Contest 3 Report

MIE443 Project

Savo Bajic - 1003051485
Maximilian Glidden - 1002277396
Catherine Kucaba - 1003278026

1.0 Problem Definition and Design Requirements

The objectives and requirements for this portion of the project are outlined in the Contest 3 Manual [1]. The primary goal of this contest is to develop an algorithm that allows a robot to autonomously explore and map an unknown disaster environment and locate 7 victims (users). An example environment is shown in Figure 1 below. In addition, the robot must interact with the victims and provide them with important emergency announcements regarding evacuation due to the team's chosen disaster, which is described later in this section. During the interaction, the robot must correctly identify the current emotional state of the victim and appropriately interact with them based on this information. The robot must locate and interact with all 7 victims within a certain time limit. Due to the Covid-19 pandemic, this project is completed only using software, as it would typically involve a physical robot navigating a real environment.

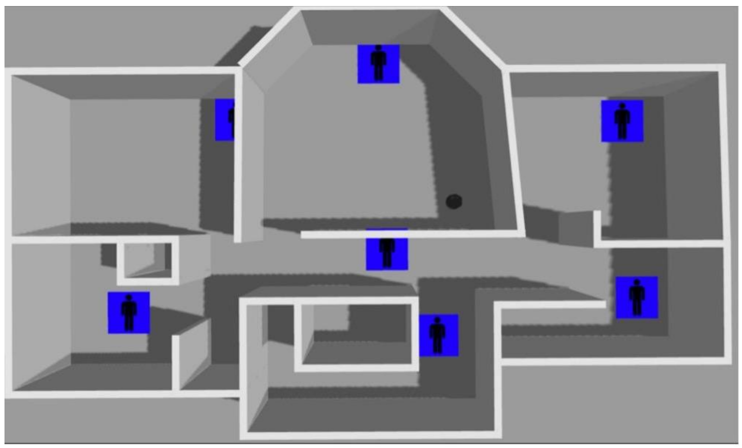

Figure 1: Example environment layout with 7 victims [1]

The robot is simulated using TurtleBot and it navigates an enclosed environment created in gazebo. While traversing the areas, the robot will use the ROS gmapping package to dynamically create a map from information provided by the sensors on the TurtleBot, including a Kinect sensor and front bumper sensors. This map is then used in the frontier based exploration algorithm to help navigate the robot to regions it has not yet explored. The robot must appropriately identify and react to the 7 different user emotions: anger, neutral, fear, sadness, happiness, surprise, and disgust. Figure 2 below displays sample images for each of the victims' emotions. The interactions must be distinct and identifiable, using primary/reactive and secondary/deliberative emotions. The team chose the disaster to be a fight between Godzilla and King Kong, due to the recent movie that was released titled *Godzilla vs. Kong* (Adam Wingard, 2021). The victims are required to evacuate the building before Godzilla and King Kong begin to fight in the city, in the event that the building is demolished in the ensuing battle. The victims are required to get to the roof of the building as helicopters are helping to evacuate citizens.

The robot will be tested in two trials, with the best counting towards the final mark. Scoring is based on two aspects: locating and classifying all of the victims' emotions and the robot's emotional responses while interacting with the victims. Additional marks will be given based on the creativity and complexity of the emotional behaviours, as well as the team's classification rate compared to the other teams.

Figure 2: Sample images of the 7 victims' emotions [1]

The concepts required for this project can be applied to many other industries that are starting to use and rely on robots to execute tasks typically performed by humans. The navigation and exploration portions of this project are similar to what search and rescue robots accomplish, as they must navigate unknown terrain to locate and possibly assist survivors after a disaster. Another similar scenario is presented for service robots in the food or medical industries. Robots that interact with human patients or customers must be able to sense and classify their emotions in order to provide appropriate emotional behaviours in response, so the person can understand and cooperate. For example, if a customer is angry, the robot should respond in a manner to calm the customer and determine what needs to be done to help them. As people begin to use robots for customer service, these robots must be able to perform the emotional labour that comes with these roles.

1.1 Requirements and Constraints

For this contest, there are some design requirements that the team must take into account when developing the algorithms for the robot. These requirements can be found in the Contest 3 Manual [1] and are also presented below.

1. The contest environment is a building with multiple rooms (see Figure 1 as an example).
    a. There are seven (7) individuals distributed throughout the environment.
    b. The victims are stationary in the environment (i.e., they do not move).
    c. There are no objects/obstacles in the environment other than walls.
    d. Victim locations are marked by a person on a blue square purely for visualization (this does not impact exploration).
    e. The robot will start at a location chosen by the teaching team.
2. The layout of the contest environment is not known to the team in advance.
    a. The teaching team will generate a new environment for the trials differing from the one provided for practice.
3. The robot must locate and interact with all 7 victims within a maximum time limit of 20 minutes.
4. There are two trials per team, with the best trial counting towards the final score.
5. For testing, the team was provided with the following:
    a. A dataset of faces expressing one of the seven emotions: anger, neutral, fear, sadness, happiness, surprise, and disgust.
    b. Sample code used to train an emotion classification convolutional neural network.
    c. A victim detector that automatically detects simulated victims.

        i. When the robot approaches a set distance from the victim, multiple images of different faces with the same emotion are provided for the CNN to classify.
        ii. For the trial runs, a different dataset of face images will be used.
        iii. The overall accuracy of the team's emotion classification method will be tested using the dataset mentioned in ii.
    d. A frontier exploration algorithm used to find the victims.
        i. The robot generates a map of the environment by navigating to the boundaries between known and unknown regions.

6. During interactions, the robot is required to have unique emotional responses to the 7 possible emotions (shown in Figure 2). It must also convey information and evacuation instructions regarding the disaster.
    a. The disaster is chosen by the team.
7. For the robot's responses, the team must choose 3 unique primary emotions, 3 unique secondary emotions and 1 additional unique emotion of either type from the following list: fear, positively excited, pride, anger, sad, discontent, hate, resentment, surprise, embarrassment, disgust, and rage.
    a. The team must implement original motion and/or sound for the interactions.
8. The team can utilize different emotion classification methods than provided, but cannot install additional software packages.
9. The team cannot use the 'detectedVictim.txt' file for emotion recognition during the trials. However, it can be used for testing and debugging in practice runs.
10. Automatically generated log files cannot be modified.

2.0 Methods and Strategy

The team implemented supervised learning for the machine learning portion of this project. During this learning method, the robot is provided many examples of input/output pairs. The robot then develops its own function(s) that can make decisions when provided a given input. For this contest, the robot was provided a dataset that included many different face images expressing the seven victims' emotions. This form of learning is usually accomplished using neural networks; for this project, the team utilized a convolutional neural network (CNN) to classify images. A k-fold approach was used for training the AI to maximize the utility of the limited training set provided to the team. The dataset was split into five equal portions (k = 5). Five models were then trained separately, with each one using a different set of data for validation (e.g., model 1 used fold 1 for validation) and the other folds for training (i.e., 2 to 5). Of these five models, the one that performed best at correctly identifying the emotions of its validation set was kept and used in the project. At times, the classification accuracy varied significantly among the five models, from ~93% to ~62% (shown in Figure 3). Other learning methods, such as unsupervised or reinforced, were not suitable nor required for this project.

```
Done training the model!
Best validation score: 92.82 | Average across folds: 81.03
[0.6161819306930693, 0.7738242574257426, 0.8217821782178217, 0.9282178217821783, 0.911664603960396
```

Figure 3: Output file from a practice run displaying each model's classification accuracy

For exploration and navigation, the team implemented a greedy frontier-based algorithm. This method utilizes an occupancy map/grid that is generated by sensor readings obtained from the robot while it moves throughout the environment. The boundaries between open space and unexplored space, as determined by the sensory information, are the frontier regions. The robot is inclined to move towards these areas in order to explore and map more of the environment. The team's robot begins by spinning in place and taking a rotating scan of the environment. It then uses built-in ROS explore functions to move the robot throughout the regions. During its motion, the robot takes in the environment (via sensors) and generates a map using the ROS gmapping package. This frontier-based algorithm was selected over other methods, such as random walk, as it is more efficient at exploring the environment and locating the victims that the robot is required to interact with. As the team saw in Contest 1, a random or weighted random walk would sometimes result in the robot becoming trapped in certain areas of the map and thus would no longer explore. Now that the robot is inclined to explore unknown regions, it will explore more of the environment and find the victims faster than the other methods.

In the simulation, the robot does not locate victims using its sensors. Instead, it identifies or senses victims when it moves within a certain set distance away from them, approximately 1m for the team's practice runs. This is due to the way the contest is set up by the teaching team. In a real world scenario, victims could be located by looking for humanoid figures or faces and then approaching them to confirm. Once the victim is located, a dataset of images is directly provided to the robot of the same emotion being expressed. Therefore, sensors are again not used for this portion of the contest. The CNN and NN are then used to process the emotions and attempt to classify it into one of the seven victims' emotions. In a real world scenario, the RGB camera sensor would be used to scan for faces and capture images to then process.

The robot's emotional responses are based on behavioural control. When the robot encounters a victim and processes their emotion, it will then react according to a set of behaviours the team programmed into the algorithm. Each victim's emotion has a distinct robot emotional reaction. The team was required to use 3 primary emotions and 3 secondary emotions, with the last one being of either category. The justification for the use of each emotion and its classification as either primary or secondary is explained below.

1. Victim's Emotion: **Anger**
   Robot's Emotional Response: **Resentment**
   Type of Emotion: **Secondary**

   The team classified resentment as a secondary emotion, as it is quite complex and is usually a mixture of other emotions. One experiences resentment when they feel that they have been treated unfairly by others, meaning that one must think/reflect before experiencing this emotion. The team used resentment as a reaction to an angry victim, since they felt that this would be the most natural response of a rescuer attempting to help this type of victim. A person may feel resentment when trying to help someone who is angry, especially because angry people tend to be stubborn and do not listen (due to their emotional state). The team also believed that resentment would be a good response to change the victim's emotion away from anger and more towards concern for their own safety, so much so that they are willing to listen to instructions and comply. When met with resentment, a person tends to evaluate themself and how they treated others in an attempt to adjust their behaviour.

2. Victim's Emotion: **Neutral**
   Robot's Emotional Response: **Pride**
   Type of Emotion: **Secondary**

   The team classified pride as a secondary emotion, as one must reflect on achievements (theirs or others) to derive that deep sense of satisfaction. This is another emotion that is complex and tends to include other emotions (like happiness, excitement, etc.). The team used pride as a reaction to a neutral victim as this was one of the few positive emotions available to the team. They believed this would be a better reaction to finding a neutral person, since a negative emotion displayed by the robot could then upset the victim. Pride is a typical emotion one would feel when finding a survivor and assisting them, especially because they may have saved that person's life, which is a major accomplishment one would be proud of. In addition, people with pride tend to display a high level of confidence, which also helps to communicate information to others. People who are confident tend to be listened to more than people who are not.

3. Victim's Emotion: **Fear**
   Robot's Emotional Response: **Embarrassment**
   Type of Emotion: **Secondary**

   The team classified embarrassment as a secondary emotion because one must think about their actions, reflect on what occurred, and then proceed to feel shame or awkwardness that leads to the emotion. The team used embarrassment as a reaction to a fearful victim for two particular reasons. Firstly, stumbling upon a person who is displaying fear may make one feel shameful or awkward for possibly inducing that fear. Thus, it may naturally follow that one would feel embarrassment for causing that kind of emotion in someone. In stressful situations, such as encountering someone who is fearful, a person may feel a sense of awkwardness on how to handle the situation, and sometimes they may resort to feeling embarrassed when trying to help. In addition, the team also believed a safe reaction to fear would be embarrassment in order to not scare the victim further, and thus have them no longer willing to listen to instructions. When a person is embarrassed, people tend to feel a sense of pity and try to help or listen to what that person may need. In a similar manner, the team is trying to ensure that the fearful victim listens to the instructions to evacuate.

4. Victim's Emotion: **Sadness**
   Robot's Emotional Response: **Anger**
   Type of Emotion: **Primary**

   The team classified anger as a primary emotion, given that it is a very primal and inherent emotion among people and animals. Although anger can be a secondary emotion, at certain times it does not require one to think and is, instead, closer to a reflex (e.g., becoming angry when in pain). The team chose anger as a reaction to a victim displaying sadness as they believed a first reaction to seeing a sad individual during an emergency would be anger. If a rescuer encountered a person who was sad, and thus possibly unwilling to move because of it, they may become angry at the individual for not caring about the situation and trying to get out safely. In addition, a rescuer may become angry that they are risking their own life to save someone who seems to not care. The team also thought that the use of anger would push the victim out of their sadness enough to evacuate safely. For example, a rescuer has to tell someone to “snap out of it” to bring that person back to the reality of the critical situation. In this way, the team would hope that the robot displaying an angry emotion would motivate the sad person to listen to the robot.

5. Victim’s Emotion: **Happiness**
   Robot’s Emotional Response: **Positively Excited**
   Type of Emotion: **Primary**

   The team classified positively excited as a primary emotion because it is one that is quickly experienced when something good happens. A person rarely has to think in order to experience this emotion and it is very similar to the feeling of joy (another primary emotion). The team chose positively excited as a reaction to happiness since they believed that positive victim emotions should be responded to positively. This is because a negative reaction may cause the victim to then feel fear or sadness, which is counterintuitive to helping them (as they may no longer want help). Therefore, the team believed “matching” the energy of the victim would be helpful in communicating important evacuation information. In addition, rescuers who encounter happy survivors tend to display similar emotions of happiness as they are not only relieved to find the person, but they also do not wish to upset the person if there are other issues present (for example, if the person is pinned or harmed, negative emotions can send them into shock).

6. Victim’s Emotion: **Surprise**
   Robot’s Emotional Response: **Surprise**
   Type of Emotion: **Primary**

   The team classified surprise as a primary emotion, since it is a typical reaction to external stimuli that catch people off guard. In addition, it is a feeling that people can rarely control, yet it is simple in understanding its nature. The team used surprise as a reaction to a surprised victim as a way to ease the tension in the situation. A person who is experiencing the emotion of surprise is put into a vulnerable state, as the shock of the situation could wear off and they could return to a neutral/positive state (for example, being surprised by a party) or the situation could worsen, such as the person developing fear or terror. Therefore, responding to a surprised person in a negative manner could be inappropriate. Instead, the team decided that the robot could reciprocate the emotion and act surprised as well to have found the person. This type of reaction could de-escalate the situation, as the person who is surprised would acknowledge that they surprised the robot in return. Thus, they may no longer be surprised and return to a calmer state in order to receive instructions.

7. Victim’s Emotion: **Disgust**
   Robot’s Emotional Response: **Discontent**
   Type of Emotion: **Secondary**

   The team classified discontent as a secondary emotion since one needs to consider their current situation in order to feel a sense of dissatisfaction with it. It is one of those emotions that, although seems simple, is actually quite complex and can stem from multiple feelings. The team used discontent as a reaction to a disgusted person in a similar manner to using resentment towards anger. The team thought that a rescuer who encountered a disgusted person may feel uncomfortable and unhappy about the situation, especially if the disgust stems from the interaction. Given that discontent is similar to disgust, but at a lesser intensity, the team would hope that the victim would not be further upset by the situation and thus unwilling to cooperate. Instead, the idea is that the robot would display a similar emotion in an attempt to make the victim understand their own emotions. Therefore, they would be more likely to receive help from the robot, especially if they see that their emotions are beginning to upset the entity that is trying to help them.

3.0 Detailed Robot Design and Implementation

3.1 Sensory Design

The simulated Turtlebot platform is equipped with a variety of sensors to aid in navigation, including touch bumpers, a depth camera, cliff sensors, and odometry data. As in Contest 1, the environment the robot is exploring in Contest 3 is flat and enclosed. Cliff sensors do not produce much useful information; there are no edges or changes in elevation for the robot to detect. Similar to Contest 1, the robot does not have a map of its environment prior to initialization. The laser scanner, odometry, touch bumpers, and gyroscope were used throughout the *move_base* and *gmapping* ROS nodes, feeding into the *explore_lite* node to aid in exploration and obstacle avoidance. During the robot’s emotional interactions, the odometry is used directly to aid in the control of the robot. The other sensors are handled directly by the aforementioned ROS topics without any action required of the team’s code to aid in their roles.

3.1.1 Bumpers

The Turtlebot has a set of three bumpers located on the front, left, and right sides. These bumpers publish their state (1 when pressed, 0 otherwise), which in turn is fed to the *gmapping* node. The *gmapping* package adds bumper contact to the map by default, improving the fidelity of the map in situations where obstacles are not detected by the depth camera (i.e when moving backwards during an interaction).

3.1.2 Depth Camera/Laser Scanner

The Turtlebot is equipped with a Microsoft Kinect camera system, which integrates both an RGB and depth camera. The depth camera can be used to approximate the functionality of a laser scanner, which generates a distance at each point in a one-dimensional array from the depth camera input. The motivation for using the depth camera in this way was to provide sensory input to the *gmapping* and *move_base* nodes regarding the presence and proximity of obstacles before the robot. This depth data is used by *gmapping* to build a map of the environment, and by *move_base* to aid in obstacle avoidance and to build a costmap of the environment for movement planning.

3.1.3 Odometry

Encoders on the motors of the turtlebot provide information on the rotational velocity of each wheel, which is used with the forward differential kinematics model to produce linear and rotational velocities. Integrating these velocities over time produces an estimate of the robot's position and heading; this is the odometry data supplied by the *odom* topic.

Odometry data is used by the *move_base* node as an input to the local planner so it is aware of the rough distance travelled. It is also used by the *gmapping* node to stitch together a map of the environment from depth camera/laser scanner data, estimating the robot's pose with respect to the map. Odometry data was also utilized directly during the robot's emotional interactions with victims in the environment to close the control loop with respect to movement. Inputting movement commands with motor velocities and durations alone is an example of open-loop control. The robot does not track how far it has actually travelled, nor can it easily estimate its position in the world frame. Checking the odometry data during movements allows the robot to more accurately assess its motion when performing the multimodal emotional response.

3.1.4 Gyroscope

The Turtlebot's gyroscope provides information on the robot's angular velocity and orientation. This data is supplied to the *move_base* node as a sensor transform, and assists in movement and local planning.

3.1.5 RGB Camera

The Turtlebot's Kinect's integrated RGB camera provides a full colour video feed of the robot's surroundings. The team did not make use of the RGB camera during Contest 3, because the victims the robot needed to find were not physical objects in the environment. Instead, the victims were represented as a set of coordinates in the environment (stored in *victims.txt*). When the robot moves within a certain radius of these coordinates, the *victimLocator.py* function is triggered. This function publishes a set of images to the robot for emotion classification, simulating the use of the RGB camera for emotional analysis.

3.2 Controller Design

The overarching control architecture used for the third contest was a behaviour-based controller, implemented using a finite state machine. Combinations of a set of behaviours were encoded in various states ("explore", "emote", etc.), with transitions between states triggered by sensory input and the code of other states. The robot can only exist in a single state at a given time, but this state may encompass multiple sub-behaviours, called substates, within the team's code. The robot's behaviours are a combination of reactive and deliberative actions. Exploring the environment is a deliberate process, where the robot considers where it has and has not been. Emotional responses are reactive to external stimuli.

The main behaviour states are "Explore" (the main/default state) and the seven emotional reactions (Pride, Positive Excitement, Embarrassment, Surprise, Discontent, Anger, Resentment). The Explore State is the main state the robot takes while navigating the environment. In this state, the robot utilises the *explore_lite* package to explore the environment using a greedy frontier-based method. The environment is mapped using the *gmapping* package, while path planning, movement, and obstacle avoidance are handled by the *move_base* node. Occasionally, the robot will conduct a scan to reset the frontiers for exploration. When the *victimLocator.py* function publishes a victim emotion (indicating a victim has been found), the "explore" behaviour terminates and an emote behaviour is triggered. The seven emotional response states correspond to seven unique behaviours; each of these behaviours is associated with the emotional state of the victim. A state is selected based on the output of the *emotionalClassifier* function, and the associated multimodal response behaviour is performed. Once this response behaviour is complete, the robot re-enters the explore state.

3.2.1 Exploration and Mapping

Exploration is the primary state the robot is in when completing this challenge, as it tries to explore the entirety of the environment to find all seven victims within it. Exploration is largely handled using ROS's own built-in algorithms, functions, and services, as well as those provided to the team for use, namely the *Explore* class from the *explore* namespace. The team used this class to automatically explore the environment by simply calling *Explore.start()* to send it exploring or *Explore.stop()* when a controlled motion is needed again (e.g. a specific reaction to a victim).

The automated exploration is guided by a greedy frontier-based exploration approach. A frontier is defined as a division between mapped (visited) and unmapped regions of the environment, that a robot may cross. This is in contrast to a barrier which is an impassable division between regions of the map. Frontier exploration has the robot seek out and approach these frontiers to map what lies beyond them. To select which frontier of the many potentially present in an unfinished map, the robot uses a "greedy" strategy. Specifically the robot records the location of all known frontier points in its immediate vicinity (roughly 1.5m along each map axis) and averages them out to find the centroid, it then approaches this centroid and updates it as it moves. This is "greedy" because it is the local optimum, in that it is the boundary which is closest to the robot that is scanned, even if it is much smaller (and thus reveals less) than other frontiers further from the robot.

The exploration is enabled by the mapping services *gmapping* provides as part of the ROS package. As the robot travels through the environment, it uses the sensor data to continuously build a map of the robot's surroundings and estimate its position within this map. It is this *gmapping* that provides the information and guidance *Explore* requires to complete its function. The results of mapping, such as the figure provided below, are viewable using another ROS utility, *RVIZ*.

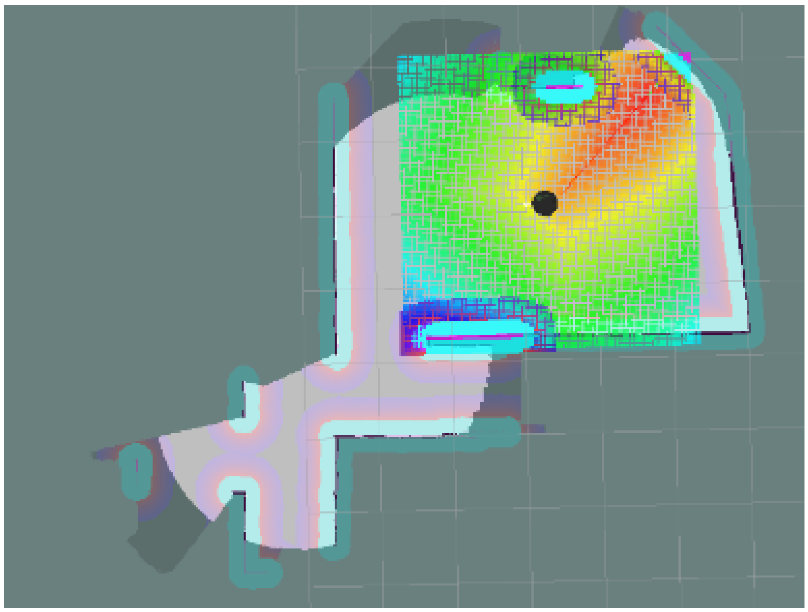

Figure 4: An example of a partially explored map. The rover lies in the center of the coloured section, the colours indicate the proximity to the robot's current exploration goal with a path drawn in red.

The team did not alter the behavior of *Explore* in any notable way, as it performed well as provided to the team. However, the team did implement supporting functions that would intervene and attempt to recover the robot from a position where *Explore* on its own would be stuck. *Explore*, due to its "greedy" algorithm, has the potential to get stuck when it arrives at a frontier centroid that does not reveal enough of the map to determine a distinct and new local frontier centroid compared to the one it just reached, and thus determines there is nowhere to go. To detect this condition, the team monitors the robot's distance travelled in the last period (roughly 10 seconds) to see if the robot has stopped moving any reasonable amount (1m). If it fails to surpass this target, the team has its code attempt a recovery, detailed more in section 3.2.2, Movement.

3.2.2 Movement

Although the majority of the robot's motion is left to be autonomously determined and handled by ROS, there are two primary states where the team's code opts to exert direct control over the robot's motion: reacting to victims and exploration recovery. To achieve direct control of the robot, the team reused the code they created for this purpose in Contest 1 as a baseline and developed it to face the unique challenges Contest 3 presented. This code is contained primarily in *movement.h* and *movement.cpp*.

Direct control is exerted on the rover by stopping the exploration (*Explore.stop()*) and then starting a publisher to the teleoperation topic within *contest3.cpp*. Any data published to this topic will emulate that of a user, and thus override any other sections of code attempting to steer the robot's motion. This allows a linear and rotational velocity to be set for the base by the code; however, most movements for the rover are meant to be measured (e.g. going 0.5m backwards). To enact such prescribed movements, odometry feedback is used to monitor the robot's motion to ensure that it moves the required distance and not to depend on maintaining course for a set duration of time. A callback function is prepared in *movement* to be used to subscribe to the odometry so it can be utilized by the rest of the program.

Upon those foundations, the main function in these files, *travel*, is built. This function is called repeatedly to execute a desired motion defined by displacements and velocities, linear and rotational. This function keeps a record of the most recent targets provided and uses the odometry to track the progress. Once the motion is complete, it returns a value of true to signal to the program it may proceed to the next task. This design made it very well suited to the state-machine approach of the team's control system because it allows motions to easily be interrupted and change. In addition, it does not hold execution away from the main loop for a significant portion of time.

The main addition to the *movement* files was the function *checkIfMoved* used to check if the rover has been moving recently. This is used to determine if the robot's exploration code has gotten "stuck" in a location and cannot determine where to go next (explained in section 3.2.1). To accomplish this, the code regularly records the robot's positional information to a circular buffer to determine the overall travel between the ten most recent recordings (covering roughly ten seconds of real-time motion). If this distance travelled falls below a set threshold (currently set to 1m), the robot is determined as stationary and this is returned to the code that called it to signal a recovery attempt is required.

The first recovery method for a stuck robot is having it complete a whole rotation about its current position to try and reveal any potential frontiers and/or barriers. Any new features will influence the calculation of the centroid and typically be enough to "dislodge" the robot. Should the robot centroid remain as it is regardless of this rotation (for example being in the center of a room so no new barriers are found) the team will then clear the frontier blacklist to try and alter the centroid's calculation.

3.2.3 Emotion Detection

Emotion detection in this competition is achieved using a convolutional neural network model trained and operated within Python (*emotionClassifier.py* and *emotionTrainingSample.py*, respectively). As the robot approaches victims it will be provided a series of images of their face and, using *emotionClassifier* with the trained model, it will process them and estimate the emotion portrayed by the victim's face. It then compares these results and selects the most common result from the processed series of images as the victim's emotion and publishes it as a ROS topic. The team's code uses a subscriber to *emotionClassifier* to collect these results and pass them onto the remainder of the code, namely to react to the victim and instruct them to evacuate properly (explained in greater detail in section 3.2.4).

The first step in accomplishing emotion detection is to design the neural network used to classify the emotions. This was done using the *emotionTrainingSample* file. The neural network has two main parts: a series of convolution blocks forming the convolutional neural network (CNN) followed by a series of fully connected layers forming a neural network (NN). The images are fed through the CNN and its four blocks of convolutions, then into the NN, which resolves it into buckets for each of the seven responses and the confidence that emotion is present. The structure of the overall network was kept the same with the exception of the addition of a fully connected layer in the NN.

This structure was trained using a dataset to try and best identify emotions in images. It was trained using a supervised learning method. A data set of pictures with corresponding labels for the emotion in each is split into a training set and validation set when developing the model. The model is trained on the training set, using any errors in identification to try and tune the model. Once complete, the model is tested on the test set to see how it does in classifying them before it undergoes another tuning iteration. The model that best classifies the validation set is kept for use in the project.

To make better use of the limited dataset, the team used a "k-fold" approach when training. This operates in much the same way as described above. However, the dataset is split into k (the team used five) groups where k models are trained in parallel using different combinations of the groups for training and validation and keeping the best performing model on it's dataset. The dataset is loaded into *emotionClassifier* which processes victim photos as mentioned previously. The most frequently appearing emotion in the batch of images is labelled as the victim's emotional state and passed on as it gets published.

When the subscriber in the team's code receives the determined emotion from *emotionClassifier*, it is recorded and the robot switches from its exploration state to the victim response state to react accordingly. The code for handling emotions is contained across the *emotionHandling.cpp* and *emotionHandling.h*, primarily by calling *emotionReaction*. The steps that follow are explained in section 3.2.4.

3.2.4 Victim Emotion Responses

The emotional responses the robot utilizes in interactions with victims is a behaviour-based controller implemented using a state-machine with substeps in the function *emotionReaction* of *emotionHandling.cpp*. As mentioned in the previous sections, the robot begins in an “explore” state while it navigates the environment. When it encounters a victim, the robot then enters the “emote” state to first determine the emotion of the victim. Once the emotion has been classified, the robot then checks the state machine to determine the emotional response behaviour that the robot will perform in the interaction.

Each victim's emotion has a different and unique robot emotional response. The robot is programmed to display one of the following emotions: pride, positively excited, embarrassment, surprise, discontent, anger, and resentment. Each emotion acts as a state and has a series of substeps that comprise the complete emotional behaviour. Each step is either a distinct motion or a set of multimedia, such as sounds and images, that display the emotion. The system progresses through all of the behaviours in the emotion state. Once the interaction is completed, the emotion state is cleared. The algorithm then switches the state back to “explore” until a new victim is located. Table 1 below outlines the victims’ emotion and the robot’s emotion reactions, including the behavioural substeps used in each emotion state. For example, when the robot classifies a victim’s emotion as neutral, the robot then interacts by displaying the emotion of pride and completes the actions in the response column.

In addition to using sound files, the team also included an image file to communicate evacuation information to the victims. This was implemented in the event that one or more of the victims was experiencing auditory issues, such as being hearing impaired or having outside noise interrupt the communication. The image file includes a neutral expressing face and text that explains the emergency and what to do. This image is constantly displayed while the robot explores and is only removed during interactions.

Using *travel* allows the team to produce the tailored motions required by emotional responses, such as reversing or rotating at different rates to produce a basic choreographed act.

Table 1: Table summarizing the emotion reactions the robot performs

Victim Emotion	Robot Emotion	Response
Neutral	Pride	1. Play sound/show picture 2. 360 degree slow spin 3. Play sound
Happiness	Positively Excited	1. Play sound/show picture 2. Move forward quickly 3. Quick spin 4. Play sound
Fear	Embarrassment	1. Play sound/show picture 2. Move backwards slowly 3. Turn one direction slowly, slightly (like being shy) 4. Play sound 5. Turn back slowly
Surprise	Surprise	1. Show picture (no sound) 2. Move backwards quickly 3. Play sound 4. Slowly move forwards (less distance) 5. Play sound
Disgust	Discontent	1. Play sound/show picture 2. Move back slowly a bit 3. slowly turn one way 4. Play sound
Sadness	Anger	1. Play sound/show picture 2. Move forward a small distance quickly (threateningly) 3. Play sound 4. Move forward again quickly
Anger	Resentment	1. Play sound/show picture 2. Turn quickly 180 degrees 3. Move straight short distance 4. Turn back 100 degrees (as if to look over its shoulder) 5. Play sound

4.0 Future Recommendations

Given additional time, the team would implement changes to further improve the robot's emotional reactions and emotional classification, and address issues with exploration. One such change would be to utilize other forms of communication when interacting with victims, such as replacing the static images and voice lines with a video of a human rescuer communicating evacuation instructions. Using a video for each emotional response would allow the team to include other elements of emotion not easily portrayed by the Turtlebot. For example, a video of a human rescuer includes human motion/gestures, body language and posture; all aspects of communication not currently captured in the robot's emotional responses. With video, these non-verbal cues are integrated directly with the warnings and instructions communicated by the robot, and help to express the severity of the disaster situation more completely. Time was a limiting factor in the preparation of media for the emotional response. In addition, given that the robot used for the contest is simulated, video files would require more resources to store and run. With more time to test and develop, the team would add a video element to their emotional responses.

Improving the accuracy and robustness of the *emotionClassifier* CNN is another area to which the team would direct future development efforts. Currently, the team aims to mitigate overfitting by implementing dropout in the hidden layers of the CNN. To further reduce overfitting, and also to provide the classifier with a larger set of data to train on, the team could implement data augmentation during the training process. This would entail generating new images in the training set via the application of rotations, zooms, skews, colour filters, and random noise to existing images. This approach would result in a much larger training set, resulting in a more generalized network with increased robustness.

An issue the team faced when exploring the environment was the *explore_lite* package struggling to select a goal frontier in situations where the robot was equidistant from all viable frontiers. This behaviour was most common when the robot was initialized in the middle of an open area, where all walls were outside the range of the frontiers. The team did include a "spin" sub-behaviour that rotated the robot 360° in situations where the robot did not move for an extended period of time. However, this did not eliminate the unwanted idle behaviour in all situations. There are several related solutions to this problem the team could implement as future work. Developing a method to more quickly identify when the robot is struggling to select a frontier would make the overall exploration process more robust, with fewer long pauses. This would reduce the time lost to the unwanted behaviour in situations where the "spin" sub-behaviour was sufficient to help the planner set a goal frontier. To handle situations where a spin alone is insufficient, the team would look to add a random walk component to the sub-behaviour. This additional random displacement would be particularly useful in cases where all frontiers are equidistant, as shown in the figure on the following page.

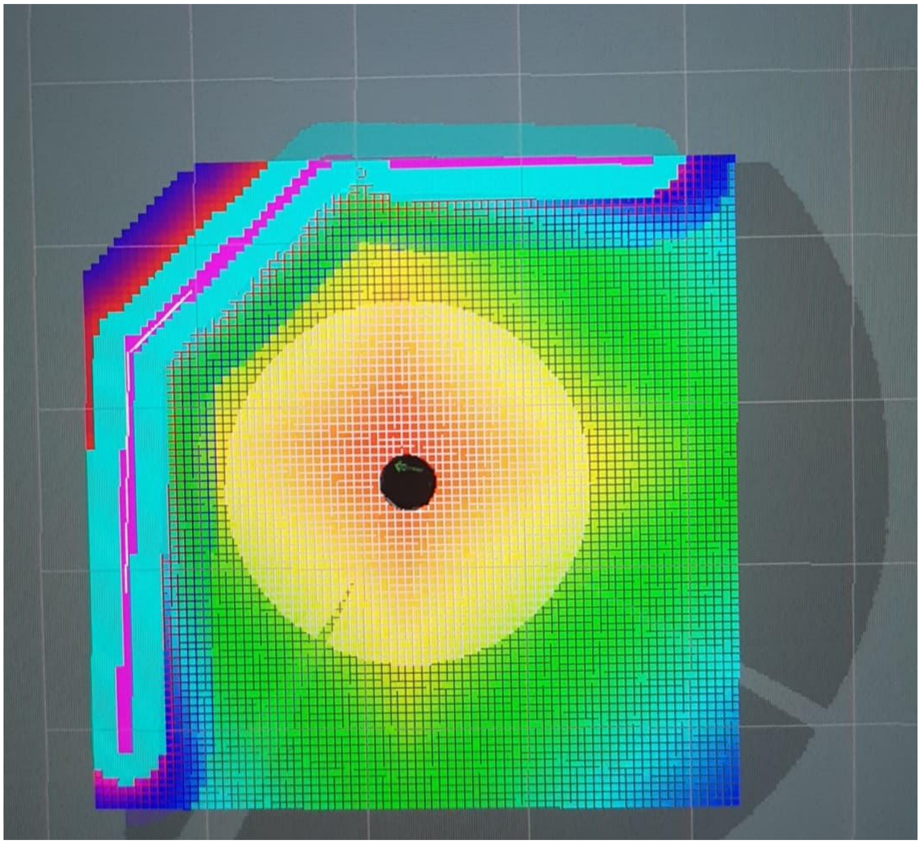

Figure 5: An example of equidistant frontiers causing failure of the planner

Random walk behaviour could be implemented in a similar manner to the first contest; a random rotation is selected, and then the robot translates in a fixed direction. The team could tune the number of random steps taken before reinitializing the *explore_lite* package to ensure the robot could decide on a frontier for exploration. If the issue persisted, the team could also look to dynamically adjust the *maxUrange* parameter in *gmapping*. This parameter governs the maximum range at which *gmapping* will record objects and add them to the map. Increasing the vision range in situations where frontiers are equidistant from the robot would allow for distant boundaries (e.g. walls) to be detected, helping the planner decide on a viable frontier.

5.0 Contribution Table

Contributions noted by value: 1 - small amount, 3 - majority, blank for none.

Table X: Contribution Table

Section	**Savo Bajic**	**Maximilian Glidden**	**Catherine Kucaba**
1.0			3
2.0	2	1	3
3.1		3	
3.2	3	2	1
4.0	1	3	1
Robot Code	3	2	2

6.0 Appendices

Appendix A: contest3.cpp

```
#include <ros/ros.h>
#include <ros/package.h>
#include "explore.h"
#include "emotionHandling.h"
#include "movement.h"
#include <time.h>

// If you cannot find the sound play library try the following command.
// sudo apt install ros-kinetic-sound-play
#include <sound_play/sound_play.h>
#include <ros/console.h>

//#define EMOTION_BYPASS // Used to skip emotion reactions in testing
(comment out before proper runs)

int main(int argc, char** argv) {
    // Monitor time elapsed
    time_t startTime = time(NULL);
    time_t secondsElapsed = 0;

    // Setup ROS.
    ros::init(argc, argv, "contest3");
    ros::NodeHandle n;

    // Class to handle sounds.
    sound_play::SoundClient sc;

    // Emotion node
    ros::Subscriber emotionSub = n.subscribe("/detected_emotion", 1,
&emotionCallback);
    clearEmotionState();

    // Manual motion setup (used for reactions and recovering exploration)
    ros::Subscriber odom = n.subscribe("odom", 1, &odomCallback);
    ros::Publisher vel_pub;
    geometry_msgs::Twist vel;
```

```
    bool spinInPlace = true; // Is rover supposed to be spinning in place
    bool manualOverride = spinInPlace; // Are we manually overriding the
bot's motion
    // Initialised to true since we want the robot to always start with a
scan

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

    ROS_WARN("STARTING MAIN LOOP!\n");
    while(ros::ok()) {
        ros::spinOnce();
        static bool overridingPrev = false; // Store overriding motion
status on previous step

        // Check for emotions
        if (readEmotion() >= 0) {
#ifdef EMOTION_BYPASS
            ROS_WARN("BYPASSING EMOTION CODE");
            victimsEncountered++; // Increment victim count as though they
were handled
            clearEmotionState();
#else
            // Sieze manual control of motion for motional reactions
            if (manualOverride == false) {
                explore.stop();
                manualOverride = true;
            }

            // Handle new emotion
            emotionReaction(sc);

            // Are we done with the remotion state?
            if (readEmotion() < 0) {
                // Release control (return to exploring)
                manualOverride = false;
                ROS_INFO("Done emotional reaction");
            }
#endif // EMOTION_BYPASS
        }
```

```
        else {
            // We're currently exploring

            if (spinInPlace) {
                // Manually spin in a circle
                manualOverride = true;
                spinInPlace = ((travel(0, 0, M_PI * 2, FAST_SPIN)) ==
false);

                // Check if we're done
                if (spinInPlace == false) {
                    ROS_INFO("Done spinning in the spot.");
                    manualOverride = false; // Release control
                }
            }
            else {
                // Check if we've moved and if not spin in place
                if (checkIfMoved(explore) == false) {
                    ROS_WARN("Robot detected as stationary. Initiating spin
move.");
                    spinInPlace = true;
                }
                else {
                    // Carry on exploring hero
                }
            }
        }

        // Are we manually controlling robot motion
        if (manualOverride) {
            // Setup control if we just siezed manual control
            if (overridingPrev == false) {
                ROS_WARN("Taking manual control of motion.");
                vel_pub =
n.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);
            }

            // Set motions needed
            vel.angular.z = angular;
            vel.linear.x = linear;
```

```
            vel_pub.publish(vel);
        }
        // Check if we have just released control this loop iteration
        if ((overridingPrev) && (manualOverride == false)) {
            travel(0, 0, 0, 0); // Reset motion code
            vel_pub.shutdown(); // Stop override when no longer needed
            ROS_WARN("Releasing manual control of motion, exploring
again.");
            explore.start(); // Explore again once control is released
        }

        overridingPrev = manualOverride; // Record override state for
reference next loop
        secondsElapsed = time(NULL) - startTime;
        ros::Duration(0.01).sleep();

        // Check if we're done reacting to all expected victims
        if (victimsEncountered == victimsExpected) {
            ROS_WARN("Interacted with all %d expected victims. Ending
search.", victimsExpected);
            break;
        }
        if (secondsElapsed > 1200) {
            ROS_FATAL("Ran out of time, 20minutes have passed");
            break;
        }
    }

    // End exploration
    explore.stop();

    // Output closing messages
    std::cout << std::endl; // Seperate from previous messages
    secondsElapsed = time(NULL) - startTime;
    ROS_INFO("Program execution took %d seconds to complete.",
int(secondsElapsed));
    ROS_FATAL("TERMINATING PROGRAM");
    return 0;
}
```

Appendix B: costmap_client.h

```
/*********************************************************************
*
* Software License Agreement (BSD License)
*
*  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.
*
*********************************************************************/
```

```
#ifndef COSTMAP_CLIENT_
#define COSTMAP_CLIENT_

#include <costmap_2d/costmap_2d.h>
#include <geometry_msgs/Pose.h>
#include <map_msgs/OccupancyGridUpdate.h>
#include <nav_msgs/OccupancyGrid.h>
#include <ros/ros.h>
#include <tf/tf.h>
#include <tf/transform_listener.h>

namespace explore
{
class Costmap2DClient
{
public:
 /**
  * @brief Contructs client and start listening
  * @details Constructor will block until first map update is received and
  * map is ready to use, also will block before trasformation
  * robot_base_frame <-> global_frame is available.
  *
  * @param param_nh node hadle to retrieve parameters from
  * @param subscription_nh node hadle where topics will be subscribed
  * @param tf_listener Will be used for transformation of robot pose.
  */
 Costmap2DClient(ros::NodeHandle& param_nh, ros::NodeHandle&
subscription_nh,
                 const tf::TransformListener* tf_listener);
 /**
  * @brief Get the pose of the robot in the global frame of the costmap
  * @return pose of the robot in the global frame of the costmap
  */
 geometry_msgs::Pose getRobotPose() const;

 /**
  * @brief Return a pointer to the "master" costmap which receives updates
from
  * all the layers.
  *
```

```
  * This pointer will stay the same for the lifetime of Costmap2DClient
object.
  */
 costmap_2d::Costmap2D* getCostmap()
 {
   return &costmap_;
 }

 /**
  * @brief Return a pointer to the "master" costmap which receives updates
from
  * all the layers.
  *
  * This pointer will stay the same for the lifetime of Costmap2DClient
object.
  */
 const costmap_2d::Costmap2D* getCostmap() const
 {
   return &costmap_;
 }

 /**
  * @brief  Returns the global frame of the costmap
  * @return The global frame of the costmap
  */
 const std::string& getGlobalFrameID() const
 {
   return global_frame_;
 }

 /**
  * @brief  Returns the local frame of the costmap
  * @return The local frame of the costmap
  */
 const std::string& getBaseFrameID() const
 {
   return robot_base_frame_;
 }

protected:
```

```
 void updateFullMap(const nav_msgs::OccupancyGrid::ConstPtr& msg);
 void updatePartialMap(const map_msgs::OccupancyGridUpdate::ConstPtr&
msg);

 costmap_2d::Costmap2D costmap_;

 const tf::TransformListener* const tf_;  ///< @brief Used for
transforming
                                          /// point clouds
 std::string global_frame_;       ///< @brief The global frame for the
costmap
 std::string robot_base_frame_;  ///< @brief The frame_id of the robot
base
 double transform_tolerance_;    ///< timeout before transform errors

private:
 // will be unsubscribed at destruction
 ros::Subscriber costmap_sub_;
 ros::Subscriber costmap_updates_sub_;
};

}  // namespace explore

#endif
```

Appendix C: costmap_client.cpp

```
/*********************************************************************
*
* Software License Agreement (BSD License)
*
*  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 <costmap_client.h>

#include <functional>
#include <mutex>
#include <string>

namespace explore
{
// static translation table to speed things up
std::array<unsigned char, 256> init_translation_table();
static const std::array<unsigned char, 256> cost_translation_table__ =
   init_translation_table();

Costmap2DClient::Costmap2DClient(ros::NodeHandle& param_nh,
                                 ros::NodeHandle& subscription_nh,
```

```
                                const tf::TransformListener* tf)
 : tf_(tf)
{
 std::string costmap_topic;
 std::string footprint_topic;
 std::string costmap_updates_topic;
 param_nh.param("costmap_topic", costmap_topic, std::string("costmap"));
 param_nh.param("costmap_updates_topic", costmap_updates_topic,
                std::string("costmap_updates"));
 param_nh.param("robot_base_frame", robot_base_frame_,
                std::string("base_link"));
 // transform tolerance is used for all tf transforms here
 param_nh.param("transform_tolerance", transform_tolerance_, 0.3);

 /* initialize costmap */
 costmap_sub_ = subscription_nh.subscribe<nav_msgs::OccupancyGrid>(
     costmap_topic, 1000,
     [this](const nav_msgs::OccupancyGrid::ConstPtr& msg) {
       updateFullMap(msg);
     });
 ROS_INFO("Waiting for costmap to become available, topic: %s",
          costmap_topic.c_str());
 auto costmap_msg = ros::topic::waitForMessage<nav_msgs::OccupancyGrid>(
     costmap_topic, subscription_nh);
 updateFullMap(costmap_msg);

 /* subscribe to map updates */
 costmap_updates_sub_ =
     subscription_nh.subscribe<map_msgs::OccupancyGridUpdate>(
         costmap_updates_topic, 1000,
         [this](const map_msgs::OccupancyGridUpdate::ConstPtr& msg) {
           updatePartialMap(msg);
         });

 /* resolve tf prefix for robot_base_frame */
 std::string tf_prefix = tf::getPrefixParam(param_nh);
 robot_base_frame_ = tf::resolve(tf_prefix, robot_base_frame_);

 // we need to make sure that the transform between the robot base frame
and
```

```
 // the global frame is available
 /* tf transform is necessary for getRobotPose */
 ros::Time last_error = ros::Time::now();
 std::string tf_error;
 while (ros::ok() &&
        !tf_->waitForTransform(global_frame_, robot_base_frame_,
ros::Time(),
                               ros::Duration(0.1), ros::Duration(0.01),
                               &tf_error)) {
   ros::spinOnce();
   if (last_error + ros::Duration(5.0) < ros::Time::now()) {
     ROS_WARN(
         "Timed out waiting for transform from %s to %s to become
available "
         "before subscribing to costmap, tf error: %s",
         robot_base_frame_.c_str(), global_frame_.c_str(),
tf_error.c_str());
     last_error = ros::Time::now();
   }
   // The error string will accumulate and errors will typically be the
same,
   // so the last
   // will do for the warning above. Reset the string here to avoid
   // accumulation.
   tf_error.clear();
 }
}

void Costmap2DClient::updateFullMap(const
nav_msgs::OccupancyGrid::ConstPtr& msg)
{
 global_frame_ = msg->header.frame_id;

 unsigned int size_in_cells_x = msg->info.width;
 unsigned int size_in_cells_y = msg->info.height;
 double resolution = msg->info.resolution;
 double origin_x = msg->info.origin.position.x;
 double origin_y = msg->info.origin.position.y;

 ROS_DEBUG("received full new map, resizing to: %d, %d", size_in_cells_x,
```

```
           size_in_cells_y);
 costmap_.resizeMap(size_in_cells_x, size_in_cells_y, resolution,
origin_x,
                    origin_y);

 // lock as we are accessing raw underlying map
 auto* mutex = costmap_.getMutex();
 std::lock_guard<costmap_2d::Costmap2D::mutex_t> lock(*mutex);

 // fill map with data
 unsigned char* costmap_data = costmap_.getCharMap();
 size_t costmap_size = costmap_.getSizeInCellsX() *
costmap_.getSizeInCellsY();
 ROS_DEBUG("full map update, %lu values", costmap_size);
 for (size_t i = 0; i < costmap_size && i < msg->data.size(); ++i) {
   unsigned char cell_cost = static_cast<unsigned char>(msg->data[i]);
   costmap_data[i] = cost_translation_table__[cell_cost];
 }
 ROS_DEBUG("map updated, written %lu values", costmap_size);
}

void Costmap2DClient::updatePartialMap(
   const map_msgs::OccupancyGridUpdate::ConstPtr& msg)
{
 ROS_DEBUG("received partial map update");
 global_frame_ = msg->header.frame_id;

 if (msg->x < 0 || msg->y < 0) {
   ROS_ERROR("negative coordinates, invalid update. x: %d, y: %d", msg->x,
             msg->y);
   return;
 }

 size_t x0 = static_cast<size_t>(msg->x);
 size_t y0 = static_cast<size_t>(msg->y);
 size_t xn = msg->width + x0;
 size_t yn = msg->height + y0;

 // lock as we are accessing raw underlying map
 auto* mutex = costmap_.getMutex();
```

```
 std::lock_guard<costmap_2d::Costmap2D::mutex_t> lock(*mutex);

 size_t costmap_xn = costmap_.getSizeInCellsX();
 size_t costmap_yn = costmap_.getSizeInCellsY();

 if (xn > costmap_xn || x0 > costmap_xn || yn > costmap_yn ||
     y0 > costmap_yn) {
   ROS_WARN("received update doesn't fully fit into existing map, "
            "only part will be copied. received: [%lu, %lu], [%lu, %lu] "
            "map is: [0, %lu], [0, %lu]",
            x0, xn, y0, yn, costmap_xn, costmap_yn);
 }

 // update map with data
 unsigned char* costmap_data = costmap_.getCharMap();
 size_t i = 0;
 for (size_t y = y0; y < yn && y < costmap_yn; ++y) {
   for (size_t x = x0; x < xn && x < costmap_xn; ++x) {
     size_t idx = costmap_.getIndex(x, y);
     unsigned char cell_cost = static_cast<unsigned char>(msg->data[i]);
     costmap_data[idx] = cost_translation_table__[cell_cost];
     ++i;
   }
 }
}

geometry_msgs::Pose Costmap2DClient::getRobotPose() const
{
 tf::Stamped<tf::Pose> global_pose;
 global_pose.setIdentity();
 tf::Stamped<tf::Pose> robot_pose;
 robot_pose.setIdentity();
 robot_pose.frame_id_ = robot_base_frame_;
 robot_pose.stamp_ = ros::Time();
 ros::Time current_time =
     ros::Time::now();  // save time for checking tf delay later

 // get the global pose of the robot
 try {
   tf_->transformPose(global_frame_, robot_pose, global_pose);
```

```
  } catch (tf::LookupException& ex) {
    ROS_ERROR_THROTTLE(1.0, "No Transform available Error looking up robot "
                            "pose: %s\n",
                       ex.what());
    return {};
  } catch (tf::ConnectivityException& ex) {
    ROS_ERROR_THROTTLE(1.0, "Connectivity Error looking up robot pose: %s\n",
                       ex.what());
    return {};
  } catch (tf::ExtrapolationException& ex) {
    ROS_ERROR_THROTTLE(1.0, "Extrapolation Error looking up robot pose: %s\n",
                       ex.what());
    return {};
  }
  // check global_pose timeout
  if (current_time.toSec() - global_pose.stamp_.toSec() >
      transform_tolerance_) {
    ROS_WARN_THROTTLE(1.0, "Costmap2DClient transform timeout. Current time: "
                           "%.4f, global_pose stamp: %.4f, tolerance: %.4f",
                      current_time.toSec(), global_pose.stamp_.toSec(),
                      transform_tolerance_);
    return {};
  }

  geometry_msgs::PoseStamped msg;
  tf::poseStampedTFToMsg(global_pose, msg);
  return msg.pose;
}

std::array<unsigned char, 256> init_translation_table()
{
  std::array<unsigned char, 256> cost_translation_table;

  // lineary mapped from [0..100] to [0..255]
  for (size_t i = 0; i < 256; ++i) {
```

```
    cost_translation_table[i] =
        static_cast<unsigned char>(1 + (251 * (i - 1)) / 97);
  }

  // special values:
  cost_translation_table[0] = 0;       // NO obstacle
  cost_translation_table[99] = 253;    // INSCRIBED obstacle
  cost_translation_table[100] = 254;   // LETHAL obstacle
  cost_translation_table[static_cast<unsigned char>(-1)] = 255;  // UNKNOWN

  return cost_translation_table;
}

}  // namespace explore
```

Appendix D: costmap_tools.h

```
#ifndef COSTMAP_TOOLS_H_
#define COSTMAP_TOOLS_H_

#include <costmap_2d/costmap_2d.h>
#include <geometry_msgs/PointStamped.h>
#include <geometry_msgs/PolygonStamped.h>
#include <ros/ros.h>

namespace frontier_exploration
{
/**
* @brief Determine 4-connected neighbourhood of an input cell, checking
for map
* edges
* @param idx input cell index
* @param costmap Reference to map data
* @return neighbour cell indexes
*/
std::vector<unsigned int> nhood4(unsigned int idx,
                                 const costmap_2d::Costmap2D& costmap)
{
```

```
 // get 4-connected neighbourhood indexes, check for edge of map
 std::vector<unsigned int> out;

 unsigned int size_x_ = costmap.getSizeInCellsX(),
              size_y_ = costmap.getSizeInCellsY();

 if (idx > size_x_ * size_y_ - 1) {
   ROS_WARN("Evaluating nhood for offmap point");
   return out;
 }

 if (idx % size_x_ > 0) {
   out.push_back(idx - 1);
 }
 if (idx % size_x_ < size_x_ - 1) {
   out.push_back(idx + 1);
 }
 if (idx >= size_x_) {
   out.push_back(idx - size_x_);
 }
 if (idx < size_x_ * (size_y_ - 1)) {
   out.push_back(idx + size_x_);
 }
 return out;
}

/**
* @brief Determine 8-connected neighbourhood of an input cell, checking
for map
* edges
* @param idx input cell index
* @param costmap Reference to map data
* @return neighbour cell indexes
*/
std::vector<unsigned int> nhood8(unsigned int idx,
                                 const costmap_2d::Costmap2D& costmap)
{
 // get 8-connected neighbourhood indexes, check for edge of map
 std::vector<unsigned int> out = nhood4(idx, costmap);
```

```
 unsigned int size_x_ = costmap.getSizeInCellsX(),
              size_y_ = costmap.getSizeInCellsY();

 if (idx > size_x_ * size_y_ - 1) {
   return out;
 }

 if (idx % size_x_ > 0 && idx >= size_x_) {
   out.push_back(idx - 1 - size_x_);
 }
 if (idx % size_x_ > 0 && idx < size_x_ * (size_y_ - 1)) {
   out.push_back(idx - 1 + size_x_);
 }
 if (idx % size_x_ < size_x_ - 1 && idx >= size_x_) {
   out.push_back(idx + 1 - size_x_);
 }
 if (idx % size_x_ < size_x_ - 1 && idx < size_x_ * (size_y_ - 1)) {
   out.push_back(idx + 1 + size_x_);
 }

 return out;
}

/**
* @brief Find nearest cell of a specified value
* @param result Index of located cell
* @param start Index initial cell to search from
* @param val Specified value to search for
* @param costmap Reference to map data
* @return True if a cell with the requested value was found
*/
bool nearestCell(unsigned int& result, unsigned int start, unsigned char
val,
                const costmap_2d::Costmap2D& costmap)
{
 const unsigned char* map = costmap.getCharMap();
 const unsigned int size_x = costmap.getSizeInCellsX(),
                    size_y = costmap.getSizeInCellsY();

 if (start >= size_x * size_y) {
```

```
    return false;
  }

  // initialize breadth first search
  std::queue<unsigned int> bfs;
  std::vector<bool> visited_flag(size_x * size_y, false);

  // push initial cell
  bfs.push(start);
  visited_flag[start] = true;

  // search for neighbouring cell matching value
  while (!bfs.empty()) {
    unsigned int idx = bfs.front();
    bfs.pop();

    // return if cell of correct value is found
    if (map[idx] == val) {
      result = idx;
      return true;
    }

    // iterate over all adjacent unvisited cells
    for (unsigned nbr : nhood8(idx, costmap)) {
      if (!visited_flag[nbr]) {
        bfs.push(nbr);
        visited_flag[nbr] = true;
      }
    }
  }

  return false;
}
}
#endif
```

Appendix E: emotionClassifier.py

```
#!/usr/bin/env python
from __future__ import print_function
import torch
import cv2
import torchvision
import numpy as np
import rospy
import roslib # Needed to import ModelStates
import argparse
from gazebo_msgs.msg import ModelStates
from geometry_msgs.msg import Pose
from std_msgs.msg import Int32
from emotionTrainingSample import EmotionClassificationNet
from mie443_contest3.msg import EmotionMsg
from mie443_contest3.msg import EmotionFaceMsg
import matplotlib.pyplot as plt

class EmotionDetector(object):

   def __init__(self, args):

       # Python 2.7 syntax.
       super(EmotionDetector, self).__init__()

       # Set hardware type
       device = torch.device('cpu')
       if args.gpu:
           device = torch.device('cuda:0')

       # Load your emotion detector.
       self.model = EmotionClassificationNet()
       self.model = self.model.to(device)
       self.model.load_state_dict(torch.load(args.model_file,
map_location=device))
       self.model.eval()

       # Visualize.
       self.vis = args.vis
       print('Setting up subscribers.')
```

```
        self.emotion_sub = rospy.Subscriber('/emotion_img', EmotionFaceMsg,
self.emotionsub)
        self.emotion_pub = rospy.Publisher('/detected_emotion', Int32,
queue_size=1)
        self.emotion_file = open('detectedVictim.txt', 'w')

    def showImBatch(self, imgs):
        img_grid = torchvision.utils.make_grid(imgs)
        self.matplotlib_imshow(img_grid)

    def matplotlib_imshow(self, img):
        img2 = img + 0.5     # uncenter
        npimg = img.numpy().transpose([1,2,0])
        cv2.imshow('Input', npimg)
        cv2.waitKey(0)

    def emotionsub(self, msg):
        with torch.no_grad():
            imgs = msg.data
            w = msg.width
            h = msg.height
            b = msg.batch
            imgs = torch.from_numpy(np.array(imgs)).view(b, 1, h,
w).float()
            if self.vis:
                print('Showing images.')
                self.showImBatch(imgs)
            emotions = self.model(imgs, True)

            # Emotion voting -- take the most often voted for emotion, ties
are broken arbitrarily.
            uniqueEmotions, counts = emotions.unique(sorted=True,
return_counts=True)
            print('uniqueEmotions:', uniqueEmotions)
            print('EmotionCounts:', counts)
            cnt_max, max_idx = counts.max(0)
            print(uniqueEmotions[max_idx])
            intmsg = Int32()
            intmsg.data = uniqueEmotions[max_idx].item()
            self.emotion_pub.publish(intmsg)
```

```
            self.emotion_file.write(str(uniqueEmotions[max_idx].item()))

    def logEmotionHistory(self):
        self.emotion_file.close()

# Parse the input arguments.
def getInputArgs():
    parser = argparse.ArgumentParser('MIE443_contest3 victim emotion
detector.')
    parser.add_argument('--gpu', dest='gpu',
default=torch.cuda.is_available(), type=bool, help='Use gpu for training')
    parser.add_argument('--model', dest='model_file',
default='mdl_best.pth', type=str, help='NN model to use for emotion
detection.')
    parser.add_argument('--vis', dest='vis', default=False,
action='store_true', help='Visualize the received images.')
    args = parser.parse_args()
    return args


if __name__ == "__main__":
    rospy.init_node('emotionDetector')
    args = getInputArgs()
    victim_locations = EmotionDetector(args)
    rospy.spin()
```

Appendix F: emotionHandling.h

```
#ifndef EMOTION_HANDLING_GROUP22_H
#define EMOTION_HANDLING_GROUP22_H

#include <ros/ros.h>
#include <ros/package.h>
#include <std_msgs/Int32.h> // Emotion messages
#include "movement.h"
#include <sound_play/sound_play.h>

#include <opencv2/core.hpp>
#include <cv.h>
#include <cv_bridge/cv_bridge.h>
```

```
#include "opencv2/highgui.hpp"

void emotionCallback (const std_msgs::Int32::ConstPtr& msg);
int32_t readEmotion(void); // Returns most recent emotion scanned
void clearEmotionState(void); // Clears emotion handler
void emotionReaction(sound_play::SoundClient &sc); // Handles emotion
reactions

void showImage(std::string fileLocation); // Show an image on screen

/*
   0=Angry
   1=Disgust
   2=Fear
   3=Happy
   4=Sad
   5=Surprise
   6=Neutral
*/
extern const char *emotionName[7]; // Emotion descriptions
extern int32_t emotionValue; // Most recent scanned emotion (-1 if no
emotion found/present)
extern int victimsEncountered; // Number of victims encountered thus far
extern const int victimsExpected; // Expected victim count

#endif
```

Appendix G: emotionHandling.cpp

```
#include "emotionHandling.h"

/*
   0=Angry
   1=Disgust
   2=Fear
   3=Happy
   4=Sad
   5=Surprise
```

```
   6=Neutral
*/

// Globals
int victimsEncountered = 0; // Number of victims encountered thus far
const int victimsExpected = 7; // Expected victim count for environment

const char *emotionName[7] = { "Angry", "Disgust", "Fear", "Happy", "Sad",
"Surprise", "Neutral"}; // Emotion descriptions

int32_t emotionValue = -1; // Stores most recent scanned emotion (-1 if no
emotion found/present)

const std::string pathToSounds = ros::package::getPath("mie443_contest3")
+ "/sounds/";
const std::string pathToImages = ros::package::getPath("mie443_contest3")
+ "/images/";

void emotionReaction(sound_play::SoundClient &soundPlayer) {
   static int emotionStep = 0;
   bool goToNextStep = false;

   const int soundPause = 5; // Length to sleep after each audio track

   // Handle appropriate emotion
   if (emotionValue == 0) {
       // Respond to anger with resentment

       if (emotionStep == 0) {
           soundPlayer.playWave(pathToSounds + "resentment1.wav");
           showImage(pathToImages + "resentment.png");
           ros::spinOnce();
           sleep(soundPause);
           emotionStep++; // Go to next step
       }
       else if (emotionStep == 1) {
           // Turn around
           goToNextStep = travel(0,0,M_PI,FAST_SPIN);
           if (goToNextStep) emotionStep++; // Go to next step once done
moving
```

```
        }
        else if (emotionStep == 2) {
            goToNextStep = travel(0.3,FAST_MOVE,0,0);
            if (goToNextStep) emotionStep++;
        }
        else if (emotionStep == 3) {
            // Turn around
            goToNextStep = travel(0,0,DEG2RAD(-100),-SLOW_SPIN);
            if (goToNextStep) emotionStep++;
        }
        else if (emotionStep == 4) {
            soundPlayer.playWave(pathToSounds + "resentment2.wav");
            ros::spinOnce();
            sleep(soundPause);
            clearEmotionState(); // Clear emotion reaction once done
reaction (movement)
        }

    }
    else if (emotionValue == 1) {
        // Respond to disgust with discontent

        if (emotionStep == 0) {
            soundPlayer.playWave(pathToSounds + "discontent1.wav");
            showImage(pathToImages + "discontent.png");
            ros::spinOnce();
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 1) {
            // Go backwards
            goToNextStep = travel(0.25,-SLOW_MOVE,0,0);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 2) {
            // Turn around
            goToNextStep = travel(0,0,M_PI,SLOW_SPIN);
            if (goToNextStep) emotionStep++;
        }
```

```
        else if (emotionStep == 3) {
            soundPlayer.playWave(pathToSounds + "discontent2.wav");
            ros::spinOnce();
            sleep(soundPause);
            clearEmotionState(); // Clear emotion reaction once done
reaction (movement)
        }
    }
    else if (emotionValue == 2) {
        // Respond to fear with embarassment

        if (emotionStep == 0) {
            soundPlayer.playWave(pathToSounds + "embarrassment1.wav");
            showImage(pathToImages + "embarrassment.png");
            ros::spinOnce();
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 1) {
            // Go backwards
            goToNextStep = travel(-0.4,-SLOW_MOVE,0,0);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 2) {
            // Turn around a bit
            goToNextStep = travel(0,0,DEG2RAD(45),-SLOW_SPIN);
            if (goToNextStep) emotionStep++;
        }
        if (emotionStep == 3) {
            soundPlayer.playWave(pathToSounds + "embarrassment2.wav");
            ros::spinOnce();
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 4) {
            // Turn around completely
            goToNextStep = travel(0,0,DEG2RAD(135),-SLOW_SPIN);
            if (goToNextStep) clearEmotionState();
        }
```

```
    }
    else if (emotionValue == 3) {
        // Respond to happiness with positive excitement

        if (emotionStep == 0) {
            soundPlayer.playWave(pathToSounds + "excited1.wav");
            showImage(pathToImages + "excited.png");
            ros::spinOnce();
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 1) {
            // Go Forwards
            goToNextStep = travel(0.2,FAST_MOVE,0,0);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 2) {
            // Turn around
            goToNextStep = travel(0,0,M_PI * 2,FAST_SPIN);
            if (goToNextStep) emotionStep++;
        }
        else if (emotionStep == 3) {
            soundPlayer.playWave(pathToSounds + "excited2.wav");
            ros::spinOnce();
            sleep(soundPause);
            clearEmotionState(); // Clear emotion reaction once done
reaction (movement)
        }

    }
    else if (emotionValue == 4) {
        // Respond to sadness with anger

        if (emotionStep == 0) {
            soundPlayer.playWave(pathToSounds + "anger1.wav");
            showImage(pathToImages + "anger.png");
            ros::spinOnce();
            sleep(soundPause);
```

```
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 1) {
            // Go Forwards
            goToNextStep = travel(0.2,FAST_MOVE,0,0);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 2) {
            soundPlayer.playWave(pathToSounds + "anger2.wav");
            ros::spinOnce();
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 3) {
            // Go Forwards
            goToNextStep = travel(0.2,FAST_MOVE,0,0);
            if (goToNextStep) clearEmotionState(); // Clear emotion
reaction once done reaction (movement)
        }
    }
    else if (emotionValue == 5) {
        // Respond to surprise with counter-surprise

        if (emotionStep == 0) {
            showImage(pathToImages + "surprise.png");
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 1) {
            // Go backwards
            goToNextStep = travel(-0.3,-FAST_MOVE,0,0);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 2) {
            soundPlayer.playWave(pathToSounds + "surprise1.wav");
            ros::spinOnce();
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
```

```
        else if (emotionStep == 3) {
            // Go Forwards
            goToNextStep = travel(0.2,SLOW_MOVE,0,0);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 4) {
            soundPlayer.playWave(pathToSounds + "surprise2.wav");
            ros::spinOnce();
            sleep(soundPause);
            clearEmotionState(); // Clear emotion reaction once done
reaction (movement)
        }

    }
    else if (emotionValue == 6) {
        // Respond to neutral with pride
        if (emotionStep == 0) {
            soundPlayer.playWave(pathToSounds + "pride1.wav");
            ros::spinOnce();
            showImage(pathToImages + "pride.png");
            sleep(soundPause);
            emotionStep++; // Go to next step
        }
        else if (emotionStep == 1) {
            // Spin around
            goToNextStep = travel(0,0,2*M_PI,SLOW_SPIN);
            if (goToNextStep) emotionStep++; // Go to next step once done
moving
        }
        else if (emotionStep == 2) {
            soundPlayer.playWave(pathToSounds + "pride2.wav");
            ros::spinOnce();
            sleep(soundPause);
            clearEmotionState(); // Clear emotion reaction once done
reaction (movement)
        }
    }

    // Check if we're done processing emotions
```

```
    if (emotionValue < 0) {
        emotionStep = 0;
        victimsEncountered++; // Increment victim count once done
interacting with them
    }
}


// Read in an image and show it on the screen
void showImage(std::string fileLocation) {

    try {
        cv::Mat img = cv::imread(fileLocation, cv::IMREAD_UNCHANGED);

        // Set transparent pixels to white
        for (int i = 0; i < img.size().width; i++) {
            for (int j = 0; j < img.size().width; j++) {

                // Is alpha less than half? (likely transparent)
                int index = 4 *i + j * 4 * img.size().width + 3;
                if (img.data[index] < 127 )  {
                    img.data[index] = 255;
                    img.data[index-1] = 255;
                    img.data[index-2] = 255;
                    img.data[index-3] = 255;
                }
            }
        }

        cv::resize(img, img,cv::Size(480,480)); // Resize to be visible on
the screen

        cv::imshow("Current Response", img);
        cv::waitKey(250); // Wait 50 ms so image appears
    }
    catch (...) {
        ROS_ERROR("Failure to read image at following location:\n\t%s",
fileLocation.c_str());
    }
```

```
}

// Act whenever we've detected an emotion present
// Currently just states and records the emotion
void emotionCallback(const std_msgs::Int32::ConstPtr& msg) {
   emotionValue = msg->data;

   ROS_INFO("Emotion %d detected (%s). Interacting with lifeform %d.",
emotionValue, emotionName[emotionValue], victimsEncountered + 1);
}

// Returns most recent emotion scanned
int32_t readEmotion(void) {
   //emotionDetected = false; // Record the scan
   return emotionValue;
}


// Clear the emotion handler state
void clearEmotionState(void) {
   showImage(pathToImages + "neutral.png"); // Normal image for victims
between reactions
   emotionValue = -1;
}
```

Appendix H: emotionTrainingSample.py

```
#!/usr/bin/env python3
import torch
import torch.nn as nn
import argparse
from tqdm.auto import tqdm
import matplotlib.pyplot as plt

# Parse the input arguments.
def getInputArgs():
```

```
    parser = argparse.ArgumentParser('Sample for training an emotion
classification model.')
    parser.add_argument('--gpu', dest='gpu',
default=torch.cuda.is_available(), type=bool, help='Use gpu for training')
    parser.add_argument('--nepoch', dest='nepoch', default=50, type=int,
help='Number of training epochs')
    parser.add_argument('--mdl', dest='mdl', default=None, type=str,
help='Model to load')
    parser.add_argument('--val', dest='val', action='store_true', help='Get
validation score')
    parser.add_argument('-f') # Added fo use in Google Colab
    args = parser.parse_args()
    return args

class EmotionClassificationNet(nn.Module):

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

            nn.Conv2d(64, 128, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            nn.Conv2d(128, 128, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            nn.Conv2d(128, 128, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
```

```
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            nn.Conv2d(128, 128, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.Dropout(0.25),
        )
        self.nn = nn.Sequential(
            nn.Linear(1152, 512),
            nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(128, 7)
        )

    def forward(self, x, test_mode=False):
        batch_size = x.shape[0]
        feats = self.cnn(x) # [128, 128, 3, 3]
        #print(feats.shape)
        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

def getDataset(splitData):
    import pathlib

    #pathToData = '/content/drive/MyDrive/2020-2021/MIE443 Robot/Contest
3/train_split.pth' #Cathy
    pathToData = '/content/drive/MyDrive/MIE443 Robot/Contest
3/train_split.pth' #Savo
```

```
    if pathlib.Path(pathToData).exists():
        if splitData:
            # This was used previously to divide data
            probs = torch.ones(train_imgs.shape[0]) * 0.3
            val_set_mask = torch.bernoulli(probs).bool()
            val_imgs = train_imgs[val_set_mask]
            val_labels = train_labels[val_set_mask]
            train_imgs = train_imgs[~val_set_mask]
            train_labels = train_labels[~val_set_mask]
            return (train_imgs, train_labels), (val_imgs, val_labels)
        else:
            train_imgs, train_labels = torch.load(pathToData)
            return train_imgs, train_labels
    else:
        print('The provided dataset does not exist!')
        exit(0)

def getDataloader(args):
    train, val = getDataset(args)
    train_dataset = torch.utils.data.TensorDataset(*train)
    val_dataset = torch.utils.data.TensorDataset(*val)

    # Due to class imbalance introduce a weighted random sampler to select
rare classes more often.
    batch_size = 128
    weights_label = train[1].unique(return_counts=True,
sorted=True)[1].float().reciprocal()
    weights = torch.zeros_like(train[1], dtype=torch.float)
    for idx, label in enumerate(train[1]):
        weights[idx] = weights_label[label]
    sampler = torch.utils.data.sampler.WeightedRandomSampler(weights,
len(weights))

    # Create the dataloaders for the different datasets.
    train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size,
                                               num_workers=2, sampler=sampler)
    val_loader = torch.utils.data.DataLoader(val_dataset,
batch_size=batch_size,
                                               shuffle=False, num_workers=2)
```

```
    return train_loader, val_loader

def train_loop(mdl, loss_fn, optim, dl, device):
    pbar = tqdm(dynamic_ncols=True, total=int(len(dl)))
    n_batch_loss = 50
    running_loss = 0
    for nex, ex in enumerate(dl):
        ims, labels, = ex
        ims = ims.to(device)
        labels = labels.to(device)

        # Optimization.
        optim.zero_grad()
        outs = mdl(ims)
        loss = loss_fn(outs, labels)
        loss.backward(loss)
        optim.step()

        # Statistics
        running_loss += loss.item()
        nex += 1
        if nex % n_batch_loss == 0:
            status = 'Loss: %.4f '%(loss / n_batch_loss)
            running_loss = 0
            pbar.set_description(status)
        pbar.update(1)
    pbar.close()
    return mdl

def calc_acc(mdl, dl, dl_type, device):
    with torch.no_grad():
        pbar = tqdm(dynamic_ncols=True, total=int(len(dl)))
        total = 0
        ncorrect = 0
        for nex, ex in enumerate(dl):
            ims, labels, = ex
            ims = ims.to(device)
            labels = labels.to(device)
            predicted = mdl(ims, True)
            total += labels.size(0)
```

```
            ncorrect += (predicted == labels).sum().item()
            status = '%s ACC: %.4f '%(dl_type, float(ncorrect) / total)
            pbar.set_description(status)
            pbar.update(1)
    pbar.close()
    return float(ncorrect) / total

def foldMaker(sizeOfData, numFolds):
    # Initialize
    width = int(sizeOfData / numFolds)
    folds = []

    # Record the indecies to use for each fold (slice)
    for sliceNum in range(numFolds):
        trainIDs = set(range(sizeOfData))
        validationIDs = set(range(sliceNum * width, (sliceNum + 1) *
width))
        trainIDs = trainIDs - validationIDs # Remove validation entries

        folds.append((list(trainIDs), list(validationIDs)))

    return folds

if __name__ == "__main__":
    args = getInputArgs()

    mdl = EmotionClassificationNet()
    ce_loss = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(mdl.parameters()) #, lr=0.005) # Learning
rate adjusted here

    if args.gpu:
        device = torch.device('cuda:0')
    else:
        device = torch.device('cpu')

    if args.mdl is not None:
        mdl.load_state_dict(torch.load(args.mdl))
    mdl = mdl.to(device)
```

```
    if args.val:
        # Doing a validation run
        print('Val ACC loop')
        _, val_dl = getDataloader(splitData = 1)
        mdl.train(False)
        val_acc = calc_acc(mdl, val_dl, 'val', device)
        print('VAL ACC: ', val_acc)
    else:
        # Training loop.

        # Read in data set and shuffle it
        dataImage, dataLabel = getDataset(splitData = 0)
        indecies = torch.randperm(dataImage.size()[0])
        dataImage = dataImage[indecies]
        dataLabel = dataLabel[indecies]

        # Prepare the folds
        k = 5
        kFolds = foldMaker(dataImage.size()[0], k)

        # Iterating through folds
        bestOverall = 0
        foldAccuracy = [0] * k

        for foldCount, (trainIDs, validationIDs) in enumerate(kFolds):
            print("Fold " + str(foldCount + 1) + " of " + str(k))

            # Prepare datasets for this fold
            trainingSet =
torch.utils.data.TensorDataset(dataImage[trainIDs], dataLabel[trainIDs])
            validationSet =
torch.utils.data.TensorDataset(dataImage[validationIDs],
dataLabel[validationIDs])

            # Setup dataloader for the fold (code taken from data loader
function)
            batch_size = 128
            weights_label = dataLabel[trainIDs].unique(return_counts=True,
sorted=True)[1].float().reciprocal()
```

```
            weights = torch.zeros_like(dataLabel[trainIDs],
dtype=torch.float)
            for idx, label in enumerate(dataLabel[trainIDs]):
                weights[idx] = weights_label[label]
            sampler =
torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights))

            # Create the dataloaders for the different datasets.
            train_dl = torch.utils.data.DataLoader(trainingSet,
batch_size=batch_size,
                                                   num_workers=2,
sampler=sampler)
            val_dl = torch.utils.data.DataLoader(validationSet,
batch_size=batch_size,
                                                   shuffle=False,
num_workers=2)

            # Go through all epochs
            for epoch in range(args.nepoch):
                loopMessage = 'Train loop %d of %d for fold %d of
%d.'%(epoch + 1, args.nepoch, foldCount + 1, k)
                print(loopMessage)

                # Train model
                mdl.train(True)
                mdl = train_loop(mdl, ce_loss, optimizer, train_dl, device)

                # Validate model
                mdl.train(False)
                #train_acc = calc_acc(mdl, train_dl, 'train', device) #
Check model to its training data
                val_acc = calc_acc(mdl, val_dl, 'Validation', device) #
Validate run with seperate (validation) data

                # See if this is best in the fold
                if val_acc > foldAccuracy[foldCount]:
                    foldAccuracy[foldCount] = val_acc

                # Update if best model overall
                if val_acc > bestOverall:
```

```
                    bestOverall = val_acc
                    torch.save(mdl.state_dict(), 'mdl_best.pth')

        # Print out summary of training
        print('Done training the model!')
        outputReview = 'Best validation score: %.2f | Average across folds: 
%.2f'%(bestOverall * 100.0, sum(foldAccuracy) * 100.0 / k)
        print(outputReview)
        print('Peak validation accuracy for each fold')
        for count, acc in enumerate(foldAccuracy):
            temp = '\tFold %d - %.2f %%'%(count + 1, acc*100.0)
            print(temp)
```

Appendix I: explore.h

```
/*********************************************************************
*
* 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.
*
*********************************************************************/
#ifndef NAV_EXPLORE_H_
#define NAV_EXPLORE_H_

#include <memory>
#include <mutex>
#include <string>
#include <vector>

#include <actionlib/client/simple_action_client.h>
#include <geometry_msgs/PoseStamped.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <ros/ros.h>
#include <visualization_msgs/MarkerArray.h>

#include <costmap_client.h>
#include <frontier_search.h>

namespace explore
{
/**
* @class Explore
* @brief A class adhering to the robot_actions::Action interface that
moves the
* robot base to explore its environment.
*/
class Explore
{
public:
 Explore();
```

```
 ~Explore();

 void start();
 void stop();

 std::vector<geometry_msgs::Point> frontier_blacklist_;

private:
 /**
  * @brief  Make a global plan
  */
 void makePlan();

 /**
  * @brief  Publish a frontiers as markers
  */
 void visualizeFrontiers(
     const std::vector<frontier_exploration::Frontier>& frontiers);

 void reachedGoal(const actionlib::SimpleClientGoalState& status,
                  const move_base_msgs::MoveBaseResultConstPtr& result,
                  const geometry_msgs::Point& frontier_goal);

 bool goalOnBlacklist(const geometry_msgs::Point& goal);

 ros::NodeHandle private_nh_;
 ros::NodeHandle relative_nh_;
 ros::Publisher marker_array_publisher_;
 tf::TransformListener tf_listener_;

 Costmap2DClient costmap_client_;
 actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
     move_base_client_;
 frontier_exploration::FrontierSearch search_;
 ros::Timer exploring_timer_;
 ros::Timer oneshot_;

  geometry_msgs::Point prev_goal_;
 double prev_distance_;
 ros::Time last_progress_;
```

```
 size_t last_markers_count_;

 // parameters
 double planner_frequency_;
 double potential_scale_, orientation_scale_, gain_scale_;
 ros::Duration progress_timeout_;
 bool visualize_;
};
}

#endif
```

Appendix J: explore.cpp

```
/*********************************************************************
*
* 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()) {
   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);
    ROS_DEBUG("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);
      });
}

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 K: frontier_search.h

```
#ifndef FRONTIER_SEARCH_H_
#define FRONTIER_SEARCH_H_

#include <costmap_2d/costmap_2d.h>

namespace frontier_exploration
{
/**
* @brief Represents a frontier
*
*/
struct Frontier {
 std::uint32_t size;
 double min_distance;
 double cost;
 geometry_msgs::Point initial;
 geometry_msgs::Point centroid;
 geometry_msgs::Point middle;
 std::vector<geometry_msgs::Point> points;
};

/**
* @brief Thread-safe implementation of a frontier-search task for an input
* costmap.
*/
class FrontierSearch
{
```

```
public:
 FrontierSearch()
 {
 }

 /**
  * @brief Constructor for search task
  * @param costmap Reference to costmap data to search.
  */
 FrontierSearch(costmap_2d::Costmap2D* costmap, double potential_scale,
               double gain_scale, double min_frontier_size);

 /**
  * @brief Runs search implementation, outward from the start position
  * @param position Initial position to search from
  * @return List of frontiers, if any
  */
 std::vector<Frontier> searchFrom(geometry_msgs::Point position);

protected:
 /**
  * @brief Starting from an initial cell, build a frontier from valid
adjacent
  * cells
  * @param initial_cell Index of cell to start frontier building
  * @param reference Reference index to calculate position from
  * @param frontier_flag Flag vector indicating which cells are already
marked
  * as frontiers
  * @return new frontier
  */
 Frontier buildNewFrontier(unsigned int initial_cell, unsigned int
reference,
                           std::vector<bool>& frontier_flag);

 /**
  * @brief isNewFrontierCell Evaluate if candidate cell is a valid
candidate
  * for a new frontier.
  * @param idx Index of candidate cell
```

```
  * @param frontier_flag Flag vector indicating which cells are already
marked
  * as frontiers
  * @return true if the cell is frontier cell
  */
 bool isNewFrontierCell(unsigned int idx,
                        const std::vector<bool>& frontier_flag);

 /**
  * @brief computes frontier cost
  * @details cost function is defined by potential_scale and gain_scale
  *
  * @param frontier frontier for which compute the cost
  * @return cost of the frontier
  */
 double frontierCost(const Frontier& frontier);

private:
 costmap_2d::Costmap2D* costmap_;
 unsigned char* map_;
 unsigned int size_x_, size_y_;
 double potential_scale_, gain_scale_;
 double min_frontier_size_;
};
}
#endif
```

Appendix L: frontier_search.cpp

```
#include <frontier_search.h>

#include <mutex>

#include <costmap_2d/cost_values.h>
#include <costmap_2d/costmap_2d.h>
#include <geometry_msgs/Point.h>

#include <costmap_tools.h>
```

```
namespace frontier_exploration
{
using costmap_2d::LETHAL_OBSTACLE;
using costmap_2d::NO_INFORMATION;
using costmap_2d::FREE_SPACE;

FrontierSearch::FrontierSearch(costmap_2d::Costmap2D* costmap,
                               double potential_scale, double gain_scale,
                               double min_frontier_size)
 : costmap_(costmap)
 , potential_scale_(potential_scale)
 , gain_scale_(gain_scale)
 , min_frontier_size_(min_frontier_size)
{
}

std::vector<Frontier> FrontierSearch::searchFrom(geometry_msgs::Point
position)
{
 std::vector<Frontier> frontier_list;

 // Sanity check that robot is inside costmap bounds before searching
 unsigned int mx, my;
 if (!costmap_->worldToMap(position.x, position.y, mx, my)) {
   ROS_ERROR("Robot out of costmap bounds, cannot search for frontiers");
   return frontier_list;
 }

 // make sure map is consistent and locked for duration of search
 std::lock_guard<costmap_2d::Costmap2D::mutex_t>
lock(*(costmap_->getMutex()));

 map_ = costmap_->getCharMap();
 size_x_ = costmap_->getSizeInCellsX();
 size_y_ = costmap_->getSizeInCellsY();

 // initialize flag arrays to keep track of visited and frontier cells
 std::vector<bool> frontier_flag(size_x_ * size_y_, false);
 std::vector<bool> visited_flag(size_x_ * size_y_, false);
```

```
// initialize breadth first search
std::queue<unsigned int> bfs;

// find closest clear cell to start search
unsigned int clear, pos = costmap_->getIndex(mx, my);
if (nearestCell(clear, pos, FREE_SPACE, *costmap_)) {
  bfs.push(clear);
} else {
  bfs.push(pos);
  ROS_WARN("Could not find nearby clear cell to start search");
}
visited_flag[bfs.front()] = true;

while (!bfs.empty()) {
  unsigned int idx = bfs.front();
  bfs.pop();

  // iterate over 4-connected neighbourhood
  for (unsigned nbr : nhood4(idx, *costmap_)) {
    // add to queue all free, unvisited cells, use descending search in
case
    // initialized on non-free cell
    if (map_[nbr] <= map_[idx] && !visited_flag[nbr]) {
      visited_flag[nbr] = true;
      bfs.push(nbr);
      // check if cell is new frontier cell (unvisited, NO_INFORMATION,
free
      // neighbour)
    } else if (isNewFrontierCell(nbr, frontier_flag)) {
      frontier_flag[nbr] = true;
      Frontier new_frontier = buildNewFrontier(nbr, pos, frontier_flag);
      if (new_frontier.size * costmap_->getResolution() >=
          min_frontier_size_) {
        frontier_list.push_back(new_frontier);
      }
    }
  }
}
```

```
 // set costs of frontiers
 for (auto& frontier : frontier_list) {
   frontier.cost = frontierCost(frontier);
 }
 std::sort(
     frontier_list.begin(), frontier_list.end(),
     [](const Frontier& f1, const Frontier& f2) { return f1.cost <
f2.cost; });

 return frontier_list;
}

Frontier FrontierSearch::buildNewFrontier(unsigned int initial_cell,
                                          unsigned int reference,
                                          std::vector<bool>& frontier_flag)
{
 // initialize frontier structure
 Frontier output;
 output.centroid.x = 0;
 output.centroid.y = 0;
 output.size = 1;
 output.min_distance = std::numeric_limits<double>::infinity();

 // record initial contact point for frontier
 unsigned int ix, iy;
 costmap_->indexToCells(initial_cell, ix, iy);
 costmap_->mapToWorld(ix, iy, output.initial.x, output.initial.y);

 // push initial gridcell onto queue
 std::queue<unsigned int> bfs;
 bfs.push(initial_cell);

 // cache reference position in world coords
 unsigned int rx, ry;
 double reference_x, reference_y;
 costmap_->indexToCells(reference, rx, ry);
 costmap_->mapToWorld(rx, ry, reference_x, reference_y);

 while (!bfs.empty()) {
   unsigned int idx = bfs.front();
```

```
    bfs.pop();

    // try adding cells in 8-connected neighborhood to frontier
    for (unsigned int nbr : nhood8(idx, *costmap_)) {
      // check if neighbour is a potential frontier cell
      if (isNewFrontierCell(nbr, frontier_flag)) {
        // mark cell as frontier
        frontier_flag[nbr] = true;
        unsigned int mx, my;
        double wx, wy;
        costmap_->indexToCells(nbr, mx, my);
        costmap_->mapToWorld(mx, my, wx, wy);

        geometry_msgs::Point point;
        point.x = wx;
        point.y = wy;
        output.points.push_back(point);

        // update frontier size
        output.size++;

        // update centroid of frontier
        output.centroid.x += wx;
        output.centroid.y += wy;

        // determine frontier's distance from robot, going by closest
gridcell
        // to robot
        double distance = sqrt(pow((double(reference_x) - double(wx)), 2.0)
+
                               pow((double(reference_y) - double(wy)),
2.0));
        if (distance < output.min_distance) {
          output.min_distance = distance;
          output.middle.x = wx;
          output.middle.y = wy;
        }

        // add to queue for breadth first search
        bfs.push(nbr);
```

```
      }
    }
  }

  // average out frontier centroid
  output.centroid.x /= output.size;
  output.centroid.y /= output.size;
  return output;
}

bool FrontierSearch::isNewFrontierCell(unsigned int idx,
                                       const std::vector<bool>&
frontier_flag)
{
  // check that cell is unknown and not already marked as frontier
  if (map_[idx] != NO_INFORMATION || frontier_flag[idx]) {
    return false;
  }

  // frontier cells should have at least one cell in 4-connected
neighbourhood
  // that is free
  for (unsigned int nbr : nhood4(idx, *costmap_)) {
    if (map_[nbr] == FREE_SPACE) {
      return true;
    }
  }

  return false;
}

double FrontierSearch::frontierCost(const Frontier& frontier)
{
  return (potential_scale_ * frontier.min_distance *
          costmap_->getResolution()) -
         (gain_scale_ * frontier.size * costmap_->getResolution());
}
}
```

Appendix M: movement.h

```
#ifndef TEAM_MOVEMENT_HEADER
#define TEAM_MOVEMENT_HEADER

#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <time.h>
#include "explore.h"

#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

// Odometery values
extern float posX, posY, yaw;

// Motion variables
extern float angular;
extern float linear;
extern const float SLOW_SPIN;
extern const float FAST_SPIN;
extern const float SLOW_MOVE;
extern const float FAST_MOVE;
extern const float MAX_LIN;
extern const float MAX_ROT;

void odomCallback (const nav_msgs::Odometry::ConstPtr& msg);
bool monitorMotion ();
void setMotion(double dist, double linSpeed, double rot, double rotSpeed);

bool checkIfMoved(explore::Explore &explorer); // Monitors if the robot is
moving

/** @name travel
  *  @brief Use this for controlled travel by repeatedly calling it.
  *  @param  dist Linear distance to travel (magnitude)
```

```
  *  @param  linSpeed Linear velocity (determines forwards or back)
  *  @param  rot Rotational displacement in radians (magnitude)
  *  @param  rotSpeed Rotational velocity rad/s (+ive is left)
  *  @return  If the movement is complete, true.
  *
  *  @note It knows when you call it the first time or change course, so
long as the parameters change.
  *  (Calling travel(1,1,0,0) and then travel(1,1,0,0) again after that is
completed will not register
  *  as a new "move", these need to be seperated by a unique call such as
travel(0,0,0,0) to repeat)
  */
bool travel(double dist, double linSpeed, double rot, double rotSpeed);

/** @name setHeading
* @brief Will align the robot optimally with some absolute heading.
* @param heading The absolute heading (rad) to aim for
* @param speed Magnitude of rotation (rad)
* @return If the movement is complete, true.
* */
bool setHeading(float heading, float speed);
#endif
```

Appendix N: movement.cpp

```
#include "movement.h"

// Movement values
float angular = 0.0; // Global
float linear = 0.0;  // Global

// Constants
const float SLOW_SPIN = (M_PI/12);
const float FAST_SPIN = (M_PI/6);
const float SLOW_MOVE = 0.1;
const float FAST_MOVE = 0.25;
const float MAX_LIN = 0.25;
const float MAX_ROT = (M_PI/6);

// Odometery values
```

```
float posX = 0.0, posY = 0.0, yaw = 0.0;

// File scoped globals
float distanceRemaining, prevX, prevY;
float rotationRemaining, prevYaw;
float rotMaintain, linMaintain;

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("Position: (%f, %f) Orientation: %f rad or %f degrees.",
posX, posY, yaw, RAD2DEG(yaw));
}

bool monitorMotion ()
{
   //Maintain speeds
   angular = rotMaintain;
   linear = linMaintain;

   // See how mauch distance has been traversed
   float dx = posX - prevX;
   float dy = posY - prevY;
   float displacement = sqrt((dx * dx) + (dy * dy));
   prevX = posX;
   prevY = posY;

   // Find remaining distance to travel
   distanceRemaining = distanceRemaining - displacement;
   if (distanceRemaining <= 0.0) linear = 0; // Stop

   // Monitor rotational displacement
   displacement = std::abs(yaw - prevYaw);
   // Check if displacement is too big (crossing from -pi to +pi)
   // Heading is kept in the range [-pi,+pi] so when turning over pi
   // e.g. -0.98PI to 0.98PI, we need this case
   if (displacement > M_PI) {
```

```
        displacement = (M_PI - abs(yaw)) + (M_PI - abs(prevYaw));
    }
    rotationRemaining = rotationRemaining - displacement;
    if (rotationRemaining <= 0.0) angular = 0; // Stop when we've rotated
as required
    prevYaw = yaw;

    rotMaintain = angular;
    linMaintain = linear;

    // Return true if motion completed (no speed)
    bool doneMotion = (angular == 0) && (linear == 0);

    if (doneMotion) ROS_DEBUG("Reached destination.");
    else {
        ROS_DEBUG("In motion: D:%.2f S:%.2f | A:%.0f S:%.0f",
distanceRemaining,
        linear, RAD2DEG(rotationRemaining), RAD2DEG(angular));
    }
    return doneMotion;
}

void setMotion(double dist, double linSpeed, double rot, double rotSpeed)
{
    // Record new travel requirements
    distanceRemaining = std::abs(dist);
    rotationRemaining = std::abs(rot);

    // Set start point
    prevX = posX;
    prevY = posY;
    prevYaw = yaw;

    // Need to change the global variable
    linear = linSpeed;
    angular = rotSpeed;

    // Records speeds to maintain
    rotMaintain = angular;
    linMaintain = linear;
```

```
    ROS_DEBUG("Set up motion: D:%.2f S:%.2f | A:%.0f S:%.0f",
distanceRemaining, linear,
        RAD2DEG(rotationRemaining), RAD2DEG(angular));
}

bool travel(double dist, double linSpeed, double rot, double rotSpeed)
{
    static double pd = 0, pls = 0, pr = 0, prs = 0; // Used to store
previous state of inputs

    // Check if this is a repeated call
    bool repeatedCall = false;
    if ((dist == pd) && (linSpeed == pls) && (rot == pr) && (rotSpeed ==
prs)) repeatedCall = true;
    pd = dist;
    pls = linSpeed;
    pr = rot;
    prs = rotSpeed;

    bool doneMotion = false;

    if (repeatedCall == true) {
        // Repeated call, so we're maintaining course
        doneMotion = monitorMotion();
    }
    else {
        // New call, set new course
        setMotion(dist, linSpeed, rot, rotSpeed);
    }

    return doneMotion;
}

bool setHeading(float heading, float speed)
{
    static float lastHeading = 0, lastSpeed = 0; // Used to monitor
parameters
    static float change = 0, rotVelocity = 0; // Internal parameters to
describe the change
```

```
    bool doneAlignment = false;

    if ((lastHeading == heading) && (lastSpeed == speed)) {
        // Parameters didn't change, repeat travel() call
        doneAlignment = travel(0, 0, change, rotVelocity);
    }
    else {
        // New parameters, therefore an initial call
        // Record parameters for future reference
        lastHeading = heading;
        lastSpeed = speed;

        // Determine change needed
        float changeForward, changeBackward, tempHeading;

        // Find forward displacement
        if (heading < yaw) tempHeading = heading + 2 * M_PI; // Loop around
        else tempHeading = heading;
        changeForward = tempHeading - yaw;

        // Find backwards displacement
        if (heading > yaw) tempHeading = heading - 2 * M_PI; // Loop back
        else tempHeading = heading;
        changeBackward = yaw - tempHeading;

        ROS_DEBUG("Going forward %.0f deg, backwards %.0f.",
RAD2DEG(changeForward), RAD2DEG(changeBackward));

        // Compare to see which direction is optimal
        if (changeBackward < changeForward) {
            // Going backwards
            rotVelocity = -speed;
            change = changeBackward;
            ROS_INFO("Aligning to heading %.0f deg, currently at %.0f.
Change of %.0f right.",
                RAD2DEG(heading), RAD2DEG(yaw), RAD2DEG(change));
        }
        else {
            // Going forwards
            rotVelocity = speed;
```

```
            change = changeForward;

            ROS_INFO("Aligning to heading %.0f deg, currently at %.0f.
Change of %.0f left.",
                RAD2DEG(heading), RAD2DEG(yaw), RAD2DEG(change));
        }



        // Start moving as needed
        doneAlignment = travel(0, 0, change, rotVelocity);
    }

    if (doneAlignment) ROS_INFO("Aligned with new heading.");
    return doneAlignment;
}

bool checkIfMoved(explore::Explore &explorer) {
    const float minMotion = 1.0;
    const int recordingStep = 1; // Time in seconds between location
recordings
    const int numRecordings = 10; // Number of recordings to store
    static time_t targetTime = time(NULL);

    static bool didItMove = true;
    static bool alreadyStuck = false;

    static int curRecording = 0; // Current point in buffer
    static float xRec[numRecordings], yRec[numRecordings];

    time_t presentTime = time(NULL);

    // Periodically check if robot moved
    if (targetTime < presentTime) {

        targetTime = presentTime + recordingStep; // Set next time

        // Record current position into buffer
        xRec[curRecording] = posX;
        yRec[curRecording] = posY;
```

```
// Find total distance traversed over the last set of steps
float travelDist = 0.0;

for (int i = (curRecording - 1); i != curRecording; i--) {
    float temp, dx, dy;
    if (i < 0) {
        // Handle loop around
        dx = xRec[numRecordings - 1] - xRec[0];
        dy = yRec[numRecordings - 1] - yRec[0];
        i = numRecordings; // Move i to end
    }
    else {
        dx = xRec[i] - xRec[i - 1];
        dy = yRec[i] - yRec[i - 1];
    }
    // Add incremental distance to total
    temp = dx * dx + dy * dy;
    travelDist += sqrt(temp);
}

// Increment to next buffer spot
curRecording++;

// If we've completely filled the buffer
if (curRecording == numRecordings) {
    curRecording = 0;
    ROS_INFO_ONCE("Done first pass of movement buffer");

    // Draw conclusion
    if (travelDist > minMotion) {
        didItMove = true;
        alreadyStuck = false;
    }
    else {
        didItMove = false;
        ROS_WARN("Robot appears to be stuck.");

        if (alreadyStuck) {
```

```
                   ROS_WARN("Already marked as stuck previously, clearing 
boundary blacklist.");
                   explorer.frontier_blacklist_.clear();
               }
               alreadyStuck = true;
           }
       }

       //ROS_INFO("Travelled %.2f, target %.2f (%d)", travelDist, 
minMotion, didItMove);
   }

   return didItMove;
}
```

Appendix O: victimLocator.py

```
#!/usr/bin/env python
from __future__ import print_function
import os
import torch
import rospy
import roslib # Needed to import ModelStates
import random
import argparse
import numpy as np
import pandas as pd
from geometry_msgs.msg import Pose
from gazebo_msgs.msg import ModelState
from gazebo_msgs.msg import ModelStates
from visualization_msgs.msg import Marker
from visualization_msgs.msg import MarkerArray
from mie443_contest3.msg import EmotionFaceMsg
#
# Parse the input arguments.
def getInputArgs():
   parser = argparse.ArgumentParser('MIE443_contest3 victim detector.')
   parser.add_argument('--victim_file', dest='victim_file', 
default='victims.csv', type=str, help='Locations of the victims in the 
environment')
```

```
    parser.add_argument('--emotion_file', dest='emotion_file',
default='train_split.pth', type=str, help='Emotion detection file')
    parser.add_argument('--n_face_samples', dest='n_face_samples',
default=10, type=int, help='Emotion detection file')
    args = parser.parse_args()
    return args

class VictimLocations(object):

    def __init__(self, args):
        #
        # Setup victims.
        victims = pd.read_csv(os.path.abspath(args.victim_file))
        victims.columns = victims.columns.str.strip()
        self.pub_victim_count = 0

        assert set(['pos_x', 'pos_y',
'radius']).issubset(set(victims.columns.astype(str).tolist()))
        self.results_file = open('detectedVictim.txt', 'w')
        self.victim_pose = victims.as_matrix(['pos_x', 'pos_y'])
        self.victim_det_rad = victims.as_matrix(['radius']).reshape(-1)
        self.publish_victims(self.victim_pose, self.victim_det_rad)
        #
        # Setup emotions.
        self.n_face_samples = args.n_face_samples
        self.emotion_lists, self.possible_emotions =
self.loadEmotion(args.emotion_file)
        random.shuffle(self.possible_emotions)
        rospy.on_shutdown(self.logVictimHistory)
        #
        # Subscribe to the location of the robot.
        self.gazebo_sub = rospy.Subscriber('/gazebo/model_states',
ModelStates, self.stateSub)
        self.emotion_pub = rospy.Publisher('/emotion_img', EmotionFaceMsg,
queue_size=1)
        while self.emotion_pub.get_num_connections() == 0:
            print('Waiting for connection for emotion pub.')
            rospy.sleep(0.5)

    def publish_victims(self, pose, rad):
```

```
        markerPub = rospy.Publisher('/gazebo/set_model_state', ModelState,
queue_size=10)
        while markerPub.get_num_connections() == 0:
            print('Waiting for connection for marker pub.')
            rospy.sleep(0.5)
        msg = ModelState()
        msg.reference_frame = 'world'
        for v_idx in range(pose.shape[0]):
            msg.model_name = 'box_contest3_clone_' + str(v_idx)
            msg.pose.position.x = pose[v_idx][0]
            msg.pose.position.y = pose[v_idx][1]
            msg.pose.position.z = -0.999
            msg.pose.orientation.x = -0.5
            msg.pose.orientation.y = -0.5
            msg.pose.orientation.z = -0.5
            msg.pose.orientation.w = 0.5
            markerPub.publish(msg)

    def loadEmotion(self, emotion_file_path):
        imgs, labels = torch.load(os.path.abspath(emotion_file_path))
        unique_emotions = labels.unique()
        emotion_imgs = {}
        for l in unique_emotions:
            emotion_mask = labels == l
            emotion_imgs[l.item()] = imgs[emotion_mask].numpy()
        return emotion_imgs, unique_emotions.numpy().tolist()

    def robotLocation(self, msg):
        assert isinstance(msg, Pose)
        return np.array([msg.position.x, msg.position.y])

    def pubEmotionSet(self, gt_emotion_idx):
        print('Publishing emotion images')
        emo_imgs = self.emotion_lists[gt_emotion_idx]
        selected_emo_imgs = emo_imgs[np.random.choice(emo_imgs.shape[0],
self.n_face_samples, replace=False), :]
        print(selected_emo_imgs.shape)
        emo_msg = EmotionFaceMsg()
        emo_msg.width = 48
        emo_msg.height = 48
```

```
        emo_msg.batch = self.n_face_samples
        emo_msg.data = selected_emo_imgs.reshape(-1,1,1)
        while self.emotion_pub.get_num_connections() == 0:
            print('Waiting for connection for emotion pub.')
            rospy.sleep(0.05)
        self.emotion_pub.publish(emo_msg)
        print('Published emotion images')

    def stateSub(self, msg):
        #
        # This publish command is unreliable, but needed to ensure the 
markers are in the ground and in place.
        if self.pub_victim_count < 200:
            self.publish_victims(self.victim_pose, self.victim_det_rad)
            self.pub_victim_count += 1
        for idx, name in enumerate(msg.name):
            if name == 'mobile_base':
                break
        else:
            return
        #
        # Check if the current state of the robot intersects with any 
victim.
        rpose = self.robotLocation(msg.pose[idx])
        vic_dist = np.sqrt(np.sum((rpose - self.victim_pose) ** 2, 
-1)).reshape(-1)
        vic_detected = vic_dist < self.victim_det_rad
        detected_poses = self.victim_pose[vic_detected]
        #
        # Clean up what has been detected.
        self.victim_det_rad = self.victim_det_rad[~vic_detected]
        self.victim_pose = self.victim_pose[~vic_detected]
        for pose in detected_poses:
            gt_emotion_idx = self.possible_emotions.pop(0)
            print('Sending emotion:', gt_emotion_idx)
            detected_victim = [pose, gt_emotion_idx]
            self.pubEmotionSet(gt_emotion_idx)
            self.results_file.write(str(detected_victim))

    def logVictimHistory(self):
```

```
        self.results_file.close()

if __name__ == "__main__":
    rospy.init_node('victimLocator')
    args = getInputArgs()
    victim_locations = VictimLocations(args)
    rospy.spin()
```

7.0 References

[1] *MIE443H1S: Contest 3: Finding and Interacting with Emotional People in an Unknown Environment*, University of Toronto, Toronto, ON, 2021, pp. 1-3.