Contest 3 Report

MIE443 Mechatronics Systems: Design & Integration
Contest 3: Follow Me Robot Companion
Due: April 9th, 2024

Team #4	
Team Member	**Student Number**
Henry (Hua Hao) Qi	1005758039
Harry Park	1005674405
Alastair Sim	1006460287
Yi Lian	1005709333

1.0 Problem Definition/ Objective

The goal of MIE443: Contest 3 is to develop, in a robot, the ability to engage with users. The interaction is carried out with a TurtleBot, which will follow a user while he/she moves in an open area environment. In addition, the robot should be able to interact with the user through the use of emotions. The team is assigned to develop four distinct emotions for the robot for four different environmental stimuli which occur during the follower task. Out of the four emotions, two must be primary/reactive emotions and two must be secondary/deliberative emotions. The task is complete once the Turtlebot has demonstrated all of its emotions and functionalities and visited all four labelled markers that the user will be instructed to walk to. The robot follower task is defined to be fixed-timed, with a robot-based subject of action, and a movement goal of movement-while. The fixed time constraint is placed as the robot must show all four emotions within a time limit. However, a time constraint is also placed for showing the correct emotion corresponding to the correct stimuli since the robot should react/respond to the stimuli within an appropriate time frame. The subject of action is robot-based since the idea of the task is to have the robot place itself according to the user in the environment. Lastly, the movement goal is movement-while since the robot is following the user while responding to different stimuli in the environment.

1.1 Contest Requirements

The contest requirements are outlined as follows:

- The TurtleBot must be able to identify and track a user.
- The competition environment for this contest will be held in an open area. Within this environment, there will be 4 labelled markers that the person leading the robot will be instructed to walk to.
- For the interaction portion of this contest, the TurtleBot will be designed to have unique emotional responses to 4 environmental stimuli:
    1. When it loses track of the person it is following.
    2. When it cannot continue to track a person due to a static obstacle in its path.
    3. and 4. Two other stimuli of our own choosing.
- For the 2nd stimulus, an obstacle will be placed to obstruct the robot's path. The object will be short enough so that it does not interfere with the depth information provided by the onboard Kinect sensor. The object will be removed from the environment after the TurtleBot has displayed its corresponding emotion.
- For this contest, all of the TurtleBot sensors are available to use. The callbacks required for the Kinect RGB and depth information, the bumper sensor and velocity commands are already provided; however, if you wish to use any other sensory data, your team will need to add the corresponding callbacks and subscribers.

- You are encouraged to use many input and output modes for the robot. Your team must implement original motion and sound for the emotion interaction part of the contest. You have access to the play_sound library which can be used to play stored .wav files. Please note: the emotions of the robot cannot be stated by the robot or team members at any time during the contest (for example "I am feeling sad.").
- Each team will present their robot in front of a panel of three judges. A time limit of 8 minutes will be allocated to the contest to show your robot's functionality to the judges. MIE 443 Contest 3 Page 3
- The robot's emotions must be chosen from the list provided below:
    a. Fear
    b. Positively excited
    c. Infatuated
    d. Pride
    e. Anger
    f. Sad
    g. Discontent
    h. Hate
    i. Resentment
    j. Surprise
    k. Embarrassment
    l. Disgust
    m. Rage

2.0 Strategy

Since the robot's emotions and environmental stimuli are decided on before the contest trials, the team decided to develop a finite state machine that specifies the conditions in which the robot will change state and the actions performed when the state changes. The overall algorithm is separated into five states: default (follower), target lost (stimulus 1), unforeseen obstacle (stimulus 2), sudden movement (stimulus 3), and user mishandling (stimulus 4). These five states correspond to 5 emotions respectively: neutral, sad (secondary), surprise (primary), fear (primary), and rage (secondary), where each emotion is portrayed through the characters of Pixar's film "Inside Out". The robot is preprogrammed to start in the default state, where the follower task begins. It will only transition to one of the other four states if any corresponding stimulus is triggered. When triggered, after expressing its emotion/reaction, the robot will revert to the default state and continue its task.

Figure 2.1: Characters from "Inside Out": Sadness, Anger, Fear, and Riley (Left to Right)

2.1 State 1 - Default State (Follower)

In the default state, the robot will carry out its task as a robot follower, where it also detects any environmental stimulus. When an object (person) is presented in front of the Turtlebot, it will attempt to remain a distance of one meter from the person by adjusting its velocities accordingly. The emotion corresponding to the default state is neutral since there are no environmental stimuli.

2.2 State 2 - Target Lost (Stimulus 1)

When the robot no longer detects the user that it is following, it will transition from the default state to the Target Lost state. This is detected through the follower code - if there are not enough points to generate a point cloud of the user, it means the robot has lost its target. Once the Turtlebot loses track of the person, it will begin to display audio and actions of sadness, as well as a visual of the character “Sadness”. The secondary emotion of sadness is chosen because the robot must think and understand that it has lost track of its human companion. Since it is a reaction to a deliberative emotion, the robot will slowly sweep 45 degrees to the left and to the right in an attempt to relocate its target. As time progresses, the robot gradually transitions from worried to sobbing and eventually cries out loud when it fails to relocate its target after a few attempts to sweep to the left and right. This process of relocating its target is continued until the user is detected again, where the robot will transition back to the default state.

2.3 State 3 - Unforeseen Obstacle (Stimulus 2)

The robot will transition from the default state to the Unforeseen Obstacle state when it encounters an obstacle that it was unaware of. This occurs when the robot’s bumper senses an obstacle but it does not detect it with its Kinect depth sensor. Once the Turtlebot encounters the obstacle, it will move back abruptly and freeze in place. This motion will be accompanied by an abrupt “*DOINK!!*” sound effect to represent surprisement. Also, the main character, Riley’s surprised face will be shown to reinforce the astonishment. In this state, the primary emotion, surprise, is chosen because this is a reactive mechanism, where the robot is startled by the obstacle. After being startled for a few seconds, the robot will transition back to the default state and attempt to follow its target again.

2.4 State 4 - Sudden Movement (Stimulus 3)

The robot will transition from the default state to the Sudden Movement state when it is suddenly picked up by the user. This is determined through the robot’s wheel drop where if the sensors detect wheel drop, then the Turtlebot has been lifted up off the ground. Once the robot has been picked up, it will play the voice line “*Wahhhhh!!!*” and frantically move its wheels simulating when a person kicks their legs in mid-air. The primary emotion fear is chosen because this is a reactive mechanism since the robot is scared of being suddenly picked up. The robot will transition back to the default state once it has been put back on the ground.

2.5 State 5 - User Mishandling (Stimulus 4)

The robot will transition from the default state to the User Mishandling state when it is mistreated by the user through multiple hits on the bumper and is picked up again and held in the air. This is determined through the robot's bumpers and an internal boolean variable that keeps track of whether the robot has been bumped before. When the robot is hit on its bumpers by the target user, as opposed to an obstacle, its anger will start to build up, as it displays the character Anger's rage building up with sound clips such as "*stop it*". Once enough rage has been built up, it will play the voice line "*STOP IT!!*" multiple times along with a visual of Anger's rage burst out. When the robot is in its rage state, it will spin in circles quickly to express its strong anger. The secondary emotion of rage is chosen because the robot must recognize that the user has ignored its request to not be hit in the bumpers. This is continued for a few seconds until the robot has calmed down and transitions back to the default state.

3.0 Robot Design

3.1 Sensory Design

The sensory design of the robot consists of proprioceptive and exteroceptive sensors. The proprioceptive sensors, which include the internal gyroscope and motor encoders, will be discussed in Section 3.1.1 and the exteroceptive sensors, which include the Kinect, wheel drop, and cliff sensors and bumpers, will be discussed in Section 3.1.2.

3.1.1 Proprioceptive Sensors

Proprioceptive sensors are used to determine the internal state of the robot. They help provide internal information about the robot, such as its orientation as well as linear and angular velocities. In our algorithm, we rely on using odometry to estimate the robot's position and orientation relative to a starting location. This starting point is given as an x and y coordinate as well as an orientation around the z-axis. While the x and y coordinate position of the robot is not used in our algorithm, the yaw is used extensively to track the turns that the robot makes. This is done using the internal gyroscope, which helps detect, measure, and maintain the angular motions of the robot. With the help of the gyroscope, we can specify a certain angle for the robot to turn, and determine when the robot is done turning by comparing the yaw of the initial and final positions of the robot. This can ensure that the intended turns for the robot are carried out accurately. As well, the gyroscope is also used to maintain the robot's angular velocity, which is crucial in helping the robot express its emotions. When the robot experiences sadness, it is programmed to turn at a low speed whereas when the robot experiences rage, it is programmed to turn quickly.

The second set of proprioceptive sensors used is the robot's motor encoders, which are used to provide information on a rotary motor's speed and position. With the help of motor encoders, the robot's linear velocity can be specified and maintained. This is mainly used in the follower code, where the robot receives a set of velocity values required to maintain a one-meter following distance to its target user.

3.1.2 Exteroceptive Sensors

Exteroceptive sensors are used to determine the external state of the robot. They can help provide external information about the environment, such as the location of the target user. In our algorithm, we rely on the Kinect depth sensor attached to the robot to detect the existence of a target user and their location. The Kinetic depth sensors use laser scanning by subscribing to the scan topic and retrieving laser distances to the object in front of it. The lasers are produced using an infrared projector and a camera that can see the tiny points that the projector produces. Using the 3D depth sensor, the robot has a 45-degree vertical field of view, a 58-degree horizontal field of view, and a nominal range of 0.8-3.5 meters, which is useful for object detection. While travelling, the follower module uses sensor feedback to help the robot follow and maintain a one-meter distance from the user. In addition to the depth sensor, the Kinect sensor is also equipped with an RGB camera which aids in object detection.

On top of the Kinect sensor, the robot is equipped with three bumpers located on the left, centre, and right sides of the base platform of the robot. These sensors help the robot detect any collisions with obstacles. The sensor feedback of the bumpers helps tell the robot if it has collided with any obstacles that may have been missed in its blind spot. Once the bumper has been pressed, our algorithm will help the robot express the emotion of surprise since it has collided with an unexpected obstacle. In addition, the team decided to utilize the bumper sensor to detect if the robot had been physically hit by its surroundings, which would cause the robot to build up its anger.

The robot is also equipped with 2 wheel drop sensors and 3 cliff sensors which detect wheel drop and measure the distance between the robot and the floor by constantly sending infrared signals to the surface. They help determine if the robot has reached an edge or been lifted if the wheels drop or if the signal doesn't bounce back immediately. This is useful in detecting when the robot is picked up by the user prompting the robot to express fear and rage.

3.2 Controller Design

To implement the strategy described in section 2.0, the controller was designed to operate in five different states: default, target lost, unforeseen obstacle, sudden movement, and user mishandling. Within states 2 and 3 of the finite state machine, the robot follows a sequential controller design, where each stimulus corresponds to an immediate action based on the robot's sensory data to express its primary emotions. In states 1 and 4, the robot plans a set list of actions to execute based on sensory data obtained at each sub-state, following a deliberative controller design. Hence, the overall architecture involves both deliberative and reactive controls, making it a hybrid controller design. Callback functions and custom functions are used to ensure the individual states are completed successfully. The details of the algorithm are explained in the following sections along with a visual representation of the overall control structure in Figure 3.2.1.

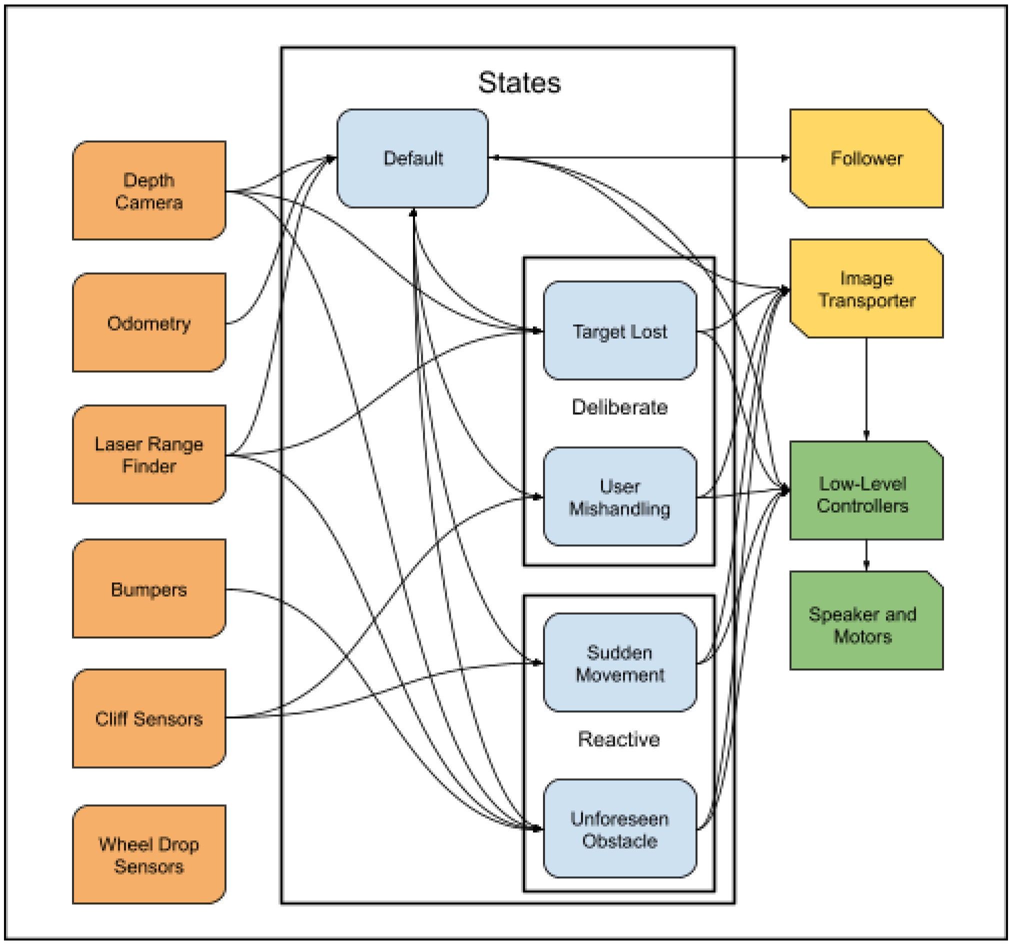


Figure 3.2.1 Overview of controller.

3.2.1 Main Contest Code (contest3.cpp)

This is the main code that initializes ROS by initializing variables and functions as well as subscribing to all necessary ROS topics. These include boolean variables that track the robot's conditions like *bumperPressed* and *robotRaised*, counting variables that track the progress of the emotion like *rxnTime* and *rageCount*, callback functions that extract robot sensor information like *bumperCB()* and *wheelCB()*, and topics that provide instructions from other files or information from sensors like *follower* and *bumper*.

Once initialization has been completed, a while loop is constructed to check for environmental stimuli periodically. Upon startup, it executes *ros::spinOnce()* to update information on subscribed topics or to publish them. The robot starts in the default state and follows the velocity commands provided by the *follower* topic. At the same time, there are condition checks that determine if the robot detected any form of stimuli through its sensors. These include checking if any of the bumpers are pressed, wheels are dropped, and if the robot still detects its target. Once all condition checks are completed, the algorithm proceeds to state identification.

State identification is carried out using the conditions from the robot's sensors. To remain in the default state, the robot must still be following its target with no interference. This means that the bumpers should not be pressed and the wheels should not be dropped. If any one of the three conditions is not met, the condition checks for other states will progress.

The first check is through the robot follower code. When the follower detects a target, it publishes velocities that the robot uses to follow its users. If no target is detected, the velocity published will be zero. This indicates that the robot has lost its target, and will enter the target lost state. In this state, the robot will attempt to display sadness. This is done by prompting the robot to turn left and right by publishing different angular velocities. These turns are controlled by the variable *rxnTime* where at certain times the robot will carry out certain actions (ie. rotate left). These actions include using the *playImage()*, *playVideo()*, and *playWave()* functions to display visuals and sounds.

The second check is through the robot's bumpers. If the bumpers have been pressed but the robot follower code is still publishing non-zero velocities, that means the robot has either hit an unseen obstacle or is being hit by something. To determine what scenario the robot is experiencing, an internal counter tracks how many times the robot's bumpers have been hit within a second. If the robot's bumper records only one hit, then it will respond with surprise. Surprise is expressed using the *playWave()* function to display a *"DOINK"* sound effect, which is a classic surprise sound effect from cartoons. The reaction also pops up a visual of Riley's surprised face, accompanied by a natural reflex of moving away from the unexpected object by publishing a negative linear velocity to prompt the robot to move backwards. If the robot's bumper records more than one hit, it will be directed to the user's mishandling state and record how many times it gets hit. Each hit contributes to its rage build-up, which is shown by displaying a series of images of anger using *playImage()*, accompanied by dialogues of annoyance. Once the count reaches five, the robot will respond with rage. Rage is expressed using the *playVideo()* function to show a video of extreme anger. It is then followed by vigorous spinning by publishing a high angular velocity.

The last check is through the robot's wheel drop. If the robot's wheels are dropped, then the robot is lifted a distance off the floor. Once the robot is lifted, it will respond with fear. Fear is expressed with a screaming sound played using *playWave()* function and accompanied by rapid wheel movement. This wheel movement is prompted by publishing a high linear velocity, which mimics the human behaviour of kicking their legs when lifted in the air.

4.0 Future Recommendations

Future recommendations for enhancing the robot's performance are as follows:

- **Play Video and Move Simultaneously:** The current algorithm uses openCV's *imshow()* function to display the frames of videos, which does not run simultaneously with the robot's movement commands. Consequently, the video cannot play any audio attached to the video file. If the audio can play and the video can be synchronized with the robot's movements, it would make the robot's expressions more natural and intuitive.

- **Play Voice Lines of Original Characters:** Current audio outputs from the robot use non-copyrighted voice clips and background music, which is not in sync with the original characters that appear in the visual outputs. If the audio clips were generated with AI to match the original characters, it would also make the expressions of the robot more appealing.

- **Additional Sensors for Various Stimuli:** In addition to the sensors used in this contest, adding more accurate height sensors or motion detection sensors would allow us to add different types of stimuli to engage with the robot rather than the limited options we had for this contest.

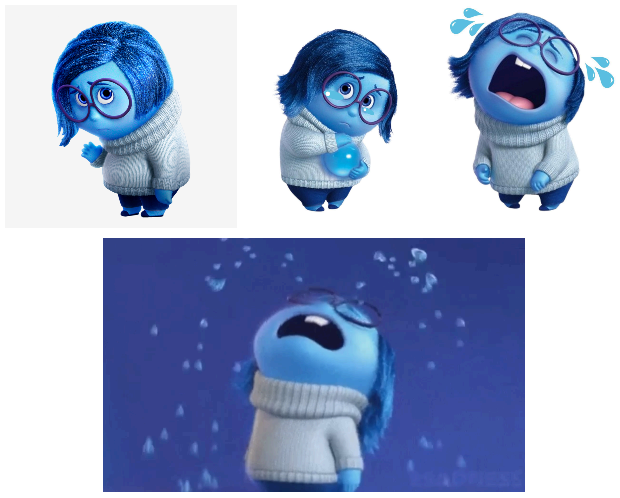
- **Adding Obstacle avoidance**: The robot could instruct the user to wait when it is blocked by a static obstacle then perform a maneuver to avoid the obstacle.

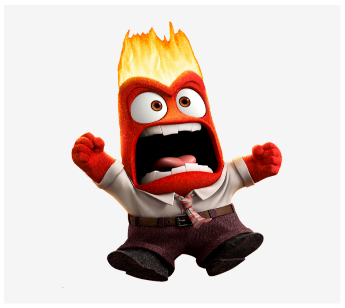
5.0 Appendix

Appendix A: Contribution Table

Section	Subsection	Henry (Hua Hao) Qi	Harry Park	Alastair Sim	Yi Lian
1.0					✔
2.0					✔
3.0	**3.1**	✔	✔		✔
	3.2	✔		✔	✔
4.0			✔	✔	
Code	**Finite State**		✔	✔	
	OpenCV	✔	✔		

Appendix B: Visuals of each Stimulus

Default State: Neutral	
Stimulus 1: Sad	
Stimulus 2: Surprise	

Stimulus 3: Fear	
Stimulus 4: Rage	

Demonstration of Visuals: https://www.youtube.com/watch?v=I3mC5yHaFqk

Appendix C: *contest3.cpp*

```
#include <header.h>
#include <ros/package.h>
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <kobuki_msgs/WheelDropEvent.h>
#include <imageTransporter.hpp>
#include <chrono>
#include <inttypes.h>
#include <unistd.h>
#include <string>

//display video
#include "opencv2/opencv.hpp"
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>
using namespace std;
using namespace cv;

int screen_w = 1366;
int screen_h = 768;

bool robotRaised = false;

int raiseCount = 0;
uint64_t timer = 0;
uint16_t maxHeight = 500;

int lostCount;

bool fearDone = false;
bool bumperDone = false;
int rageCount = 0;


uint8_t      bumperState[3]      =      {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
```

```
uint8_t                              wheelState[2]                              =
{kobuki_msgs::WheelDropEvent::RAISED,kobuki_msgs::WheelDropEvent::RAISED};

bool bumperPressed;

std::chrono::time_point<std::chrono::system_clock> rxnTimeStart;
uint64_t rxnTime = 0;
uint64_t rxnTimeTotal = 0;

geometry_msgs::Twist follow_cmd;
geometry_msgs::Twist raw_cmd;
int world_state;

void rawFollowerCB(const geometry_msgs::Twist msg){
    raw_cmd = msg;
}

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

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

}

void wheelCB(const kobuki_msgs::WheelDropEvent::ConstPtr& msg){ //left is
0, right is 1
    wheelState[msg->wheel] = msg->state;

}

string   path_to_sounds   =   ros::package::getPath("mie443_contest3")   +
"/sounds/";
string   path_to_videos   =   ros::package::getPath("mie443_contest3")   +
"/videos/";
string                                                       emotionArray[4]=
{"Sad_short.mp4","Surprised_short.mp4","Fear_ahh_short.mp4","Anger_short.m
p4"};            // 0=sad, 1=surprised, 2=fear, 3=angy
```

```
// change playimage to display image
int playImage(string emoName) {
    // image = imread(PATH) to overwrite
    namedWindow("Emotion Image", WINDOW_NORMAL);
    resizeWindow("Emotion Image", screen_w, screen_h);
    Mat image = imread(path_to_videos + emoName + ".jpg");
    imshow("Emotion Image", image);
    char c = waitKey(10);
}



int playVideo(String emoName) {
    // ---Code below for displaying video---

            // Create a Video Capture object and open the input file
               // If the input is web camera, pass 0 instead of the video
file
            VideoCapture cap(path_to_videos + emoName + ".mp4");
            namedWindow("Emotion Video", WINDOW_NORMAL);
            resizeWindow("Emotion Video", screen_w, screen_h);

            // Check if camera opened successfully
            if (!cap.isOpened()){
                cout << "Error opening video stream or file" << endl;
                return -1;
            }
            while (1){
                Mat frame;
                // capture frame-by-frame
                cap >> frame;
                // If the frame is empty, break immediately
                if (frame.empty())
                    break;
                // Display the resulting frame
                imshow("Emotion Video", frame);
                // Press ESC on keyboard to exit
                char c=(char)waitKey(25);
                if (c==27)
                    break;
            }
            // When everything done, release the video capture object
            cap.release();
            // --- End of code for displaying video ---
}

void closeVisual(int sleepTime)
{
    sleep(sleepTime);
    destroyAllWindows();
    sleep(1);
}

//--------------------------------------------------------------

int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
    sound_play::SoundClient sc;
    teleController eStop;

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

    //subscribers
                                  ros::Subscriber        follower        =
nh.subscribe("follower_velocity_smoother/smooth_cmd_vel",              10,
&followerCB);
                                ros::Subscriber       rawFollower        =
nh.subscribe("follower_velocity_smoother/raw_cmd_vel",                 10,
&rawFollowerCB);
    ros::Subscriber bumper = nh.subscribe("mobile_base/events/bumper", 10,
&bumperCB);
                                 ros::Subscriber        wheelDrop        =
nh.subscribe("mobile_base/events/wheel_drop", 10, &wheelCB);

    // 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);

    int world_state = 0;

    double angular = 0.0;
    double linear = 0.2;

    geometry_msgs::Twist vel;
    vel.angular.z = angular;
    vel.linear.x = linear;

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


         std::cout << "This is the output from follower: \n" << follow_cmd
<< std::endl;

        //---Check Sensory Inputs ---

        //check bumper

        bumperPressed = false;
          if (bumperState[0] == kobuki_msgs::BumperEvent::PRESSED) //left
bumper
        {
            ROS_INFO("Left Bumper PRESSED!!");
            bumperPressed = true;
        }
```

```
         if (bumperState[1] == kobuki_msgs::BumperEvent::PRESSED) //centre
bumper
        {
            ROS_INFO("Centre Bumper PRESSED!!");
            bumperPressed = true;
        }
        if (bumperState[2] == kobuki_msgs::BumperEvent::PRESSED)
        {
            ROS_INFO("Right Bumper PRESSED!!");
            bumperPressed = true;
        }



            if (wheelState[0] == kobuki_msgs::WheelDropEvent::DROPPED ||
wheelState[1]==kobuki_msgs::WheelDropEvent::DROPPED)
        {
            robotRaised = true;
            ROS_INFO("LIFTED UP!!! Raise count: %d", raiseCount);
        }

        else
        {
            robotRaised = false;
            ROS_INFO("ON GROUND...");
        }


        //---Actions to trigger world states---

        // DEFAULT: state 0 (follower)

        if (!fearDone)
        {
            // 1. lose track - sad...
                    if (world_state == 0 && raw_cmd.linear.z == 0.1 &&
raw_cmd.linear.x == 0 && raw_cmd.angular.z == 0)
            {
                if (world_state != 1)
                {
                }
                lostCount++;
                if (lostCount > 8)
                {
                    if (world_state !=1)
                    {
                        //playImage();
                        sleep(0.5);
                        sc.playWave(path_to_sounds+"sad_bgm.wav");
                        playImage("sad1");
                        ROS_INFO("Reaction timer started!");
                        rxnTimeStart = std::chrono::system_clock::now();
                    }
                world_state = 1;
                }

            }
            else lostCount = 0;
            // 2. obstacle - surprise...! (primary)

            if (!bumperDone)
            {
                if (bumperPressed && world_state == 0)
            {
                if (world_state != 2)
                {
                    sleep(0.5);
                    sc.playWave(path_to_sounds + "surprise1.wav");
                    playImage("surprise");
                    rxnTimeStart = std::chrono::system_clock::now();
                }
                world_state = 2;
            }
            }
            // 3. - pick up
            if (robotRaised)
            {
                ROS_INFO("WORLD STATE: %d", world_state);
```

```
                if (world_state != 3)
                {
                    sleep(0.5);
                    sc.playWave(path_to_sounds + "fear1.wav");
                    playImage("fear1");
                    ROS_INFO("rxn time starting!!!");
                    rxnTimeStart = std::chrono::system_clock::now();

                }

                    world_state = 3; // RAGE(2') if held up at low height

            }
        }
        else
        {
            //increment counters
            if (bumperPressed){ //add counting with robot raise
                if (rageCount ==0) //hit once
                {
                    sleep(0.5);
                    sc.playWave(path_to_sounds + "rage1.wav");
                    playImage("rage1");
                }

                else if (rageCount ==1) //hit twice
                {
                    sc.stopWave(path_to_sounds + "rage1.wav");
                    sleep(0.5);
                    sc.playWave(path_to_sounds+ "rage2.wav");
                    playImage("rage2");

                }
                else if (rageCount ==2) //hit 3times
                {
                    sc.stopWave(path_to_sounds+"rage2.wav");
                    sleep(0.5);
                    sc.playWave(path_to_sounds+"rage3.wav");
                    playImage("rage3");
                }

                else
                {
                    if (world_state != 4)
                    {
                        sc.stopWave(path_to_sounds+"rage3.wav");
                        sleep(0.5);
                        sc.playWave(path_to_sounds+"rage4.wav");
                        rxnTimeStart = std::chrono::system_clock::now();
                    }
                    world_state = 4;
                }
                rageCount ++;
            }


        }

        // Default state: follower
        if(world_state == 0)
        {
            //fill with your code
            vel_pub.publish(follow_cmd);
            ROS_INFO("Following target...");

            if (rageCount == 1) playImage("rage1");
            else if (rageCount == 2) playImage("rage2");
            else if (rageCount == 3) playImage("rage3");
            else if (rageCount == 4) playImage("rage4");
            else playImage("default");


        }
        // 1. lose track of person - sad... (2')
        else if(world_state == 1)
        {
            ROS_INFO("SAD...");
            if (rxnTime < 3) //turn left
            {
                vel.angular.z = 0.6;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad1");
            }

            else if (rxnTime < 4) //stop after turning to left
            {
                vel.angular.z = 0;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad1");
            }

            else if (rxnTime < 9) //turn right
            {
                vel.angular.z = -0.6;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad1");
            }

            else if (rxnTime < 10) //stop and show emotion (sad 1)
            {
                vel.angular.z = 0;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad2");
                sleep(0.5);
                sc.playWave(path_to_sounds+"sad_bgm.wav");
            }

            else if (rxnTime < 16) // turn left again
            {
                vel.angular.z = 0.6;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad2");
            }
            else if (rxnTime < 17) //stop for a sec
            {
                vel.angular.z = 0;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad2");
            }

            else if (rxnTime < 20)
            {
                vel.angular.z = -0.6;
                vel.linear.x = 0;
                vel_pub.publish(vel);
                playImage("sad2");
            }

            //exit statement
            else
            {
                sleep(0.5);
                sc.playWave(path_to_sounds+"sad_final.wav");

                playImage("sad3");
                sleep(2);
                sc.playWave(path_to_sounds+"sad_final.wav");


                playVideo("sad_final");
                closeVisual(0);

                world_state = 0;
                rxnTime = 0;
                lostCount = 0;
            }
        }

        else if (world_state == 2)
        {

            ROS_INFO("SURPRISE!!");
```

```
            if (rxnTime == 0)
            {
                vel.angular.z = 0;
                vel.linear.x = -0.3;
                vel_pub.publish(vel);
            }

            else if (rxnTime < 3)
            {
                ROS_INFO("FROZEN...!");
                vel.angular.z = 0;
                vel.linear.x = 0;
                vel_pub.publish(vel);

            }

            // exit Statement
            else
            {
                sc.stopWave(path_to_sounds+"surprise1.wav");
                closeVisual(0);
                world_state = 0;
                rxnTime = 0;
                bumperDone = true;
            }

        }

        else if (world_state == 3)
        {
            if (robotRaised)
            {
                if (rxnTime < 3) //air,state 3: start video of fear, start
wheel spinning
                {
                    ROS_INFO("(STATE 3) PUT ME DOWN....!");
                    //fear
                    playImage("fear1");
                    vel.angular.z = 0;
                    vel.linear.x = 0.5;
                    vel_pub.publish(vel);
                }
                else //exit statement
                {
                    sleep(1);
                    sc.stopWave(path_to_sounds+"fear1.wav");
                    ROS_INFO("back to world state 0");
                    vel.angular.z = 0;
                    vel.linear.x = 0;
                    vel_pub.publish(vel);

                }
            }
            else
            {
                world_state = 0;
                rxnTime = 0;
                fearDone = true;
            }

        }

        else if (world_state == 4)
        {
            rageCount = 0;
            playImage("rage4");
            // RAge
            if (rxnTime < 15 && !robotRaised)
            {
                vel.angular.z = 2.5;
                vel.linear.x = -0.1;
                vel_pub.publish(vel);
            }

            else
            {
                closeVisual(0);
                sc.stopWave(path_to_sounds+"rage4.wav");
                vel.angular.z = 0;
                vel.linear.x = 0;
```

```
                vel_pub.publish(vel);
                rageCount = 0;
                world_state = 0;
                rxnTime = 0;
            }
        }

          if (world_state == 1 || world_state == 2 || world_state == 3 ||
world_state == 4)
        {
                                                              rxnTime     =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-rxnTimeStart).count();

        }

        ROS_INFO("Reaction Time: %" PRIu64, rxnTime);
                                                     secondsElapsed      =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-start).count();
        loop_rate.sleep();
    }
    ROS_INFO("Finished in: %" PRIu64, secondsElapsed);
    return 0;
}
```

Appendix D: *follower.cpp*

```
/*
 * Copyright (c) 2011, Willow Garage, Inc.
 * 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 Willow Garage, Inc. 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 <ros/ros.h>
#include <pluginlib/class_list_macros.h>
#include <nodelet/nodelet.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Image.h>
#include <visualization_msgs/Marker.h>
#include <turtlebot_msgs/SetFollowState.h>

#include "dynamic_reconfigure/server.h"
#include "turtlebot_follower/FollowerConfig.h"

#include <depth_image_proc/depth_traits.h>


namespace turtlebot_follower
{

//* The turtlebot follower nodelet.
/**
 * The turtlebot follower nodelet. Subscribes to point clouds
 * from the 3dsensor, processes them, and publishes command vel
 * messages.
 */
class TurtlebotFollower : public nodelet::Nodelet
{
public:
  /*!
   * @brief The constructor for the follower.
   * Constructor for the follower.
   */
  TurtlebotFollower() : min_y_(0.1), max_y_(0.5),
                        min_x_(-0.2), max_x_(0.2),
                        max_z_(0.8), goal_z_(0.6),
                        z_scale_(1.0), x_scale_(5.0)
  {

  }

  ~TurtlebotFollower()
  {
    delete config_srv_;
  }

private:
  double min_y_; /**< The minimum y position of the points in the box. */
  double max_y_; /**< The maximum y position of the points in the box. */
  double min_x_; /**< The minimum x position of the points in the box. */
  double max_x_; /**< The maximum x position of the points in the box. */
  double max_z_; /**< The maximum z position of the points in the box. */
   double  goal_z_;  /**<  The  distance  away  from  the  robot  to  hold  the
centroid */
   double  z_scale_;  /**<  The  scaling  factor  for  translational  robot  speed
*/
  double x_scale_; /**< The scaling factor for rotational robot speed */
   bool     enabled_;  /**<  Enable/disable  following;  just  prevents  motor
commands */

  // Service for start/stop following
  ros::ServiceServer switch_srv_;

  // Dynamic reconfigure server
          dynamic_reconfigure::Server<turtlebot_follower::FollowerConfig>*
config_srv_;

  /*!
   * @brief OnInit method from node handle.
   * OnInit method from node handle. Sets up the parameters
   * and topics.
   */
  virtual void onInit()
  {
    ros::NodeHandle& nh = getNodeHandle();
    ros::NodeHandle& private_nh = getPrivateNodeHandle();

    private_nh.getParam("min_y", min_y_);
    private_nh.getParam("max_y", max_y_);
    private_nh.getParam("min_x", min_x_);
    private_nh.getParam("max_x", max_x_);
    private_nh.getParam("max_z", max_z_);
    private_nh.getParam("goal_z", goal_z_);
    private_nh.getParam("z_scale", z_scale_);
    private_nh.getParam("x_scale", x_scale_);
    private_nh.getParam("enabled", enabled_);

    cmdpub_ = private_nh.advertise<geometry_msgs::Twist> ("cmd_vel", 1);
                                                  markerpub_           =
private_nh.advertise<visualization_msgs::Marker>("marker",1);
    bboxpub_ = private_nh.advertise<visualization_msgs::Marker>("bbox",1);
          sub_=   nh.subscribe<sensor_msgs::Image>("depth/image_rect",   1,
&TurtlebotFollower::imagecb, this);

             switch_srv_    =    private_nh.advertiseService("change_state",
&TurtlebotFollower::changeModeSrvCb, this);

                                       config_srv_          =          new
dynamic_reconfigure::Server<turtlebot_follower::FollowerConfig>(private_nh
);

dynamic_reconfigure::Server<turtlebot_follower::FollowerConfig>::CallbackT
ype f =
        boost::bind(&TurtlebotFollower::reconfigure, this, _1, _2);
    config_srv_->setCallback(f);
  }

   void  reconfigure(turtlebot_follower::FollowerConfig  &config,  uint32_t
level)
  {
    min_y_ = config.min_y;
    max_y_ = config.max_y;
    min_x_ = config.min_x;
    max_x_ = config.max_x;
    max_z_ = config.max_z;
    goal_z_ = config.goal_z;
    z_scale_ = config.z_scale;
    x_scale_ = config.x_scale;
  }

  /*!
   * @brief Callback for point clouds.
   * Callback for depth images. It finds the centroid
   * of the points in a box in the center of the image.
   * Publishes cmd_vel messages with the goal from the image.
   * @param cloud The point cloud message.
   */
  void imagecb(const sensor_msgs::ImageConstPtr& depth_msg)
  {

    // Precompute the sin function for each row and column
    uint32_t image_width = depth_msg->width;
    float x_radians_per_pixel = 60.0/57.0/image_width;
    float sin_pixel_x[image_width];
    for (int x = 0; x < image_width; ++x) {
      sin_pixel_x[x] = sin((x - image_width/ 2.0)  * x_radians_per_pixel);
    }

    uint32_t image_height = depth_msg->height;
    float y_radians_per_pixel = 45.0/57.0/image_width;
    float sin_pixel_y[image_height];
    for (int y = 0; y < image_height; ++y) {
      // Sign opposite x for y up values
                sin_pixel_y[y]   =   sin((image_height/   2.0   -   y)      *
y_radians_per_pixel);
    }

    //X,Y,Z of the centroid
    float x = 0.0;
    float y = 0.0;
    float z = 1e6;
    //Number of points observed
    unsigned int n = 0;

    //Iterate through all the points in the region and find the average of
the position
                const    float*    depth_row    =    reinterpret_cast<const
float*>(&depth_msg->data[0]);
    int row_step = depth_msg->step / sizeof(float);
       for  (int  v  =  0;  v  <  (int)depth_msg->height;  ++v,  depth_row  +=
row_step)
    {
     for (int u = 0; u < (int)depth_msg->width; ++u)
     {
                                                     float      depth      =
depth_image_proc::DepthTraits<float>::toMeters(depth_row[u]);
        if (!depth_image_proc::DepthTraits<float>::valid(depth) || depth >
max_z_) continue;
       float y_val = sin_pixel_y[v] * depth;
       float x_val = sin_pixel_x[u] * depth;
       if ( y_val > min_y_ && y_val < max_y_ &&
            x_val > min_x_ && x_val < max_x_)
       {
         x += x_val;
         y += y_val;
         z = std::min(z, depth); //approximate depth as forward.
         n++;
       }
     }
    }

     //If there are points, find the centroid and calculate the command
goal.
    //If there are no points, simply publish a stop goal.
    if (n>4000)
    {
      x /= n;
      y /= n;
      if(z > max_z_){
           ROS_INFO_THROTTLE(1, "Centroid too far away %f, stopping the
robot\n", z);
        if (enabled_)
        {
                               cmdpub_.publish(geometry_msgs::TwistPtr(new
geometry_msgs::Twist()));
        }
        // Added 0.1 z velocity to indicate robot has lost target
        geometry_msgs::TwistPtr cmd(new geometry_msgs::Twist());
        cmd->linear.z = 0.1;
        cmdpub_.publish(cmd);
        return;
      }
      ROS_INFO_THROTTLE(1, "Centroid at %f %f %f with %d points", x, y, z,
n);
      publishMarker(x, y, z);

      if (enabled_)
      {
        geometry_msgs::TwistPtr cmd(new geometry_msgs::Twist());
        cmd->linear.x = (z - goal_z_) * z_scale_;
        cmd->angular.z = -x * x_scale_;
        cmdpub_.publish(cmd);
      }
    }
    else
    {
       ROS_INFO_THROTTLE(1, "Not enough points(%d) detected, stopping the
robot", n);
      publishMarker(x, y, z);

      if (enabled_)
      {
                               cmdpub_.publish(geometry_msgs::TwistPtr(new
geometry_msgs::Twist()));
      }
      // Added 0.1 z velocity to indicate robot has lost target
      geometry_msgs::TwistPtr cmd(new geometry_msgs::Twist());
      cmd->linear.z = 0.1;
      cmdpub_.publish(cmd);
    }

    publishBbox();
  }

  bool changeModeSrvCb(turtlebot_msgs::SetFollowState::Request& request,
                       turtlebot_msgs::SetFollowState::Response& response)
  {
    if ((enabled_ == true) && (request.state == request.STOPPED))
    {
      ROS_INFO("Change mode service request: following stopped");
```

```
                                cmdpub_.publish(geometry_msgs::TwistPtr(new
geometry_msgs::Twist()));
      enabled_ = false;
    }
    else if ((enabled_ == false) && (request.state == request.FOLLOW))
    {
      ROS_INFO("Change mode service request: following (re)started");
      enabled_ = true;
    }

    response.result = response.OK;
    return true;
  }

  void publishMarker(double x,double y,double z)
  {
    visualization_msgs::Marker marker;
    marker.header.frame_id = "/camera_rgb_optical_frame";
    marker.header.stamp = ros::Time();
    marker.ns = "my_namespace";
    marker.id = 0;
    marker.type = visualization_msgs::Marker::SPHERE;
    marker.action = visualization_msgs::Marker::ADD;
    marker.pose.position.x = x;
    marker.pose.position.y = y;
    marker.pose.position.z = z;
    marker.pose.orientation.x = 0.0;
    marker.pose.orientation.y = 0.0;
    marker.pose.orientation.z = 0.0;
    marker.pose.orientation.w = 1.0;
    marker.scale.x = 0.2;
    marker.scale.y = 0.2;
    marker.scale.z = 0.2;
    marker.color.a = 1.0;
    marker.color.r = 1.0;
    marker.color.g = 0.0;
    marker.color.b = 0.0;
    //only if using a MESH_RESOURCE marker type:
    markerpub_.publish( marker );
  }
```

```
void publishBbox()
{
  double x = (min_x_ + max_x_)/2;
  double y = (min_y_ + max_y_)/2;
  double z = (0 + max_z_)/2;

  double scale_x = (max_x_ - x)*2;
  double scale_y = (max_y_ - y)*2;
  double scale_z = (max_z_ - z)*2;

  visualization_msgs::Marker marker;
  marker.header.frame_id = "/camera_rgb_optical_frame";
  marker.header.stamp = ros::Time();
  marker.ns = "my_namespace";
  marker.id = 1;
  marker.type = visualization_msgs::Marker::CUBE;
  marker.action = visualization_msgs::Marker::ADD;
  marker.pose.position.x = x;
  marker.pose.position.y = -y;
  marker.pose.position.z = z;
  marker.pose.orientation.x = 0.0;
  marker.pose.orientation.y = 0.0;
  marker.pose.orientation.z = 0.0;
  marker.pose.orientation.w = 1.0;
  marker.scale.x = scale_x;
  marker.scale.y = scale_y;
  marker.scale.z = scale_z;
  marker.color.a = 0.5;
  marker.color.r = 0.0;
  marker.color.g = 1.0;
  marker.color.b = 0.0;
  //only if using a MESH_RESOURCE marker type:
  bboxpub_.publish( marker );
}

ros::Subscriber sub_;
ros::Publisher cmdpub_;
ros::Publisher markerpub_;
ros::Publisher bboxpub_;
```

```
};

PLUGINLIB_DECLARE_CLASS(turtlebot_follower,             TurtlebotFollower,
turtlebot_follower::TurtlebotFollower, nodelet::Nodelet);

}
```