).sleep();

    // Publishers
    ros::Publisher vel_pub = n.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop",
1);
    geometry_msgs::Twist vel;

    /*ros::Publisher led1_pub = n.advertise<kobuki_msgs::Led>("mobile_base/commands/led1",
1);
    ros::Publisher led2_pub = n.advertise<kobuki_msgs::Led>("mobile_base/commands/led2", 1);
    kobuki_msgs::Led colour1, colour2;
    const uint8_t BLACK = 0, GREEN = 1, ORANGE = 2, RED = 3;*/

    // Subscribers
    ros::Subscriber bumper_sub = n.subscribe("mobile_base/events/bumper", 10,
&bumperCallback);
    ros::Subscriber odom = n.subscribe("odom", 1, &odomCallback);
    ros::Subscriber laser_sub = n.subscribe("scan", 10, &laserCallback);
    ros::Subscriber frontier_sub = n.subscribe("contest3/frontiers", 10, &markerCallback);
    ros::Subscriber emotion_sub = n.subscribe("/detected_emotion", 1, &emotionCallback);

    // Robot pose object + subscriber.
    RobotPose robotPose(0,0,0);
    ros::Subscriber amclSub = n.subscribe("/amcl_pose", 1, &RobotPose::poseCallback,
&robotPose);

    bool validPlan = true, navSuccess = false;
    float oldX = posX, oldY = posY, rotationSpeed = M_PI/4;
    int idx = 0, prevNumRed = 0;
    float xx, yy;
    geometry_msgs::Point point;
    std::vector<int> redFrontiersSortedIndices;
    std::vector<float> radii = {1.5, 1.0, 2.0, 2.5, 3.0, 0.5};
    std::vector<float> angles = {0.0, M_PI/3, -M_PI/3, 2*M_PI/3, -2*M_PI/3, M_PI-0.01};

    // Contest count down timer
    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    uint64_t secondsElapsed = 0;

    std::cout << "Spin 1 started \n";
    rotateThruAngle(copysign(2*M_PI - 0.01, chooseAngular(50, 0.6)), rotationSpeed, yaw,
0.1, &vel, &vel_pub, &secondsElapsed, start);
    std::cout << "Spin 1 complete \n";
    while(ros::ok() && secondsElapsed <= 1200) {
        explore.start();
        //For publishing colours to LEDs
        //colour1.value = RED;
        //ROS_INFO("Colour: %d", colour1.value);
        //led1_pub.publish(colour1);
        //colour2.value = GREEN;
        //ROS_INFO("Colour: %d", colour2.value);
        //led2_pub.publish(colour2);

        if(emotionDetected >=0){
            explore.stop();
            ROS_INFO("Emotion detected %d", emotionDetected);
            //Do actions here
            robotReaction(sc);
            //Emotion responses finished
            emotionDetected = -1;
            explore.start();
        }

        if(anyBumperPressed()){
            bumperPressedAction(&vel, &vel_pub, &secondsElapsed, start);
        }

        if(emotionDetected == -1){
            if(redFrontiers.size() > prevNumRed){
                std::cout << "New red frontier!!";
                /*explore.stop();
                moveThruDistance(-0.6, 0.25, posX, posY, &vel, &vel_pub, &secondsElapsed,
start);
                rotateThruAngle(copysign(2*M_PI - 0.01, chooseAngular(50, 0.6)),
rotationSpeed, yaw, 0.2, &vel, &vel_pub, &secondsElapsed, start);
                explore.start();*/
            }
            prevNumRed = redFrontiers.size();
        }

        if(emotionDetected == -1 && allFrontiersRed){
            explore.start();
            ros::spinOnce();
            //Navigate to closest point of closest frontier(s)
            if(redFrontiers.size() == 1){
                // Single red frontier
                point = getClosestPoint(redFrontiers[0], robotPose);
                navSuccess = navigateNearby(point, radii, angles, n, robotPose);
                ros::spinOnce();
                // Spin 15-60 deg in place if frontiers still red
                if (emotionDetected == -1 && allFrontiersRed){
                    std::cout << "Spin in 1 frontier red started \n";
                    rotateThruAngle(copysign(randBetween(M_PI/12, M_PI/3), chooseAngular(50, 
0.6)), rotationSpeed, yaw, 0.0, &vel, &vel_pub, &secondsElapsed, start);
                    std::cout << "Spin in 1 frontier red ended \n";
                }
                ros::spinOnce();
                // Go to furthest point in frontier if still red
                if(emotionDetected == -1 && allFrontiersRed){
                    point = getFurthestPoint(redFrontiers[0], robotPose);
                    navSuccess = navigateNearby(point, radii, angles, n, robotPose);
                }
            }
            else{
                // Multiple red frontiers
                // Sort the red frontiers by distance closest to the turtlebot
                // Try navigating to closest point of closest frontier
                redFrontiersSortedIndices = orderIndices(redFrontiers, robotPose);
                for(int i = 0; i < redFrontiersSortedIndices.size(); i++){
                    idx = redFrontiersSortedIndices[i];
                    //point = getCentroid(redFrontiers[idx]);
                    ros::spinOnce();
                    point = getClosestPoint(redFrontiers[idx], robotPose);
                    navSuccess = navigateNearby(point, radii, angles, n, robotPose);
                    ros::spinOnce();
                    if(emotionDetected != -1){break;}
                    // Spin 15-60 deg in place if all frontiers still red
                    if (emotionDetected == -1 && allFrontiersRed){
                        std::cout << "Spin in 2+ frontier red started \n";
                        rotateThruAngle(copysign(randBetween(M_PI/12, M_PI/3), 
chooseAngular(50, 0.6)), rotationSpeed, yaw, 0.0, &vel, &vel_pub, &secondsElapsed, start);
                        std::cout << "Spin in 2+ frontier red ended \n";
                    }
                    ros::spinOnce();
                    // Go to furthest point in frontier if all still red
                    if(emotionDetected == -1 && allFrontiersRed){
                        point = getFurthestPoint(redFrontiers[idx], robotPose);
                        navSuccess = navigateNearby(point, radii, angles, n, robotPose);
                        ros::spinOnce();
                    }
                }
            }
            // Do below if none of the above worked
            if(!navSuccess && emotionDetected == -1 && allFrontiersRed){
                std::cout << "ALL NAV ATTEMPTS UNSUCCESSFUL! \n";
                explore.stop();
                explore.start();
                // Force movement
                linear = 0.25;
                angular = 0;
                ros::spinOnce();

                if(emotionDetected == -1 && allFrontiersRed){
                    moveThruDistance(0.85, 0.15, posX, posY, &vel, &vel_pub,
&secondsElapsed, start);
                }
                //Rotate 10 to 45 in clearer direction 60% of the time
                if(emotionDetected == -1 && allFrontiersRed){
                    std::cout << "Spin in all nav failed started \n";
                    rotateThruAngle(copysign(randBetween(M_PI/12, M_PI/4), chooseAngular(50,
0.6)), rotationSpeed, yaw, 0.0, &vel, &vel_pub, &secondsElapsed, start);
                    std::cout << "Spin in all nav failed ended \n";
                }
            }
        }

        ros::spinOnce();
        secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start).cou
nt();
        ros::Duration(0.01).sleep();
    }
    explore.stop();
    std::cout << "20 MINUTES ELAPSED";
    return 0;
}
```

# Appendix B - Complete C++ Code - explore.cpp

A copy of the code can be found here.

```
/*********************************************************************
 *
 * Software License Agreement (BSD License)
 *
 *  Copyright (c) 2008, Robert Bosch LLC.
 *  Copyright (c) 2015-2016, Jiri Horner.
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of the Jiri Horner nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 *********************************************************************/

#include <explore.h>

#include <thread>

inline static bool operator==(const geometry_msgs::Point& one,
                              const geometry_msgs::Point& two)
{
  double dx = one.x - two.x;
  double dy = one.y - two.y;
  double dist = sqrt(dx * dx + dy * dy);
  return dist < 0.01;
}

namespace explore
{
Explore::Explore()
  : private_nh_("~")
  , tf_listener_(ros::Duration(10.0))
  , costmap_client_(private_nh_, relative_nh_, &tf_listener_)
  , move_base_client_("move_base")
  , prev_distance_(0)
  , last_markers_count_(0)
{
  double timeout;
  double min_frontier_size;
  private_nh_.param("planner_frequency", planner_frequency_, 1.0);
  private_nh_.param("progress_timeout", timeout, 30.0);
  progress_timeout_ = ros::Duration(timeout);
  private_nh_.param("visualize", visualize_, false);
  private_nh_.param("potential_scale", potential_scale_, 1e-3);
  private_nh_.param("orientation_scale", orientation_scale_, 0.0);
  private_nh_.param("gain_scale", gain_scale_, 1.0);
  private_nh_.param("min_frontier_size", min_frontier_size, 0.5);

  search_ = frontier_exploration::FrontierSearch(costmap_client_.getCostmap(),
                                                 potential_scale_, gain_scale_,
                                                 min_frontier_size);

  if (visualize_) {
    marker_array_publisher_ =
        private_nh_.advertise<visualization_msgs::MarkerArray>("frontiers", 10);
  }

  ROS_INFO("Waiting to connect to move_base server");
  move_base_client_.waitForServer();
  ROS_INFO("Connected to move_base server");

  exploring_timer_ =
      relative_nh_.createTimer(ros::Duration(1. / planner_frequency_),
                               [this](const ros::TimerEvent&) { makePlan(); });
}

Explore::~Explore()
{
  stop();
}

void Explore::visualizeFrontiers(
    const std::vector<frontier_exploration::Frontier>& frontiers)
{
  std_msgs::ColorRGBA blue;
```

```
blue.r = 0;
blue.g = 0;
blue.b = 1.0;
blue.a = 1.0;
std_msgs::ColorRGBA red;
red.r = 1.0;
red.g = 0;
red.b = 0;
red.a = 1.0;
std_msgs::ColorRGBA green;
green.r = 0;
green.g = 1.0;
green.b = 0;
green.a = 1.0;

ROS_DEBUG("visualising %lu frontiers", frontiers.size());
visualization_msgs::MarkerArray markers_msg;
std::vector<visualization_msgs::Marker>& markers = markers_msg.markers;
visualization_msgs::Marker m;

m.header.frame_id = costmap_client_.getGlobalFrameID();
m.header.stamp = ros::Time::now();
m.ns = "frontiers";
m.scale.x = 1.0;
m.scale.y = 1.0;
m.scale.z = 1.0;
m.color.r = 0;
m.color.g = 0;
m.color.b = 255;
m.color.a = 255;
// lives forever
m.lifetime = ros::Duration(0);
m.frame_locked = true;

// weighted frontiers are always sorted
double min_cost = frontiers.empty() ? 0. : frontiers.front().cost;

m.action = visualization_msgs::Marker::ADD;
size_t id = 0;
for (auto& frontier : frontiers) {
  m.type = visualization_msgs::Marker::POINTS;
  m.id = int(id);
  m.pose.position = {};
  m.scale.x = 0.1;
  m.scale.y = 0.1;
  m.scale.z = 0.1;
  m.points = frontier.points;
  if (goalOnBlacklist(frontier.centroid)) {
    m.color = red;
  } else {
    m.color = blue;
    }
    markers.push_back(m);
    ++id;
    m.type = visualization_msgs::Marker::SPHERE;
    m.id = int(id);
    m.pose.position = frontier.initial;
    // scale frontier according to its cost (costier frontiers will be smaller)
    double scale = std::min(std::abs(min_cost * 0.4 / frontier.cost), 0.5);
    m.scale.x = scale;
    m.scale.y = scale;
    m.scale.z = scale;
    m.points = {};
    m.color = green;
    markers.push_back(m);
    ++id;
  }
  size_t current_markers_count = markers.size();

  // delete previous markers, which are now unused
  m.action = visualization_msgs::Marker::DELETE;
  for (; id < last_markers_count_; ++id) {
    m.id = int(id);
    markers.push_back(m);
  }

  last_markers_count_ = current_markers_count;
  marker_array_publisher_.publish(markers_msg);
}

void Explore::makePlan()
{
  // find frontiers
  auto pose = costmap_client_.getRobotPose();
  // get frontiers sorted according to cost
  auto frontiers = search_.searchFrom(pose.position);
  ROS_DEBUG("found %lu frontiers", frontiers.size());
  for (size_t i = 0; i < frontiers.size(); ++i) {
    ROS_DEBUG("frontier %zd cost: %f", i, frontiers[i].cost);
  }

  if (frontiers.empty()) {
    std::cout << "Frontiers Empty! Calling stop \n";
    stop();
    return;
  }

  // publish frontiers as visualization markers
  if (visualize_) {
    visualizeFrontiers(frontiers);
  }
  // find non blacklisted frontier
  auto frontier =
      std::find_if_not(frontiers.begin(), frontiers.end(),
                       [this](const frontier_exploration::Frontier& f) {
                         return goalOnBlacklist(f.centroid);
                       });
  if (frontier == frontiers.end()) {
    stop();
    return;
  }
  geometry_msgs::Point target_position = frontier->centroid;
  // time out if we are not making any progress
  bool same_goal = prev_goal_ == target_position;
  prev_goal_ = target_position;
  if (!same_goal || prev_distance_ > frontier->min_distance) {
    // we have different goal or we made some progress
    last_progress_ = ros::Time::now();
    prev_distance_ = frontier->min_distance;
  }
  // black list if we've made no progress for a long time
  if (ros::Time::now() - last_progress_ > progress_timeout_) {
    frontier_blacklist_.push_back(target_position);
    std::cout << "Adding current goal to black list";
    makePlan();
    return;
  }

  // we don't need to do anything if we still pursuing the same goal
  if (same_goal) {
    return;
  }

  // send goal to move_base if we have something new to pursue
  move_base_msgs::MoveBaseGoal goal;
  goal.target_pose.pose.position = target_position;
  goal.target_pose.pose.orientation.w = 1.;

  goal.target_pose.header.frame_id = costmap_client_.getGlobalFrameID();
  goal.target_pose.header.stamp = ros::Time::now();
  move_base_client_.sendGoal(
      goal, [this, target_position](
                const actionlib::SimpleClientGoalState& status,
                const move_base_msgs::MoveBaseResultConstPtr& result) {
        reachedGoal(status, result, target_position);
      }, MoveBaseClient::SimpleActiveCallback(), &feedbackCb);
}

bool Explore::goalOnBlacklist(const geometry_msgs::Point& goal)
{
  constexpr static size_t tolerace = 5;
  costmap_2d::Costmap2D* costmap2d = costmap_client_.getCostmap();
  // check if a goal is on the blacklist for goals that we're pursuing
  for (auto& frontier_goal : frontier_blacklist_) {
    double x_diff = fabs(goal.x - frontier_goal.x);
    double y_diff = fabs(goal.y - frontier_goal.y);

    if (x_diff < tolerace * costmap2d->getResolution() &&
        y_diff < tolerace * costmap2d->getResolution())
      return true;
  }
  return false;
}

void Explore::reachedGoal(const actionlib::SimpleClientGoalState& status,
                          const move_base_msgs::MoveBaseResultConstPtr&,
                          const geometry_msgs::Point& frontier_goal)
{
  ROS_DEBUG("Reached goal with status: %s", status.toString().c_str());
  if (status == actionlib::SimpleClientGoalState::ABORTED) {
    frontier_blacklist_.push_back(frontier_goal);
    ROS_DEBUG("Adding current goal to black list");
  }

  // find new goal immediatelly regardless of planning frequency.
  // execute via timer to prevent dead lock in move_base_client (this is
  // callback for sendGoal, which is called in makePlan). the timer must live
  // until callback is executed.
  oneshot_ = relative_nh_.createTimer(
      ros::Duration(0, 0), [this](const ros::TimerEvent&) { makePlan(); },
      true);
}

void Explore::start()
{
  exploring_timer_.start();
}

void Explore::stop()
{
  move_base_client_.cancelAllGoals();
  exploring_timer_.stop();
  ROS_INFO("Exploration stopped.");
}

}  // namespace explore

// int main(int argc, char** argv)
// {
//   ros::init(argc, argv, "explore");
//   if (ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME,
//                                      ros::console::levels::Debug)) {
```

```
//      ros::console::notifyLoggerLevelsChanged();
//    }
//    explore::Explore explore;
//    ros::spin();

//    return 0;
// }
```

## Appendix C - Complete C++ Code - navigation.cpp

A copy of the code can be found here.

```
#include <navigation.h>

//bool cancelGoals = false;
void feedbackCb(const move_base_msgs::MoveBaseFeedback::ConstPtr& feedback){
    //std::cout << "In feedback callback \n";
    ros::spinOnce();
    if(emotionDetected != -1){
        std::cout << "Emotion detected while in move_base! \n";
        //cancelGoals = true;
    }
}

bool Navigation::moveToGoal(float xGoal, float yGoal, float phiGoal, float timeout){
       // Set up and wait for actionClient.
    MoveBaseClient ac("move_base", true);
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server to come up");
    }
       // Set goal.
    geometry_msgs::Quaternion phi = tf::createQuaternionMsgFromYaw(phiGoal);
    move_base_msgs::MoveBaseGoal goal;
    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();
    goal.target_pose.pose.position.x =  xGoal;
    goal.target_pose.pose.position.y =  yGoal;
    goal.target_pose.pose.position.z =  0.0;
    goal.target_pose.pose.orientation.x = 0.0;
    goal.target_pose.pose.orientation.y = 0.0;
    goal.target_pose.pose.orientation.z = phi.z;
    goal.target_pose.pose.orientation.w = phi.w;
    ROS_INFO("Sending goal location ...");
       // Send goal and wait for response.
    ac.sendGoal(goal, MoveBaseClient::SimpleDoneCallback(),
MoveBaseClient::SimpleActiveCallback(), &feedbackCb);
    //ac.sendGoal(goal);
    ac.waitForResult(ros::Duration(timeout));
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
        ROS_INFO("You have reached the destination");
        return true;
    } else {
        ROS_INFO("The robot failed to reach the destination");
        return false;
    }
}
```

## Appendix D - CNN Training Code

All training code for the CNN classifier can be found here, and has not been reproduced below due to the file with outputs being too long. Our final model architecture is given below, and is referred to as `EmotionClassificationNet_Better` in the Colab notebook linked above.

```
class EmotionClassificationNet_Better(nn.Module):

    def __init__(self, dropout=0.25):
        super(EmotionClassificationNet_Better, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(1, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),

            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),

            nn.Conv2d(128, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),

            nn.Conv2d(128, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),

            nn.Conv2d(128, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(),
            nn.Dropout(dropout),
        )
        self.nn = nn.Sequential(
            nn.Linear(1152*2,512),
            nn.LeakyReLU(),
            nn.Dropout(dropout),
            nn.Linear(512, 256),
            nn.LeakyReLU(),
            nn.Dropout(dropout),
            nn.Linear(256, 7)
        )

    def forward(self, x, test_mode=False):
        batch_size = x.shape[0]
        feats = self.cnn(x)
        out = self.nn(feats.view(batch_size, -1))
        #
        # If we are testing then return prediction index.
        if test_mode:
            _, out = torch.max(out, 1)
        return out
```

## Appendix E - Attribution Table

| **Category** | | **Farhan** | **Henry** | **Yilin** |
|---|---|---|---|---|
| **Code** | Strategy | RS | RS | |
| | Handler Functions | RD | RD | RS, RD |
| | High-level Strategy Functions | RD, MR | RD | |
| | Integration | MR, ET | | |
| | Clean-up | MR, ET | | |
| | Testing | MR, ET, FP | FP | FP |
| **Report** | FOCs | | | RD |
| | Strategy | ET | RD | RD |
| | Sensors | MR, ET | ET | RS, RD |
| | Controller Design | RD, MR, ET | | RS |
| | Complete Report | FP | FP, ET | CM, RD, ET, FP |

| | |
|---|---|
| RS | Research |
| RD | Wrote first draft |
| MR | [CODE] Major revision to strategy, functions used<br>[REPORT] Major revision to organization |
| ET | [CODE] Edited for errors, de-bugging<br>[REPORT] Edited for grammar and spelling |
| FP | [CODE] Final check for applicability to practice worlds<br>[REPORT] Final check for flow and consistency |
| CM | Responsible for compiling the elements into the complete document |