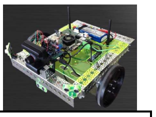


Glasgow Caledonian University

Intelligent Robotics and Mechatronics

Module project – REPORT ‘Follow the Walls’

Name: Michael Jeans
Supervisor: Mario Mata

Date: 1.4.22

Contents

1) Introduction.

The aim of the project is to program a 3 wheeled robot so that it can follow objects and walls using three sonar sensors mounted on the front of the robot. This project will use a Robot Operating System (ROS) with a system of nodes to direct the robot. VMware and ubuntu will be utilised for programming and testing the robot.

The aims of the project are:

1. Robot must be able to find a wall or object to follow
2. Robot must keep a constant distance to object it is following
3. Robot must be able to handle corners
4. Robot must be able avoid collisions
5. Additional features can be added

This report will detail the ROS program and include a software description of the solution. It will go on to assess the solution through testing, analysis of the process and provide next steps of the project. Full commented code is available in the appendices.

2) ROS node/topic diagram. Program/Software description

To understand how the program works a map has been drawn to show the ROS, Figure 1. By collecting data by subscribing to other nodes, each node can then process that data and publish newly calculated data for other nodes to use. Figure 1 shows the "wallfollow" node subscribes to data from nodes that calculate the distance to an object from each of the sonar sensors. The wallfollow node then used this data to calculate what velocity command to publish using a twist message. Another node will subscribe to this topic to update the PWM signal to the motor drivers.

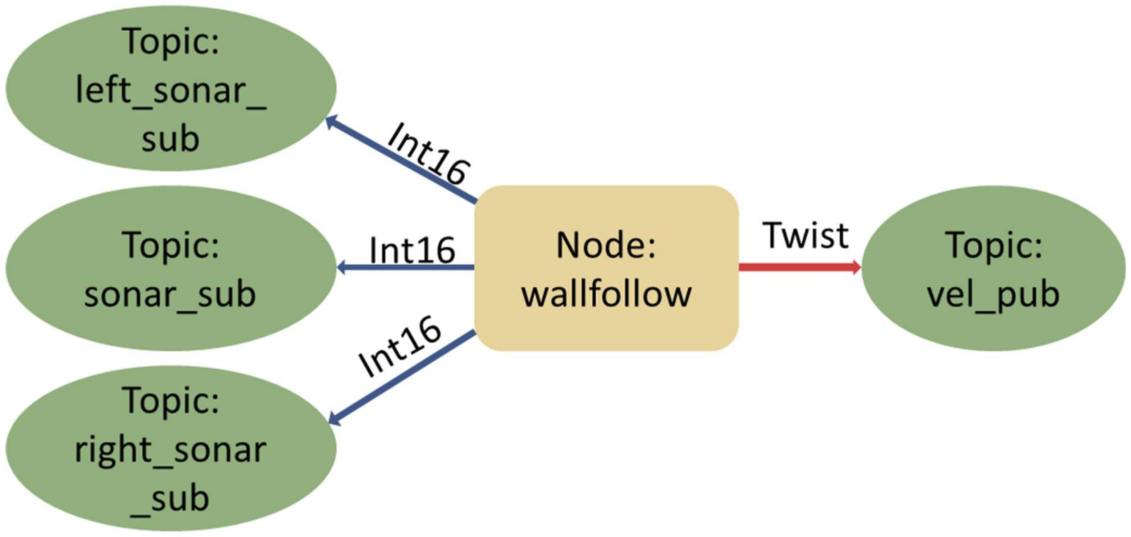
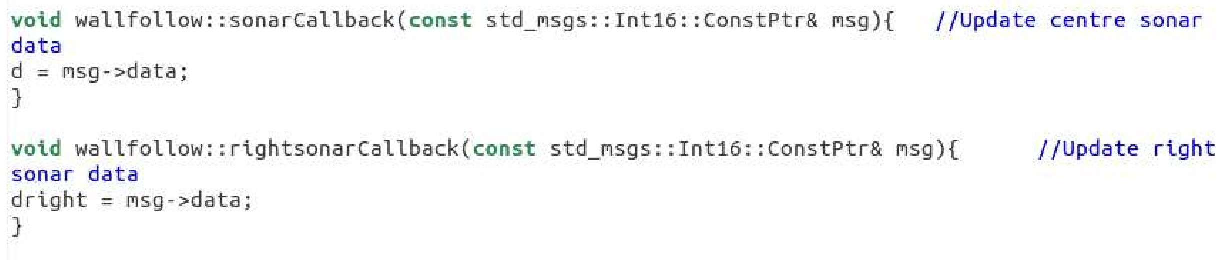


Figure 1 - wallfollow node

How distance from sonar sensor is gained.

The wallfollow node collects data from three different nodes. Each of these is done via a callback function. The sonar distance data is collected for the centre and right sensors Figure 2. The final left sensor callback will be discussed later.

The message type is stated in the callback function allowing the data to be extracted and saved to 'd' and 'dright'. These values are updated every time the previous node publishes new data to the topic, which is everytime the sonar sensor takes a new reading.



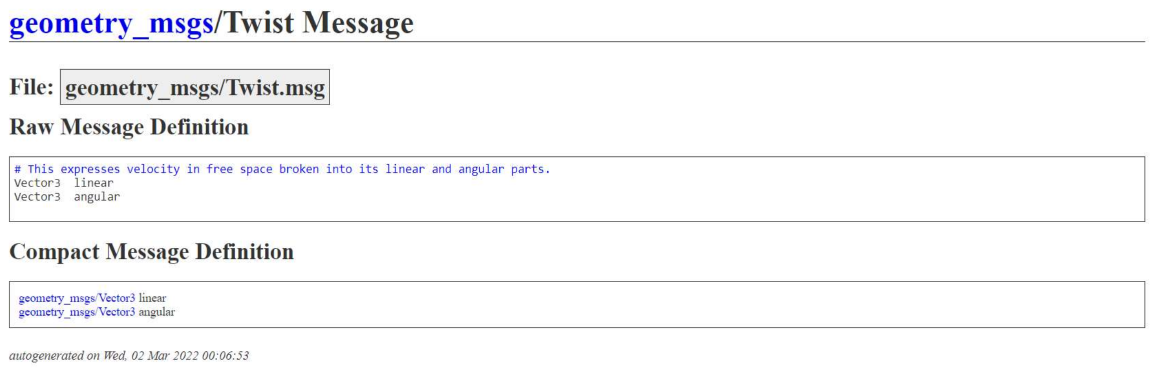
```
void wallfollow::sonarCallback(const std_msgs::Int16::ConstPtr& msg){   //Update centre sonar
data
d = msg->data;
}

void wallfollow::rightsonarCallback(const std_msgs::Int16::ConstPtr& msg){      //Update right
sonar data
dright = msg->data;
}
```

Figure 2 - Collecting data when a subscribed topic publishes

How the velocity message is published

The velocity instructions are published to the vel_pub topic in the form of a twist notification [1]. The twist notification gives the angular velocity and linear velocity components, each in the form 'vector3'. This allows another node to calculate the PWM required at each motor driver to drive the robot at the required resultant velocity.



geometry_msgs/Twist Message

File: geometry_msgs/Twist.msg

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

Compact Message Definition

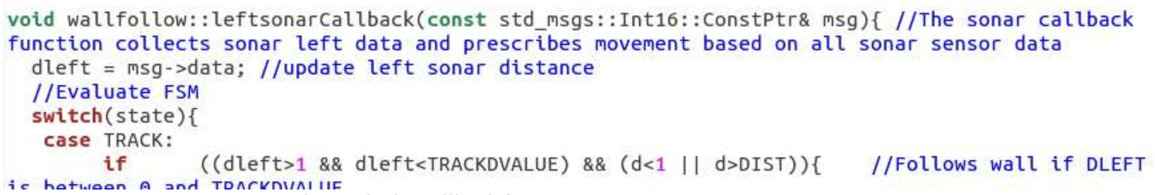
```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

autogenerated on Wed, 02 Mar 2022 00:06:53

Figure 3 - Twist message definition [1]

The code to prescribe movement message, inside the left sonar callback:

Every time the left sonar updates from the previous node the left sonar callback will run. This is effectively like using an event-based interrupt. There are several paths the code can take once it is in the callback.

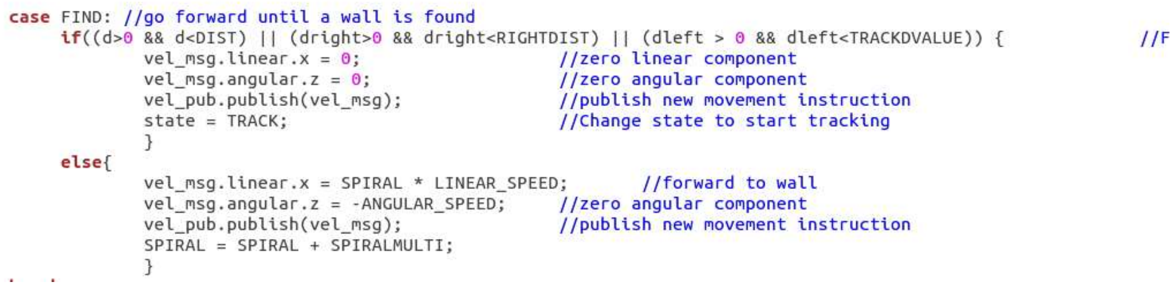


```
void wallfollow::leftsonarCallback(const std_msgs::Int16::ConstPtr& msg){ //The sonar callback
function collects sonar left data and prescribes movement based on all sonar sensor data
  dleft = msg->data; //update left sonar distance
  //Evaluate FSM
  switch(state){
   case TRACK:
        if      ((dleft>1 && dleft<TRACKDVALUE) && (d<1 || d>DIST)){    //Follows wall if DLEFT
is between 0 and TRACKDVALUE
```

Figure 4 - Running tasks inside the callback function

FIND

Firstly, the robot must find a wall to follow, this is number 1 in aims and objectives. For this the robot will travel in a growing spiral until any of the sonar sensors find an object. The is not the quickest way to get to a wall or object, which would be a straight line, however the robot does not know where the nearest wall is yet. This spiral allows for the robot to find the closest object the quickest. The robot turns in an anticlockwise spiral as this makes the transition between FIND and TRACK smoother as the robot will be using the left sonar as the primary sonar for tracking. Figure 5 shows the implementation of the spiral by increasing a linear velocity until the robot finds an object. There is however a limit to this method as the robot has a maximum speed, it will eventually converge to a large circular path. To refine the spiral further, to further increase the radius of the expanding spiral, the angular velocity could also be brought closer and closer to zero. Once a sonar signal is detected, most likely by the left sensor as it triggers at a larger distance, the node sends a new twist message to reduce over velocity to zero and to change the robot's mission from FIND to TRACK.



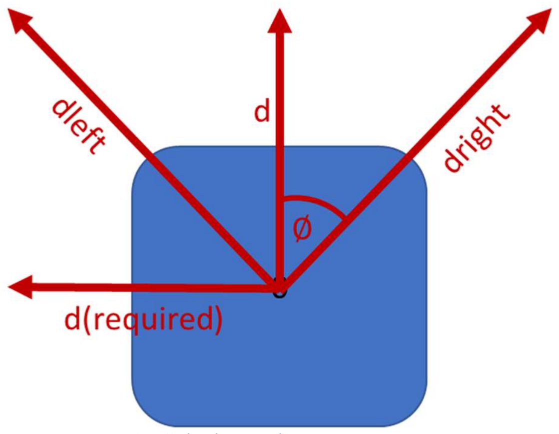
```
  case FIND: //go forward until a wall is found
       if((d>0 && d<DIST) || (dright>0 && dright<RIGHTDIST) || (dleft > 0 && dleft<TRACKDVALUE)) {            //F
               vel_msg.linear.x = 0;                  //zero linear component
               vel_msg.angular.z = 0;                 //zero angular component
               vel_pub.publish(vel_msg);              //publish new movement instruction
               state = TRACK;                         //Change state to start tracking
               }
       else{
               vel_msg.linear.x = SPIRAL * LINEAR_SPEED;       //forward to wall
               vel_msg.angular.z = -ANGULAR_SPEED;     //zero angular component
               vel_pub.publish(vel_msg);              //publish new movement instruction
               SPIRAL = SPIRAL + SPIRALMULTI;
               }
```

Figure 5 - FIND case

TRACK

The track velocity commands are contained within the TRACK switch-case. There are only few different conditions in this mode to cover all the aims. To make the tracking slightly more complicated the robot needs the ability to follow internal walls in a room, and to follow round objects in that room. This can be completed using the same algorithm, with the robot moving clockwise around the room and anticlockwise round an object. The robot only needs to be able to follow, turn left and turn right to satisfy all the remaining aims. The obstacle avoidance is included within these instructions.

To set the distance the robot is away from the wall while tracking, d, dleft and dright were set using calculations in Figure 6. This distance allows the robot to turn without collision.



Set d(required) at distance from wall

$$d = \frac{d(required)}{\tan(\emptyset)}$$

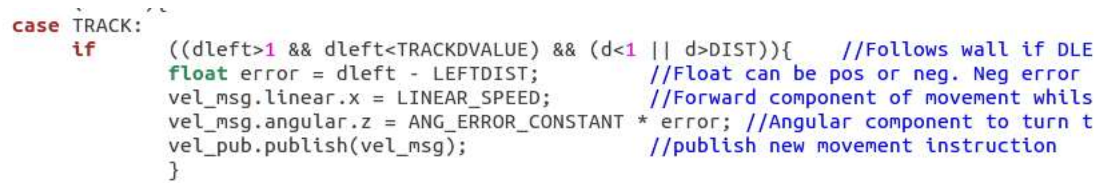
$$dleft = \frac{d(required)}{\sin(\emptyset)}$$

$$dleft = dright$$

Figure 6 - Calculating distances

Follow

An error-based approach is used for this the follow command. The updated left sensor data is compared with a fixed value, TRACKDVALUE. To get an error value. This value is used to calculate an angular velocity value by multiplying it with a proportional constant. This proportional constant dampens the response allowing the robot to smoothly track objects. Using this error method allows the robot to not only follow straight lines but also to follow curves and doorways. However, this again has its limitations due to the sensor location. If the curve of the wall is too tight, the sonar will lose the wall. Adding another sonar sensor on the left side of the robot, at 90degrees to the x axis, would overcome this challenge. This follow mode accounts for differences in wheel circumference as a new error is being calculated every time the sonar updates and the time delta is short.

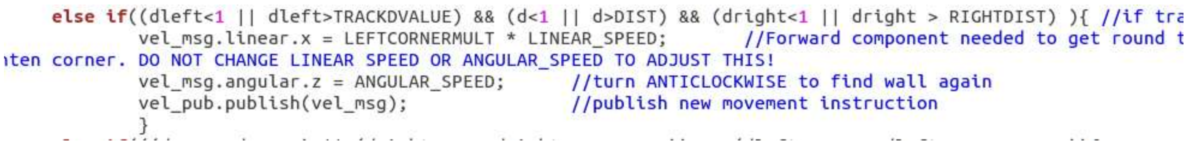


```
case TRACK:
    if      ((dleft>1 && dleft<TRACKDVALUE) && (d<1 || d>DIST)){    //Follows wall if DLE
            float error = dleft - LEFTDIST;          //Float can be pos or neg. Neg error
            vel_msg.linear.x = LINEAR_SPEED;         //Forward component of movement whils
            vel_msg.angular.z = ANG_ERROR_CONSTANT * error; //Angular component to turn t
            vel_pub.publish(vel_msg);                //publish new movement instruction
            }
```

Figure 7 - Following a wall/object

Left turn

The left turn command solves potential issues when the left sonar signal is lost. When the left sonar signal is lost, and no objects are sensed by the other sensors, the robot presumes it is following an object or left curve. The robot will continue moving with both forward and with anticlockwise angular velocity until the left sensor distance returns. At some point the left sonar signal returns the robot returns to following the wall. The tightness of this turn can be edited using the LEFTCORNERMULT value which when below one reduces the linear velocity component and increases it when above one. If the linear component is reduced too much there will be a chance of a collision, due to blind spots created by the current sensor position.

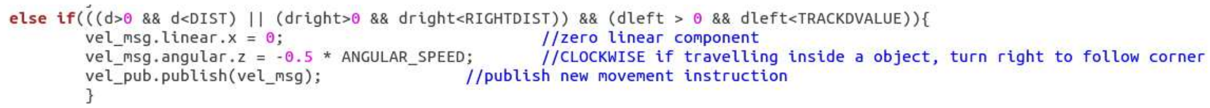


```
    else if((dleft<1 || dleft>TRACKDVALUE) && (d<1 || d>DIST) && (dright<1 || dright > RIGHTDIST) ){ //if tra
            vel_msg.linear.x = LEFTCORNERMULT * LINEAR_SPEED;       //Forward component needed to get round t
ıten corner. DO NOT CHANGE LINEAR SPEED OR ANGULAR_SPEED TO ADJUST THIS!
            vel_msg.angular.z = ANGULAR_SPEED;      //turn ANTICLOCKWISE to find wall again
            vel_pub.publish(vel_msg);               //publish new movement instruction
            }
```

Figure 8 - Left turn

Right turn

There are a couple of different circumstances that would constitute a right turn. Firstly, if an object or wall is detected by the front or right sensor while the left sensor is still within tracking range. And secondly if only the right sensor detects an object. In both these scenarios the outcome is the same and a twist message is published with no forward velocity and allows the robot to rotate clockwise until the left sensor detects the object.



```
else if(((d>0 && d<DIST) || (dright>0 && dright<RIGHTDIST)) && (dleft > 0 && dleft<TRACKDVALUE)){
        vel_msg.linear.x = 0;                           //zero linear component
        vel_msg.angular.z = -0.5 * ANGULAR_SPEED;       //CLOCKWISE if travelling inside a object, turn right to follow corner
        vel_pub.publish(vel_msg);               //publish new movement instruction
        }
```

Figure 9 - Right turn

Software conclusion

The robot uses the left sonar as the primary sonar, and will always travel with it's left side tracking the wall. There is currently no mapping included in this design, so the robot will continue to follow the wall/object until it runs out of charge. Also the sonar distances revert to zero when there is no signal so zero values have .

3) Testing and results discussion

The following tests need to be competed to prove that the aims can be accomplished using this method. A video of the final assault course which includes all manoeuvres in the following tests is available.

FIND test

The robot is placed in a location far away from objects. The node is run. The robots sweep spiral is observed. A successful outcome is when the robot finds an object without collision

Results:

Test number:	*Pass/Fail:*	*Comments:*
1	*Pass*	*Found object and started following. Spiral is tight, takes a long time to reach object.*
2	*Pass*	*Found object and started following. Spiral is improved, but still slow.*
3	*Pass*	*Found object and started following. Spiral seems reasonably well optimised.*

Follow test

The robot is placed alongside a wall. Program is started, distance is monitored. A successful outcome is when the robot tracks the wall/object at the same distance without collision.

Results:

Test number:	*Pass/Fail:*	*Comments:*
1	*Pass*	*Following wall at steady distance. Optimisation is needed as error is large causing the robot to oscillate across x axis.*
2	*Pass*	*Following wall at steady distance. ANG_ERROR_CONSTANT decreased. Error is reduced, robot tracks smoothly at same distance.*

Right turn test

The robot is placed alongside a wall pointing towards a corner. Program is started. A successful outcome is when the robot avoids collision and rotates and continues along wall.

Results:

Test number:	*Pass/Fail:*	*Comments:*
1	*Pass*	*Following wall at steady distance. Stops at required distance and rotates before following wall.* *Possibly the angle of the sonar sensors and distances calculated is slightly wrong as minimal wobble out of corner.*

Left turn test

Robot is placed alongside an object and program is started. Successful outcome is losing edge turns left until edge is regained.

Results:

Test number:	*Pass/Fail:*	*Comments:*
1	*Pass*	*Robot turns and regains edge without collision.* *Suggested optimisation of adding left sensor to cover blind spots.*

Assault course test

Random course includes curves, inside and outside corners and random objects.

Results:

Test number:	*Pass/Fail:*	*Comments:*
1	*Pass*	*Robot navigates successfully with no collisions.*

Performance

The robot passed all set tests with good levels of performance and without any collisions. Optimisations were made to improve the performance of the robot; these optimisations passed each test.

Further improvements

Add in angular component to spiral calculation to widen maximum range of FIND case.

Angle of sonars may be incorrect. This could be down to tolerance of sonar sensor, or an incorrectly calculated angle between sensors. If a small tolerance for distance from wall is needed, then sensors should be calibrated before use. Sonar at 45 degrees might not be reflecting the signal back, a left sonar, at 90 degrees to the wall would reflect directly back giving a stronger signal.

Add in mapping using odometry so that the robot can map its surroundings.

4) Additional functionality

Functionality of the robot has been optimised using the spiral FIND functionality alongside the curve following functionality using the proportional error method. The functionality is explained in the Software Description and is shown in the accompanying video.

5) Reflection and conclusions

All the aims have been met with extra functionality added. Creating a node, using minimal base code is something that really was challenging. Using ROS to code has broadened my knowledge and has given me the opportunity to apply for internships and graduate positions. This class compared to others really gets you focused on practical applications of the theory.

All the learning throughout the classes and tutorials has been condensed into this project. However, this project has not only been about applying knowledge as the coursework was left open for interpretation, giving me a chance to come up with my own ideas and importantly be able to implement my ideas.

Bibliography and Appendices

[1] ROS, “geometry_msgs/Twist Documentation,” *ROS Documentation*, 2021. https://docs.ros.org/en/api/geometry_msgs/html/msg/Twist.html (accessed May 02, 2022).

Code:

```
#include <math.h>

#include <ros/ros.h>

#include <nav_msgs/Odometry.h>

#include <tf/transform_broadcaster.h>

#include "geometry_msgs/Twist.h"

#include "std_msgs/Int16.h"

#include "std_msgs/Int64.h"



#define LINEAR_SPEED 0.4 // m/s

#define ANGULAR_SPEED 0.4 // rad/s

#define ANG_ERROR_CONSTANT 0.05 //can tweak if turning too fast or too slow

#define LEFTCORNERMULT 1 // Left corner multiplier increases/decreases linear
component of left corner scenario

#define TRACKDVALUE 140 // set outer angle limit of tracking whilst moving forward

#define LEFTDIST 60 // cm//Can edit if too far or too close while tracking wall

#define DIST 52 // cm                    //Can edit if too far or too close before performing turn

#define RIGHTDIST 60 // cm  //Can edit if too far or too close before performing turn

#define SPIRALMULTI 0.02 // Spiral multiplier to reduce or increase spiral velocity
component



enum STATE { FIND, TRACK }; //possible states. Either Find wall or Track wall



class wallfollow

{
  public:
    wallfollow(); //constructor method
  private:
    //callback functions to attach to subscribed topics
    void sonarCallback(const std_msgs::Int16::ConstPtr& msg);
    void leftsonarCallback(const std_msgs::Int16::ConstPtr& msg);
    void rightsonarCallback(const std_msgs::Int16::ConstPtr& msg);
    //node variables
    STATE state; //robot's state
    ros::NodeHandle nh; //ROS node handler
    ros::Publisher vel_pub; //publisher for /cmd_vel
    geometry_msgs::Twist vel_msg; //Twist message to publish
    ros::Subscriber sonar_sub; //subscriber for central sonar
    ros::Subscriber left_sonar_sub; //subscriber for left sonar
    ros::Subscriber right_sonar_sub; //subscriber for right sonar
    int dleft, dright, d;
    float SPIRAL; //FIND spiral tightness
};


// MAIN function.


int main(int argc, char **argv){
 //initialize ROS node
  ros::init(argc, argv, "wallfollow");
  //Create the object that contains node behaviour
  wallfollow wallfollow; //this calls the constructor method
  //keep ROS updating the callback functions until node is killed
  ros::spin(); //Go to idle, the callback functions will do everything
```

```
}



//Constructor. ROS initializations are done here.

wallfollow::wallfollow(){

  //subscribe to topics

  left_sonar_sub = nh.subscribe<std_msgs::Int16>("/arduino/sonar_3", 10,
&wallfollow::leftsonarCallback, this);   //Subscribe to left sonar

  sonar_sub = nh.subscribe<std_msgs::Int16>("/arduino/sonar_2", 10,
&wallfollow::sonarCallback, this);                 //Subscribe to centre sonar

  right_sonar_sub = nh.subscribe<std_msgs::Int16>("/arduino/sonar_1", 10,
&wallfollow::rightsonarCallback, this);  //Subscribe to right sonar

  //advertise topics to publish

  vel_pub = nh.advertise<geometry_msgs::Twist>("/cmd_vel", 10);  //Publish twist data to
move robot

  //init auxiliary variables

  state = FIND;  //Start spiralling to find wall

  SPIRAL = 0;

}



void wallfollow::sonarCallback(const std_msgs::Int16::ConstPtr& msg){   //Update centre
sonar data

d = msg->data;

}



void wallfollow::rightsonarCallback(const std_msgs::Int16::ConstPtr& msg){        //Update
right sonar data

dright = msg->data;

}
```

```
void wallfollow::leftsonarCallback(const std_msgs::Int16::ConstPtr& msg){ //The sonar
callback function collects sonar left data and prescribes movement based on all sonar
sensor data

  dleft = msg->data; //update left sonar distance

  //Evaluate FSM

  switch(state){

   case TRACK:

        if      ((dleft>1 && dleft<TRACKDVALUE) && (d<1 || d>DIST)){      //Follows wall if
DLEFT is between 0 and TRACKDVALUE

                float error = dleft - LEFTDIST;                  //Float can be pos or neg. Neg
error gives clockwise turn/Pos error gives anticlockwise turn. Error value gives magnitude
to turn speed

                vel_msg.linear.x = LINEAR_SPEED;  //Forward component of movement
whilst tracking

                vel_msg.angular.z = ANG_ERROR_CONSTANT * error;      //Angular
component to turn to follow wall. Should allow for smooth motion even on curved walls.

                vel_pub.publish(vel_msg);            //publish new movement instruction

                }

        else if((dleft<1 || dleft>TRACKDVALUE) && (d<1 || d>DIST) && (dright<1 || dright >
RIGHTDIST) ){ //if travelling on the outside of an object and suddenly losses dleft

                vel_msg.linear.x = LEFTCORNERMULT * LINEAR_SPEED; //Forward
component needed to get round the corner without lots of stopping and turning. Can
reduce linear speed via multiplier to tighten corner. DO NOT CHANGE LINEAR SPEED OR
ANGULAR_SPEED TO ADJUST THIS!

                vel_msg.angular.z = ANGULAR_SPEED;      //turn ANTICLOCKWISE to find
wall again

                vel_pub.publish(vel_msg);            //publish new movement instruction

                }

        else if((d>0 && d<DIST) || (dright>0 && dright<RIGHTDIST)){

                vel_msg.linear.x = 0;                          //zero linear component

                vel_msg.angular.z = -0.5 * ANGULAR_SPEED;        //CLOCKWISE if travelling
inside a object, turn right to follow corner

                vel_pub.publish(vel_msg);            //publish new movement instruction

                }
   break;


   case FIND: //go forward until a wall is found

        if((d>0 && d<DIST) || (dright>0 && dright<RIGHTDIST) || (dleft > 0 &&
dleft<TRACKDVALUE)) {                    //FOUND WALL! STOP and change state to track

                vel_msg.linear.x = 0;                    //zero linear component

                vel_msg.angular.z = 0;                           //zero angular component

                vel_pub.publish(vel_msg);                //publish new movement instruction

                state = TRACK;                                   //Change state to start tracking

                }

        else{

                vel_msg.linear.x = SPIRAL * LINEAR_SPEED;          //Spiral to find wall

                vel_msg.angular.z = -ANGULAR_SPEED;      //zero angular component

                vel_pub.publish(vel_msg);                //publish new movement instruction

                SPIRAL = SPIRAL + SPIRALMULTI;

                }

   break;

  }//end switch

}//end sonarCallback
```