MIE 443 Contest 3:

Follow Me Robot Companion

Yu-Tung Chen - 1005024910
Kevon Seechan - 1004941708
Diego Gomez - 1005139690
Miguel Gomez - 1005138628

April 13th, 2023

1.0 INTRODUCTION

1.1 Objectives

The purpose of this project is to program a turtlebot to act as a companion robot. The robot should be able to locate and follow a user while they move around an open environment. The robot should also be able to display 4 different emotions. As discussed in lecture, emotions can be generally divided into two parts namely: primary and secondary emotions. Primary emotions are considered to be reactive in nature and cause global changes in internal states, whereas secondary emotions are considered to be deliberative in nature and are typically triggered by deliberative subsystems. The team must take these definitions into account and ensure that of the 4 emotional displays, 2 are primary and 2 are secondary. Additionally, the emotional displays should be triggered by 4 unique emotional stimuli and, as part of the evaluation criteria, must be creative and engaging.

1.2 Requirements and Constraints

The robot will be given 8 minutes to follow its user to 4 different checkpoints. At each checkpoint, the robot must display an emotional response to a particular environmental stimuli. Two of these stimuli have been predetermined, they are:

1. Robot loses track of the person it is following
2. Robot cannot continue to track the person due to a static obstacle in its path. See the figure below for clarification.

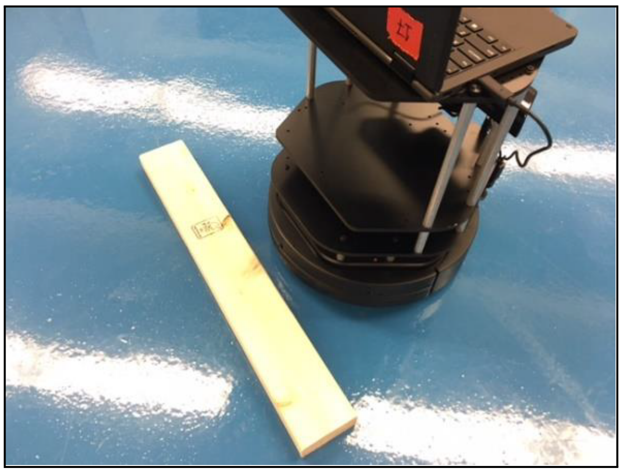

Figure 1: Static Obstacle in Robot's Path

The other two stimuli must be developed by the team. Additionally, the team must create original and unique motion and sound to display the emotional reactions without explicitly stating the emotion the robot is showing. For example a sound cannot be used to say "I am angry." Each display will be judged by a panel of 3 judges and be scored based on if they correctly determine the emotion and if they consider the display to be creative and complex.
Finally, all emotions must be chosen from the list seen in Appendix B.

2.0 STRATEGY

The flowchart below describes the general flow of the algorithm.

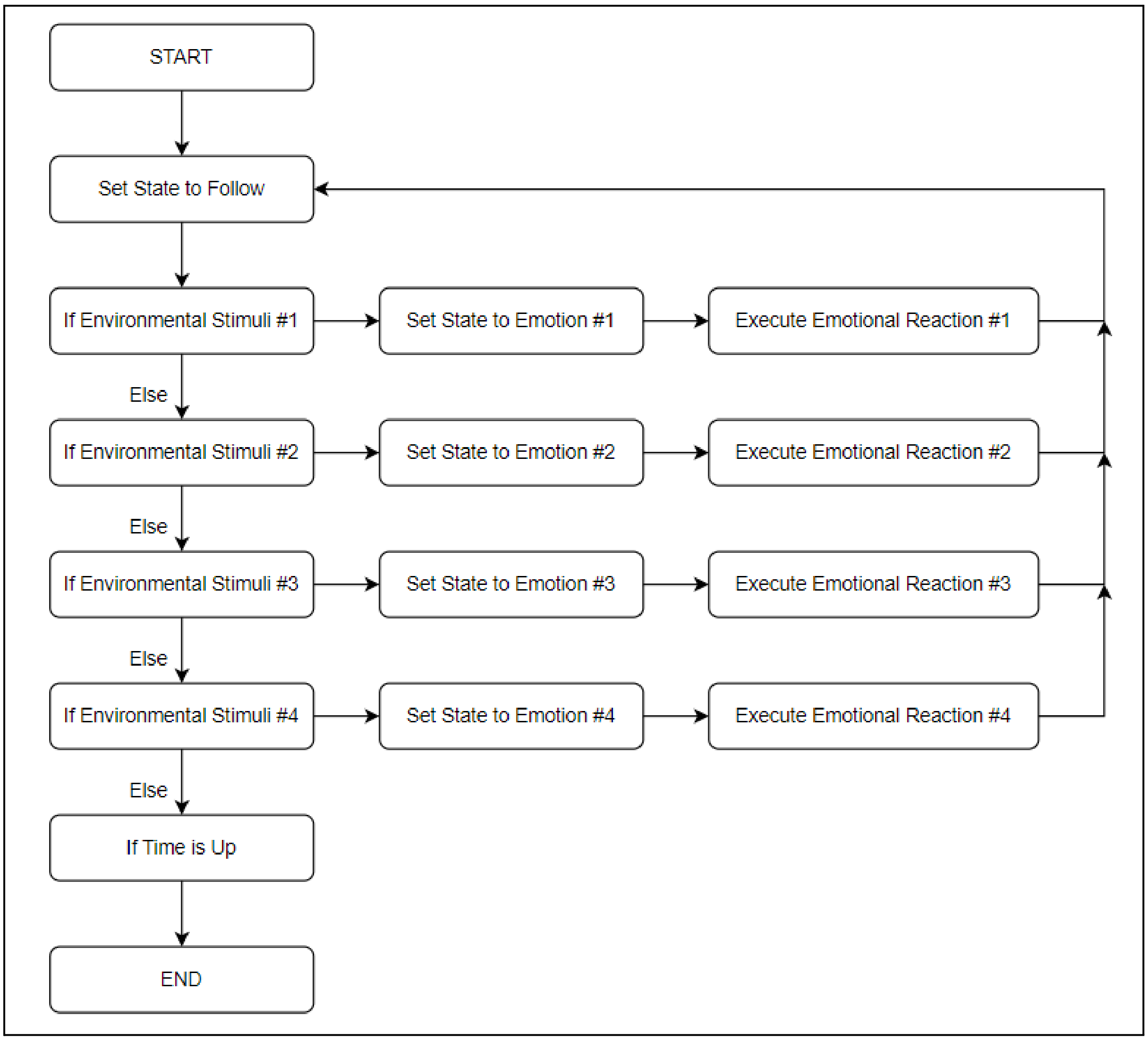
START
Set State to Follow
If Environmental Stimuli #1
Set State to Emotion #1
Execute Emotional Reaction #1
Else
If Environmental Stimuli #2
Set State to Emotion #2
Execute Emotional Reaction #2
Else
If Environmental Stimuli #3
Set State to Emotion #3
Execute Emotional Reaction #3
Else
If Environmental Stimuli #4
Set State to Emotion #4
Execute Emotional Reaction #4
Else
If Time is Up
END

Figure 2: Flowchart of General Strategy

The team decided to implement a state and reaction based algorithm to complete contest 3. The algorithm's default state is set such that the robot is constantly searching for and following the user. While following, the code is checking for the 4 different stimuli that can trigger an emotional response. If a stimulus is found to be true, the code executes the instructions for the corresponding emotion and then returns to the follower state. This structure keeps the code modular ensuring that it executes quickly and is easy to debug and update.

There are two major challenges of contest 3, the first being that the robot must recognise the environmental stimuli and the second being that the robot must display recognisable emotions.

2.1 Environmental Stimuli

In addition to the given stimuli mentioned in section 1.2, the team decided to use the following two stimuli:

- Robot gets picked up off the floor.
- Robot gets tilted such that one of its wheels leaves the floor.

To ensure that the robot can easily recognise these stimuli and change its state, the team decided to make use of a different sensor and their associated callbacks and subscribers for each situation. For the first case where the robot loses its user, the team used the data generated by the given follower callback function. For the second case of where the robot is obstructed, the team used the bumpers and a bumper callback function. For the third and fourth case where the robot gets picked up off the floor and tilted, the team used the wheel drop sensors and a wheel drop callback function.

By having a distinct sensor and function for each environmental scenario, it ensures that the emotional responses of the robot are kept separate and that no overlap can occur.

2.2 Emotional Responses

The table below shows the emotions associated with each environmental stimulus.

Table 1.0: Table of Stimuli and their associated emotions

Stimulus	*Emotions*
Loses track of the person	Sadness (Primary)
Static obstacle in path	Angry (Primary)
Picked up off the floor	Infatuation (Secondary)
Tilted one side	Fear (Secondary)

The team used sound, motion and images to display the different emotions. The actions associated with each emotion is as follows:

- Sadness - A video of a character displaying sadness is put on the laptop screen, audio sounds are played with the robot sweeping side to side slowly
- Angry - A video of a character displaying anger is put on the laptop screen, audio sounds are played and the robot rapidly rotates in one spot.
- Infatuation - A video of a character displaying infatuation is put on the laptop screen, audio sounds are played with the robot driving slowly in circles.
- Fear - A video of a character displaying fear is put on the laptop screen, audio sounds are played with the robot driving back and forth.

3.0 DETAILED ROBOT DESIGN AND IMPLEMENTATION

3.1 Sensory Design

The turtlebot contains various sensors such as encoders, bumper sensors, cliff sensors, wheel drop sensors, gyroscope and a Kinect laser module. The bumpers, wheel drop and Kinect laser module sensors were used by the team to complete contest 3.

3.1.1 Kinect Module - IR Laser Depth Scanner

The sensor projects IR light into the environment and uses the time it takes for the light to bounce back as a method of calculating the distance away from an object. This data is used as a part of the given follower callback function and as such, the team indirectly made use of this sensor. By monitoring the follower callback function, the team was able to set the robot to its 'sadness state' whenever the turtlebot was not able to find the person it was following (1st environmental stimulus).

3.1.2 Bumper Sensors

The turtlebot is equipped with 3 bumper sensors on the front, left and right of the robot's base. These sensors are a type of proximity sensor that can send a signal to the TurtleBot's control system whenever they detect a hit. The team decided to use this sensor to detect when there is an obstacle in the robot's path (2nd environmental stimulus). A callback function combined with a boolean variable was created to detect if any of the bumpers were hit. If a hit occurred, the boolean variable was set to true and this updated the robot's state to its 'anger state.'

3.1.3 Wheel Drop Sensors

The turtlebot has two wheels, each wheel has its own wheel drop sensor. These sensors are a type of electro-mechanical switch. When the wheels are in contact with the ground, the switch is closed, when they are lifted off the ground, the switch is open. Whenever the switches open, they send a signal to the TurtleBot's control system. As such, these sensors were used to detect when the robot was lifted off the floor or tilted (3rd and 4th environmental stimulus). Similar to the bumpers in section 3.1.1 above, the team created a callback function and boolean variable that can change the robot state. The function works such that whenever the sensors detect that at least one of the wheels are lifted off the ground, it then checks whether both wheels are dropped or only one. If both wheels are dropped, the robot enters the "infatuation state". If only one wheel is dropped, then it enters the "fear state".

3.2 Controller Design

The design of the controller for contest 3 can be described as modular and state based. The code is structured such that it is broken up into 5 sections, each section representing a different state. There is a default state called the ‘followerState’ to which the robot is initialised. The other states each represent a different emotion, they are: ‘sadState’, ‘angryState’, ‘infatuationState’ and ‘fearState’. The code uses different callback functions that retrieve robot sensor data. A series of if-else statements are then used to analyse the data and the state of the robot. The diagram below describes the control architecture and the link between high level control and low level control.

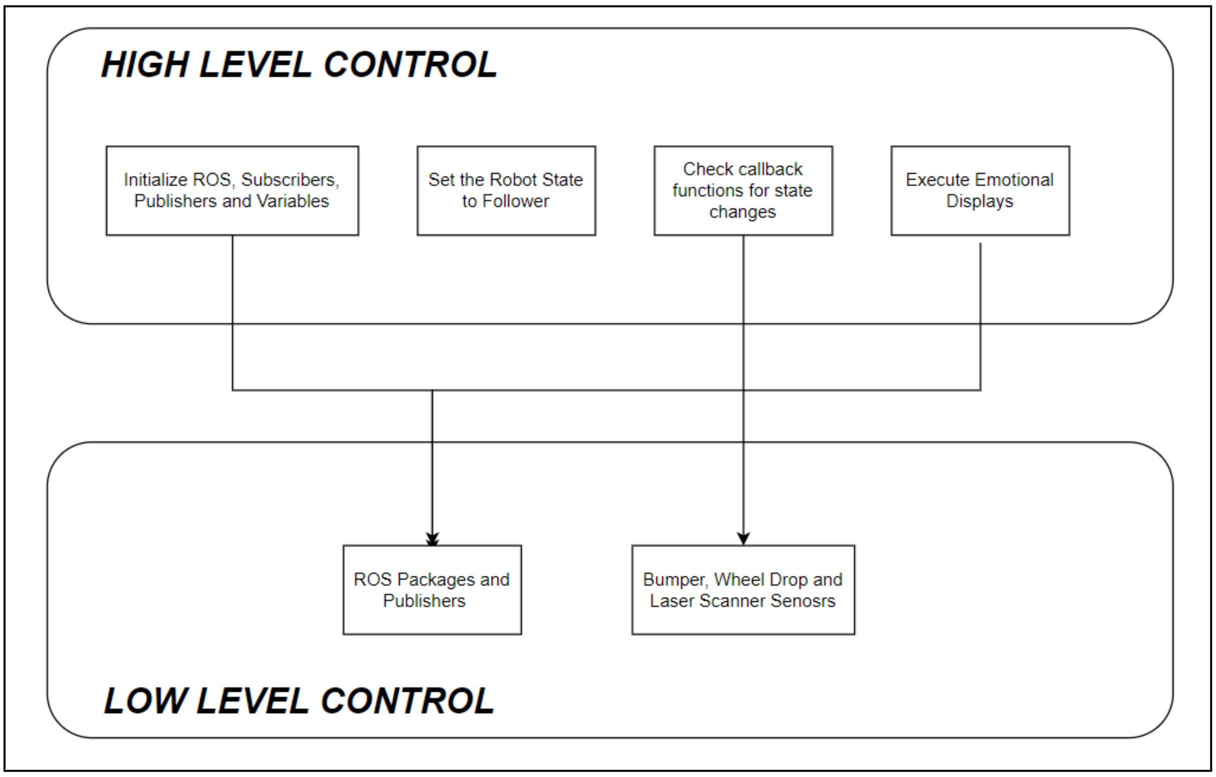
HIGH LEVEL CONTROL
Initialize ROS, Subscribers, Publishers and Variables
Set the Robot State to Follower
Check callback functions for state changes
Execute Emotional Displays
ROS Packages and Publishers
Bumper, Wheel Drop and Laser Scanner Senosrs
LOW LEVEL CONTROL

Figure 3: Link Between High and Low Level Control

The high level control involves the callBack functions and movement functions that the team created to adjust the robot’s state and control the robot’s motion. These functions make use of the components that belong to the low level structure such as the different sensors and ROS packages. Particularly, the high level functions rely mainly on the kinetic laser sensor, bumper sensors, wheel drop sensors and the ‘sound_play’ package. This controller design was adapted from the intelligent control and robot emotions lecture slides.

While the different movement, sound and video functions represent a degree of planning in the code, the overall coding structure is very much reactive in nature. The robot is allowed to interact with its environment and change its behaviour based on the events detected by its sensors.

3.2.1 States & Functions

This section describes the different states, their associated algorithms and the manner in which the robot changes states.

Default State - followerState

The team was given a variable called 'worldState'. The different robot states were assigned to this variable. At the beginning of the code, the 'worldState' is set to 'followerState'. This ensures the robot begins the contest by searching for and following the user.

This state makes use of the given follower callback function: *followerCB()*. It relies on the follower.cpp file given to the team. The data from the function is used to pass velocity instructions to the robot via the *vel_pub.publish()* function. These instructions allow the robot to follow the person in front of it.

Emotion #1 - sadnessState

The robot enters the 'sadnessState' whenever it loses track of the person it is following. The team made use of the laser callback function: *laserCallback()* to determine when the Robot loses its user. This function takes a parameter of type *sensor_msgs::LaserScan::ConstPtr* which contains information about the laser depth scanner. The function calculates the minimum distance from the robot to the object in front of it. It checks if the minimum distance is greater than a specified threshold and if so, it sets the 'worldState' variable to 'sadnessState'.

In the main function, the code uses if-else statements to check the value of 'worldState'. If 'worldState' is set to 'sadnessState' the robot executes the following instructions. First the function *stopMovement()* is called to set the robot's velocity to zero. Then, the video, sound and motion functions associated with anger are called simultaneously. To play videos, the team developed the *displayVideo()* function which uses the openCV library to play a video frame by frame on the laptop screen.

All sounds were played using the ROS *sound_play* package. The motion of sadness was implemented using the cryingSweep() function which uses velocity commands to force the robot to sweep left and right for a set period of time. Additionally, the team implemented the use of threads to ensure that all actions occurred simultaneously. Once the robot completed the sadness instructions, the 'worldState' variable was reset to 'followerState'.

Emotion #2 - angerState

The robot enters the 'angerState' whenever one of its 3 bumpers is hit. To determine if a bumper is hit, the team created the bumper callback function: *bumperCB()*. This function takes in a pointer to a message of type kobuki_msgs::BumperEvent. It updates the state of the bumper array based on the received bumper message. If the bumper state is pressed, the function changes the value of the 'worldState' variable to 'angerState'.

If the ‘worldState’ is found to be ‘angerState’, the robot executes the following instructions. First, the function *stopMovement()* is called to set the robot's velocity to zero. Then, the video, sound and motion functions associated with anger are called simultaneously. The motion of anger was controlled using the *angerRotate()* function which uses velocity commands to force the robot to rotate on the spot. Once all of these instructions are completed, the ‘worldState’ is reset to ‘followerState’.

Emotion #3 - fearState

The robot enters the ‘fearState’ whenever one of its wheels is lifted off the ground (the robot is tilted). The team created the *wheelDropCB()* callback function to check the status of the wheels. The function takes a parameter of type *kobuki_msgs::WheelDropEvent::ConstPtr* which contains information about the state of the wheels. If the function finds either the left or right wheel is lifted (raised), it sets the robot’s state to ‘fearState’.

In the main function the ‘fearState’ instructions are as follows: *stopMovement()* is called, the video and sound are played and movement function is called. The fear movements are controlled by the *fearMotion()* which forces the robot to drive forward and backward until the emotional display is complete. Similarly to the other emotions, the ‘worldState’ is reset to ‘followerState’ once all actions are complete.

Emotion #4 - infatuationState

The ‘infatuationState’ is also set by the *wheelDropCB()* callback function. If the function finds that both wheels are ‘dropped’ (robot lifted off the floor), the ‘worldState’ variable is set to ‘infatuationState’. The instructions for this state are also controlled by an if-else statement located in the main function.

The robot is stopped with the *stopMovement()* function, the associated sounds and video is played and the movements of infatuation are executed by the *infatuationCircles()* function. This function forces the robot to drive in circles until the emotional display is complete. Lastly, the ‘worldState’ is reset to ‘followerState’.

4.0 EXPLANATION OF FULL CODE

The program begins by initialising the ROS node, setting up subscribers for the follower velocity, wheel drop, bumper, and laser scan topics. It also initialises publishers for teleoperation commands. A countdown timer is started and the robot's camera data is transported for both RGB and depth. The different state-related variables are also initialised to their respective numbers and the ‘worldState’ variable is set to the ‘followState’.

Next, a while loop is run until 480 seconds (8 minutes) have passed or the program is terminated. Within the loop, the robot's state is checked using a series of if-else statements. The first statement checks if the state is equal to ‘followerState’, the 2nd checks if it is equal to ‘sadnessSate’ and the 3rd, 4th and 5th statements check if the state is equal to ‘angerState’, ‘fearState’ and ‘infatuationState’ respectively.

The robot executes a different set of instructions based on whatever the current state is. For example, the robot moves forward at a high speed in the anger state, it moves forward and backward in the fear state and moves around in circles in the infatuation state. The code also uses threads to play videos and sounds asynchronously.

The while loop ends when the time limit is reached, or the ROS node is interrupted. Overall, the program makes use of ROS' modular framework to create a state machine that controls a turtleBot’s actions.

5.0 FUTURE RECOMMENDATIONS

If the team had more time to complete contest 3, more research would be done on the relationship between sound, motion and emotion. The TurtleBot has no facial features and as such, the team found it difficult to properly and accurately display the different emotions. The use of sound made did help to make the emotions more clear, however the team found that it did not completely remove confusion. For example, sounds of anger and rage, sadness and discontentment are very similar. This made it difficult to distinguish between emotions that are similar.

Therefore, with more time, the team would have created a library of sounds and surveyed peers and the general public to determine the exact sounds that are associated with a particular emotion. This would help clarify the minor differences between sounds of similar emotions. Additionally, humans/animals are capable of using facial muscles and limbs to portray emotions. The TurtleBot lacks these capabilities and is only able to rotate or move in an x/y direction. As such, the team found it difficult to display emotions with these movement limitations. Given more time, the team would have studied more on the relationship between movement and emotions and the ways in which these movements can be applied to a limbless robot.

Another consideration for the future would be to implement more environmental stimuli and more emotional reactions. Also, the team can explore having the turtlebot portray different emotional reactions to the same stimulus. An individual human/animal can respond differently to the same stimuli based on a variety of different factors. For example, a human can get angry or sad or almost any other emotion if someone blocks their path. To simulate this, the team can incorporate a random feature to the robot’s coding structure. For example, if the robot encounters an obstacle that prevents it from following the person, instead of immediately getting enraged, a random function can be used to generate a number. Based on this number the robot can either enter its angry or sad state. This will add an unpredictability to the robot’s behaviour that will more closely match human emotions.

6.0 APPENDIX

Appendix A - Contribution Table

Team Member	Tasks
Yu-Tung Chen	Help developed the wheel drop callback function, infatuation state and responsible for robot testing. Recorded and edited the video for all emotion states. Edited Section 3.1 and 2.2 of the report.
Kevon Seechan	Responsible for developing the overall code structure, ensuring the robot correctly switches states when the appropriate sensors are triggered. Helped develop the report.
Miguel Gomez	Responsible for developing all emotion scripts, robot actions for each state and robot testing.
Diego Gomez	Responsible for the development of the sadness emotion ie. when it triggers and what audio is played when it triggers, also responsible for robot testing.

Appendix B - List of Emotions

1. Fear
2. Hate
3. Positively excited
4. Resentment
5. Infatuated
6. Surprise
7. Pride
8. Embarrassment
9. Anger
10. Disgust
11. Sad
12. Rage
13. Discontent

Appendix C - Contest3.cpp

```
#include <header.h>
#include <ros/package.h>
#include <imageTransporter.hpp>
#include <chrono>
#include <kobuki_msgs/WheelDropEvent.h>
#include <kobuki_msgs/BumperEvent.h>
#include <cstdlib>
#include <sensor_msgs/LaserScan.h>
#include <thread>

#define DEG2RAD(deg) ((deg) * M_PI / 180.);

using namespace std;
geometry_msgs::Twist follow_cmd;


//Different Emotional States
int worldState = 0;
int followerState = 0;
int sadnessState = 0;
int angerState = 0;
int fearState = 0;
int infatuationState = 0;

//for wheel drop callback function
const uint8_t N_WHEELS = 2;
uint8_t wheel[N_WHEELS] = {kobuki_msgs::WheelDropEvent::RAISED,
kobuki_msgs::WheelDropEvent::RAISED};
bool wheelDrop = false;

//for bumper callback function
const uint8_t N_BUMPER = 3;
uint8_t bumper[N_BUMPER] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED };
bool anyBumperPressed = false;
uint8_t lastBumperPressed = 0;

//for laser callback function
float minLaserDist = std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=5;
float avgLaserDist = 0;
const float personThreshold = 2.5;



//follower callback function
```

```
void followerCB(const geometry_msgs::Twist msg){
   follow_cmd = msg;
}

//bumper callback function
void bumperCB(const kobuki_msgs::BumperEvent::ConstPtr& msg){
        bumper[msg->bumper] = msg->state;

        if(msg->state == kobuki_msgs::BumperEvent::RELEASED){
                ROS_INFO("%d released", msg->bumper);
        }
        else if(msg->state == kobuki_msgs::BumperEvent::PRESSED){
                anyBumperPressed = true;
                ROS_INFO("%d bumper impact detected", msg->bumper);

                //change state if hit
                worldState = angerState;
        }
}

//wheel drop callback function
void wheelDropCB(const kobuki_msgs::WheelDropEvent::ConstPtr& msg){
   const int wheelIndex = msg->wheel;
   const int state = msg->state;

   if (wheelIndex < 0 || wheelIndex > 1){
      ROS_INFO("Invalid wheel number: %d", wheelIndex);
      return;
   }

   wheel[wheelIndex] = state;

   //if both wheels are dropped, set to infatuation
   if (wheel[0] == kobuki_msgs::WheelDropEvent::DROPPED && wheel[1] ==
kobuki_msgs::WheelDropEvent::DROPPED) {
      worldState = infatuationState;
   }
   //if only one wheel is dropped set to fear
   else if (wheel[0] == kobuki_msgs::WheelDropEvent::RAISED && wheel[1] ==
kobuki_msgs::WheelDropEvent::DROPPED) {
      worldState = fearState;
   }
   else if (wheel[0] == kobuki_msgs::WheelDropEvent::DROPPED && wheel[1] ==
kobuki_msgs::WheelDropEvent::RAISED) {
      worldState = fearState;
   }
}
```

```
}



//this callback function returns the minimum laser distance
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
   minLaserDist = std::numeric_limits<float>::infinity();
   avgLaserDist = 0;

   nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
   desiredNLasers = desiredAngle*M_PI / (180*msg->angle_increment);
   //ROS_INFO("Size of laser scan array: %i and size of offset: %i", nLasers,
desiredNLasers);

   //calculate beginning and end of for loop based on desired angle
   int32_t beginIndex = 0, endIndex = nLasers;
   //ROS_INFO("endIndex: %i", endIndex);

   if (desiredAngle * M_PI / 180 < msg->angle_max && -desiredAngle * M_PI / 180 >
msg->angle_min) {
      beginIndex = nLasers / 2 - desiredNLasers;
      endIndex = nLasers / 2 + desiredNLasers;
   }

   float min_dist = std::numeric_limits<float>::infinity();
   float sum_dist = 0;
   int count = 0;
   //loop through and find minimum and non zero values to get an average
   for(int i = beginIndex; i < endIndex; i++){
      float reading = msg->ranges[i];
      // Check if the reading is within a certain range
      if (reading >= msg->range_min && reading <= msg->range_max) {
         min_dist = std::min(min_dist, reading);
         sum_dist += reading;
         count++;
         //ROS_INFO("MIN_DIST Value: %.2fm", min_dist);
      }
   }

   //set minLaserDist based on min_dist
   minLaserDist = min_dist;

   //check if the distance is greater than the threshold, if it is
   //the robot has lost the person, therefore set state to sadness State
   if (minLaserDist > personThreshold){
            worldState = sadnessState;
   }

   //if(minLaserDist == std::numeric_limits<float>::infinity()) minLaserDist = 0;
   ROS_INFO("Min Laser Distance: %.2fm, Avg Laser Distance: %.2fm", minLaserDist,
avgLaserDist);
```

```
}

//Motion Functions\
//function to stop all movement for a set amount of seconds
void stopMovement(ros::Publisher vel_pub, geometry_msgs::Twist velocity, double stop){
        velocity.linear.x = 0;
        velocity.linear.z = 0;
        vel_pub.publish(velocity);
        ros::Duration(stop).sleep();
}

//crying motion
void cryingSweep(ros::Publisher vel_pub, geometry_msgs::Twist velocity, double
angularVelocity, double sadTime) {
    ros::Time startTime = ros::Time::now();

    while ((ros::Time::now() - startTime).toSec() < sadTime) {
        // Rotate to the left
        velocity.angular.z = angularVelocity;
        velocity.linear.x = 0;
        vel_pub.publish(velocity);
        ros::Duration(0.1).sleep(); // Wait for the robot to turn left

        // Rotate to the right
        velocity.angular.z = -angularVelocity;
        velocity.linear.x = 0;
        vel_pub.publish(velocity);
        ros::Duration(0.1).sleep(); // Wait for the robot to turn right
    }

        ROS_INFO("Sad Motion Complete");
    // Stop the robot
    velocity.angular.z = 0;
    velocity.linear.x = 0;
    vel_pub.publish(velocity);
}


//motion for anger
void angerRotate(ros::Publisher vel_pub, geometry_msgs::Twist velocity, double
rotationSpeed, double rotationTime) {
    double fullCircle = DEG2RAD(360);
    double numRotations = rotationTime / (fullCircle / rotationSpeed);
    velocity.linear.x = 0.0;

    //rotate the robot
    for (int i = 0; i < numRotations; i++) {
        velocity.angular.z = rotationSpeed;
        vel_pub.publish(velocity);
        ros::Duration(fullCircle / rotationSpeed).sleep();
    }

            ROS_INFO("Angry Motion Complete");
    //stop rotating
    velocity.angular.z = 0.0;
    vel_pub.publish(velocity);
}

//function to reverse the robot
void reverse(ros::Publisher vel_pub, geometry_msgs::Twist velocity, double distance, double
speed) {
    double time = distance / std::abs(speed); // Time needed to travel the distance
    double start_time = ros::Time::now().toSec(); // Get the current time

    // Reverse the robot
    velocity.linear.x = speed;
    vel_pub.publish(velocity);

    // Keep reversing until the desired distance is reached
    while (ros::Time::now().toSec() - start_time < time) {
        ros::Duration(0.01).sleep();
    }

    // Stop the robot
            ROS_INFO("Reversing Complete");
    velocity.linear.x = 0;
    vel_pub.publish(velocity);
}

// motion for fear for a certain length of time
void fearMotion(ros::Publisher vel_pub, geometry_msgs::Twist velocity, double distance,
double speed, double duration) {
    double startTime = ros::Time::now().toSec();

    while ((ros::Time::now().toSec() - startTime) < duration) {
        // move forward
        velocity.linear.x = speed;
        velocity.angular.z = 0.0;
        vel_pub.publish(velocity);
        ros::Duration(0.1).sleep();

        // stop the robot
        velocity.linear.x = 0.0;
        vel_pub.publish(velocity);
        ros::Duration(0.05).sleep();
        // move backward
        velocity.linear.x = -speed;
        velocity.angular.z = 0.0;
        vel_pub.publish(velocity);
        ros::Duration(0.1).sleep();

        // stop the robot
        velocity.linear.x = 0.0;
        vel_pub.publish(velocity);
        ros::Duration(0.05).sleep();
    }
        ROS_INFO("Fear Motion Complete");
}


//motion for infatuation
void infatuationCircles(ros::Publisher vel_pub, geometry_msgs::Twist velocity, double radius,
double linearSpeed, double duration){
    double circumference = 2 * M_PI * radius;
    double distance = circumference;

    //calculate angular velocity
    double angularSpeed = linearSpeed / radius;
    double startTime = ros::Time::now().toSec();

    while((ros::Time::now().toSec() - startTime) < duration){
        velocity.linear.x = linearSpeed;
        velocity.angular.z = angularSpeed;
        vel_pub.publish(velocity);
        ros::Duration(0.01).sleep();
    }

        ROS_INFO("Infatuation Motion Complete");
    //stop the robot
    velocity.linear.x = 0;
    velocity.angular.z = 0;
    vel_pub.publish(velocity);
}

//this functions allows videos to be displayed
void displayVideo(const std::string& filePath){
    //open the video file using video capture
    cv::VideoCapture cap(filePath);

    //check if gif file was successfully opened
    if(!cap.isOpened()){
        ROS_INFO("Could not open file");
        return;
    }

    //create a full-screen window to display the gif
    cv::namedWindow("VIDEO", cv::WINDOW_FULLSCREEN);

    //read and display each frame of the gif file multiple times
    cv::Mat frame;
    while (cap.read(frame)){
        cv::imshow("VIDEO", frame);
        cv::waitKey(16.68333);   //wait for 16.68333ms
    }
    cap.set(cv::CAP_PROP_POS_FRAMES, 0); //set the video capture to the beginning of
the file for the next loop

    //release the videocapture and destroy the window
    cap.release();
    cv::destroyWindow("VIDEO");
        ROS_INFO("Done Playing Video");
}



int main(int argc, char **argv)
{

        ros::init(argc, argv, "image_listener");
        ros::NodeHandle nh;
        sound_play::SoundClient sc;
        string path_to_sounds = ros::package::getPath("mie443_contest3") + "/sounds/";
        teleController eStop;

        //subscribers
        ros::Subscriber follower =
nh.subscribe("follower_velocity_smoother/smooth_cmd_vel", 10, &followerCB);
        ros::Subscriber wheel = nh.subscribe("mobile_base/events/wheel_drop", 10,
&wheelDropCB);
        ros::Subscriber bumper = nh.subscribe("mobile_base/events/bumper", 10,
&bumperCB);
        ros::Subscriber laser_sub = nh.subscribe("scan", 10, &laserCallback);

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


        // contest count down timer
        ros::Rate loop_rate(10);
        std::chrono::time_point<std::chrono::system_clock> start;
```

```
	start = std::chrono::system_clock::now();
	uint64_t secondsElapsed = 0;

	imageTransporter rgbTransport("camera/image/",
sensor_msgs::image_encodings::BGR8); //--for Webcam
	//imageTransporter rgbTransport("camera/rgb/image_raw",
sensor_msgs::image_encodings::BGR8); //--for turtlebot Camera
	imageTransporter depthTransport("camera/depth_registered/image_raw",
sensor_msgs::image_encodings::TYPE_32FC1);

	//define the states
	followerState = 0;
	sadnessState = 1;
	angerState = 2;
	fearState = 3;
	infatuationState = 4;

	//set the state to follow initially
	worldState = followerState;

	geometry_msgs::Twist vel;

	//time to stop robot at the beginning of each state
	double stopTime = 3.0;

	//for reversing
	double reverseDist = 0.2;
	double reverseSpeed = 0.5;

	//for sad state
	double sadVel = 0.5;
	double sadTime = 30;

	//for anger state
	double angerSpeed = 2.0;
	double angerTime = 30;

	//for fear state
	double fearDist = 0.5;
	double fearSpeed = 1.0;
	double fearTime = 30;

	//for infatuation state
	double infatuationRadius = 0.5;
	double infatuationSpeed = 0.5;
	double infatuationTime = 40;

	//velocity variables
```

```
        double angular = 0;
        double linear = 2.0;
        vel.angular.z = angular;
        vel.linear.x = linear;

        //Files when running on Kevon Computer
        //list of file paths for sadness
        std::string sadVideoPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Sadness.mp4";
        string sadSoundPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Sadness.wav";

        //list of file paths for anger
        std::string angerVideoPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Anger.mp4";
        string angerSoundPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Anger.wav";

        //list of file paths for fear
        std::string fearVideoPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Fear.mp4";
        string fearSoundPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Fear.wav";

        //list of file paths for infatuation
        std::string infatuationVideoPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Infatuation.mp4";
        string infatuationSoundPath =
"/home/turtlebot/catkin_ws/src/mie443_contest3/sounds/Infatuation.wav";


        while(ros::ok() && secondsElapsed <= 480){
                ros::spinOnce();

                //default state of the robot; follow state
                if (worldState == followerState){
                        ROS_INFO("Follower State");
                        //vel_pub.publish(follow_cmd);
                        vel_pub.publish(vel);
                }

                //robot loses track of person it is following --> sadness state
                else if (worldState == sadnessState){
                        ROS_INFO("Entering Sad State");

                        //stop robot movement
                        ROS_INFO("Stopping Robot");
                        stopMovement(vel_pub, vel, stopTime);
```

```
			//play the video
			ROS_INFO("Beginning Video");
			std::thread sadVideoThread(displayVideo, sadVideoPath);

			//play the sound
			ROS_INFO("Beginning Sound");
			std::thread sadSoundThread([&sc,
sadSoundPath](){sc.playWave(sadSoundPath);});

			//execute sad movement
			ROS_INFO("Beginning Sad Movement");
			cryingSweep(vel_pub, vel, sadVel, sadTime);

			sadVideoThread.join();
			sadSoundThread.join();

			//reset state
			ROS_INFO("Resetting to Follower State");
			worldState = followerState;
		}

		//bumper trigger --> anger state
		else if (worldState == angerState){
			ROS_INFO("Entering Anger State");

			//stop robot movement
			ROS_INFO("Stopping Robot");
			stopMovement(vel_pub, vel, stopTime);

			//reversing robot
			ROS_INFO("Reversing Robot");
			reverse(vel_pub, vel, reverseDist, reverseSpeed)

			//play the video
			ROS_INFO("Beginning Video");
			std::thread angerVideoThread(displayVideo, angerVideoPath);

			//play the sound
			ROS_INFO("Beginning Sound");
			std::thread angerSoundThread([&sc,
angerSoundPath](){sc.playWave(angerSoundPath);});

			//anger movmement
			ROS_INFO("Beginning Angry Movement");
			angerRotate(vel_pub, vel, angerSpeed, angerTime);

			angerVideoThread.join();
			angerSoundThread.join();

			//reset state
			ROS_INFO("Resetting to Follower State");
			worldState = followerState;
		}

		//one wheel drop sensor triggered --> fear state
		else if (worldState == fearState){
			ROS_INFO("Entering Fear State");

			//stop robot movement
			ROS_INFO("Stopping Robot");
			stopMovement(vel_pub, vel, stopTime);

			//play the video
			ROS_INFO("Beginning Video");
			std::thread fearVideoThread(displayVideo, fearVideoPath);

			//play the sound
			ROS_INFO("Beginning Sound");
			std::thread fearSoundThread([&sc,
fearSoundPath](){sc.playWave(fearSoundPath);});

			//execute fear motion
			ROS_INFO("Beginning Fear Movement");
			fearMotion(vel_pub, vel, fearDist, fearSpeed, fearTime);

			fearVideoThread.join();
			fearSoundThread.join();

			//reset state
			ROS_INFO("Resetting to Follower State");
			worldState = followerState;
		}

		//both wheel drop sensors --> infatuation state
		else if (worldState == infatuationState){
			ROS_INFO("Entering Infautation State");

			//stop robot movement
			ROS_INFO("Stopping Robot");
			stopMovement(vel_pub, vel, stopTime);

			//play the video
			ROS_INFO("Beginning Video");
			std::thread infatuationVideoThread(displayVideo,
infatuationVideoPath);
```

```
                              //play the sound
                              ROS_INFO("Beginning Sound");
                              std::thread infatuationSoundThread([&sc,
infatuationSoundPath](){sc.playWave(infatuationSoundPath);});

                              //infatuation motion
                              ROS_INFO("Beginning Infatuation Movement");
                              infatuationCircles(vel_pub, vel, infatuationRadius, infatuationSpeed,
infatuationTime);

                              infatuationVideoThread.join();
                              infatuationSoundThread.join();

                              //reset state
                              ROS_INFO("Resetting to Follower State");
                              worldState = followerState;
                    }

                    secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start).co
unt();
                    loop_rate.sleep();
          }
          return 0;
}
```