Diploma in

Mechatronics Engineering

Internship Program Report

12 weeks

Title:

Internship Program

NYP Supervisor:

Dr. Edwin Foo

company Supervisor:

Dr. Edwin Foo

Dr. Michael Lau

Student:

LAI ZEYU (160081F)

Abstract

We decided to design a waiter robot to deliver food to the restaurant. In the kitchen, the chef can use the network to manipulate the robot to send food to different tables.

In this project we ended up using the same moving base as the social robots we made before. We have improved based on the original omnidirectional wheel to make the robot move more stable and accurate. And now, the robot can be moved to a specific location.

Contents

company

NewRIIS is Newcastle University's new cutting-edge research facility in Singapore.

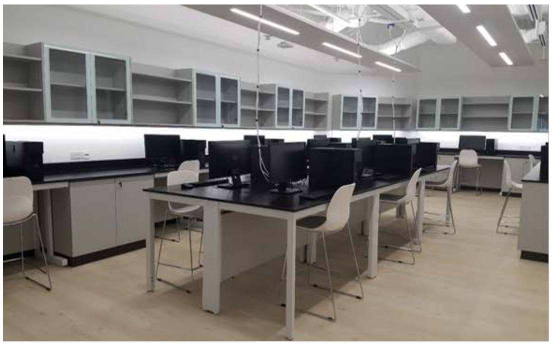
Research

you can see many different areas of research projects in NewRIIS. NewRIIS also builds on Newcastle University's established and excellent global research reputation. The University performs **high-quality research** across a wide range of disciplines and locations. Areas of particular focus for NewRIIS will include: Energy and Sustainability; Smart Grids, Process Safety, Marine Technology; Data Visualization and others.

Facilities

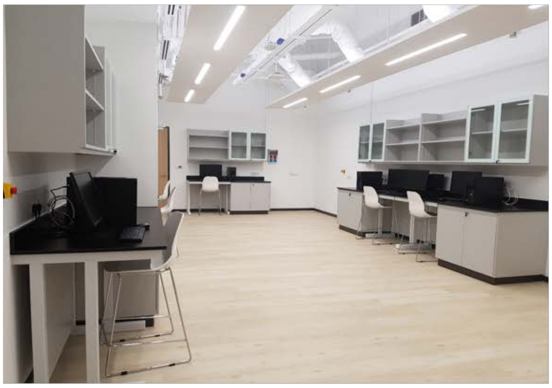
Located at the Devan Nair Institute for Employment and Employability in Jurong East, NewRIIS houses four research laboratories, a visualization suite, two 50-seat classrooms, open-plan research areas and seminar facilities.

- **Research laboratories** equipped with the latest technology and software for teaching
- **Visualization suite** including one of two Microsoft Surface Hubs for collaboration via cloud technology
- **Classrooms** for teaching, sharing research and opportunities to develop new ways of thinking
- **Open-plan research areas** designed to foster and encourage collaborative study amongst groups
- **Seminar facilities** for group discussion, presentations and facilities for international conferencing
- **Study booths** for private study and research, as well as breakout spaces for small groups



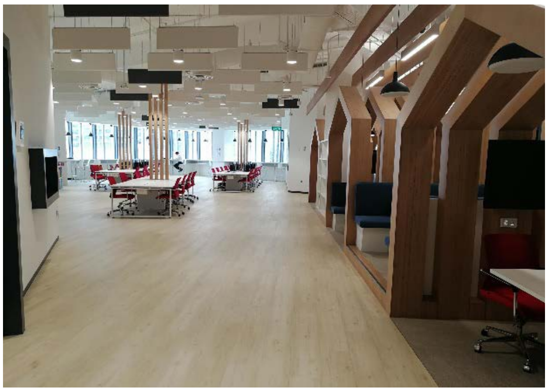
Computer Lab 1

Computer Lab 2



Energy Lab 1

Energy Lab 2



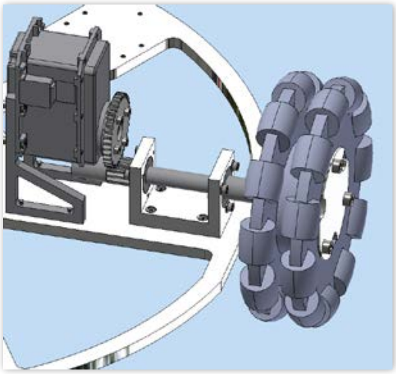
Background

Map maker

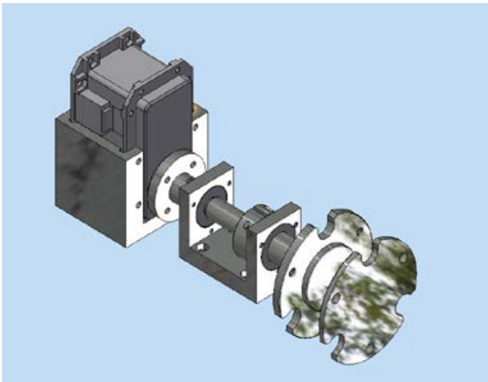
Map maker is used to draw the map for navigation, the original map maker used indigo, so we need change the system to kinetic to fit our other ros system.

waiter robot

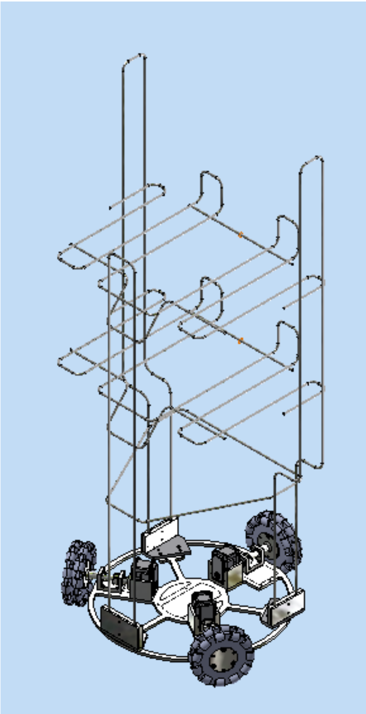
This waiter robot uses omnidirectional wheel move base and the Mechanical part design by haihong. The move base part is like social robots, we just change some part to make the robot move more stable and accurate



Before



now



The picture on the left is the structure of the entire robot, and the picture on the right is the hardware we use.

Due to structural problems, the radar cannot scan 360 degrees, but can scan 340 degrees after removing the occlusion. This is enough for omnidirectional movement.

Introduction

The base of ros navigation can see in my fyp report, in this report, I will introduce the improvements and additions.

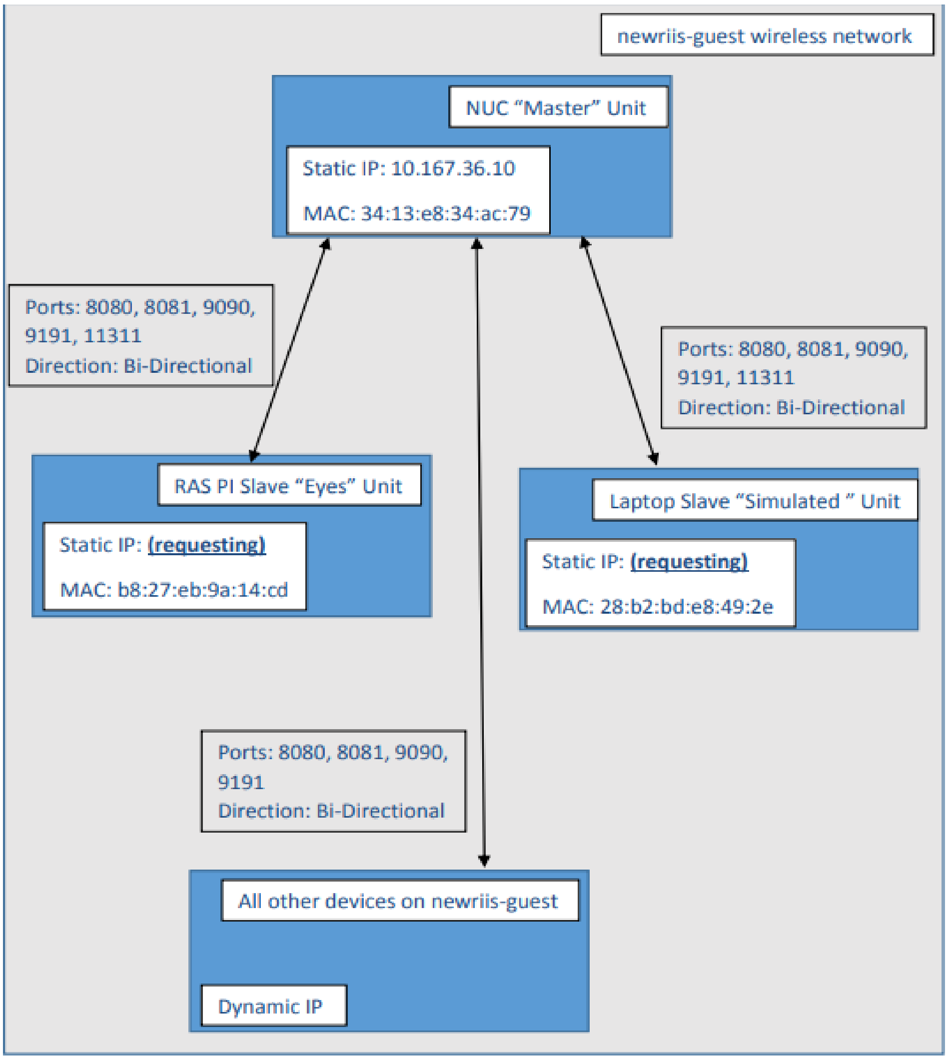
Mapping

Map maker

Map maker Map makers include raspberry pi, rplidar and a screen. system is ubantu mate which we get image file form turlebot , the ros system already can use we just need install the rplidar part and the hector mapping part.

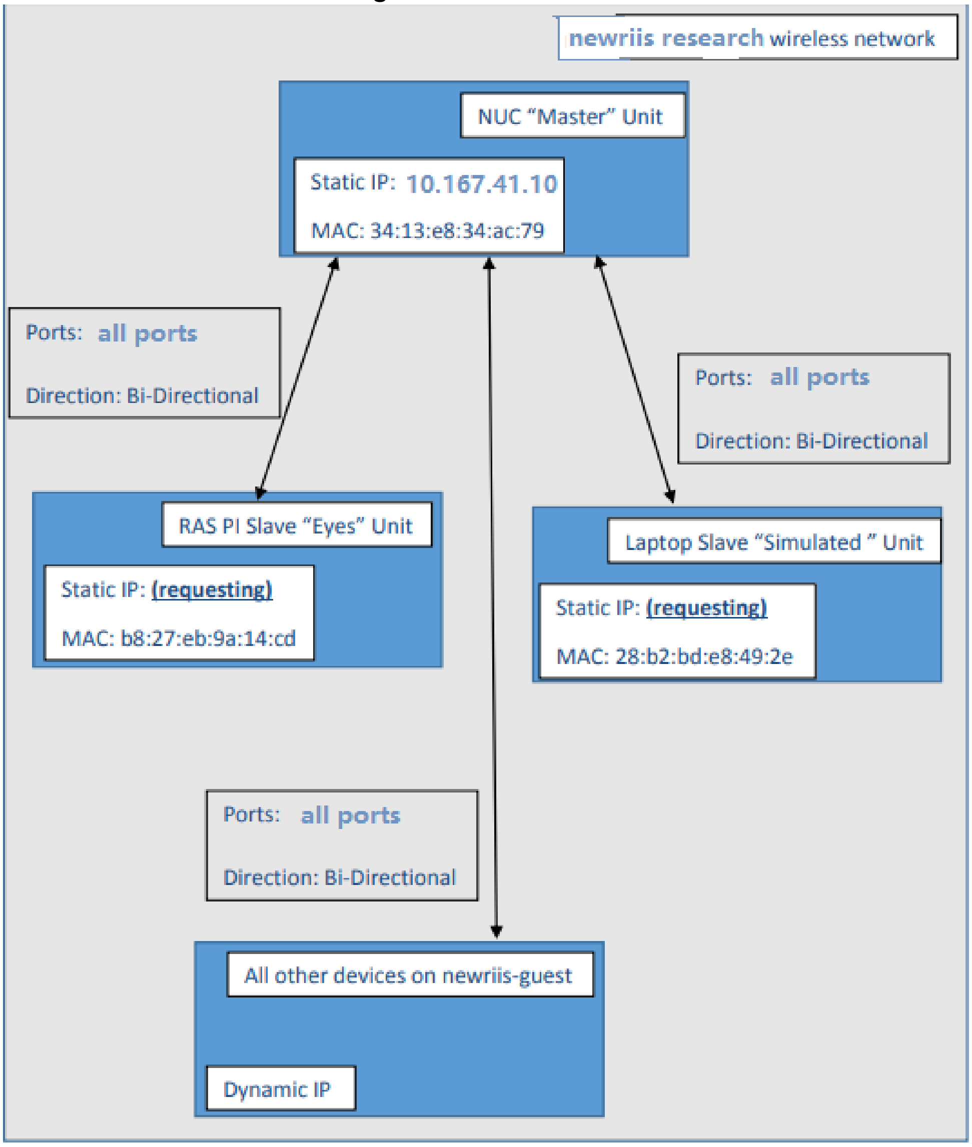
In map maker we use hector mapping not slamming Gmapping because the map maker not have move part so can't get velocity data in wheel and hector mapping only need scan data. The map obtained in this way may not be accurate, but it is enough to use it as the initial map for testing.

Setting up the network



The network of laboratories is directly connected to the network of the UK Newcastle university. Due to security considerations, all network ports in the lab were initially closed.
So, we contacted the British staff and wanted to open some ports. The above picture is the modified network port connection diagram.

After the setup is complete, the map maker can be connected to the nuc, but the robot cannot be controlled by the network. And such a setup may have network security issues, so the Newcastle university changed the way the network is connected. The figure below is the new network connection method.



Now a dedicated LAN is set up for the robot so that all network ports can be opened without security issues.

Set up ros master

We need set the bashrc file in nuc and raspberry pi.
In nuc:
Run:
gedit ~/.bashrc

Add the words in file:
export ROS_HOSTNAME=10.167.41.10

export ROS_MASTER_URI=http:// 10.167.41.11:11311
In raspberry pi:
Run:
gedit ~/.bashrc

Add the words in file:
export ROS_HOSTNAME=10.167.41.11
export ROS_MASTER_URI=http:// 10.167.41.11:11311

Now we can test the map maker.

In raspberry pi:
Run:
roslaunch rplidar_ros rplidar.launch
roslaunch rplidar_ros hector_mapping.launch

In nuc:
Run:
rosrun rviz

now if you see the map in the rviz means the map maker can work now.

Tuning

Base controller & Odometry source

Because the motor part change, so, I remove the gear ratio, adjusted wheel direction and tf deviation.

```
wheel_command.request.left_vel = (-((sqrt(3)) / 2)*(in_real_x_velocity) + (1.0/ 2)*(in_real_y_velocity) + (L )*(in_angular)) /(1*R);
wheel_command.request.right_vel = (((sqrt(3)) / 2)*(in_real_x_velocity) + (1.0/ 2)*(in_real_y_velocity) + (L )*(in_angular)) /(1*R);
wheel_command.request.center_vel = (-(in_real_y_velocity)  + (L )*(in_angular)) /(1*R);
```

```
int32_t VelocityControl::convertVelocity2Value(float velocity)
{
  //ROS_ERROR("convertVel2Value %f",velocity);
  return (int32_t) (velocity /1.0 * multi_driver_->multi_dynamixel_[MOTOR]->velocity_to_value_ratio_);//
}
```

```
if (multi_driver_->multi_dynamixel_[MOTOR]->model_name_.find("PRO") != std::string::npos)
{
 setVelocity(convertVelocity2Value(left_vel), convertVelocity2Value(-right_vel),convertVelocity2Value(center_vel));//
}
```

```
dynamixel_workbench_msgs::WheelCommandNew msg;
vel_ctrl.controlLoop();
msg.left_vel_new =-left_vel_c*1.0; //left_vel;
msg.right_vel_new = -right_vel_c*1.0; //right_vel;
msg.center_vel_new =-center_vel_c*1.0; //center_vel;
dynamixel_wheel_vel_pub_.publish(msg);
```

```
double linear_x_vel = 0.99*( -((vl ) / sqrt(3.0)) + ((vr ) / sqrt(3.0)));//(1.05)
double linear_y_vel = 1.0*(vr  + vl  -(2.0)*vc)/3.0;
double angular = 1.08*((vl / L) + (vr / L) + (vc / L))/3.0; //(1.115)
```

Sensor source

```
const size_t degree_90 = 90;
const size_t degree_270 = 270;
const size_t left_degrees = 180;
const size_t right_degrees = 180;


for (size_t i = 0; i < node_count; i++){
     scan_msg.ranges[i] = std::numeric_limits<float>::infinity();
 }

 for (size_t i = 0; i < node_count; i++)
 {
     float read_value = (float) nodes[i].distance_q2/4.0f/1000;
     if (i < right_degrees)
     {
        if(i>150&&i<160)scan_msg.ranges[2*degree_90 - i] = std::numeric_limits<float>::infinity();
        else
        {
           if (read_value == 0.0) scan_msg.ranges[2*degree_90 - i] = std::numeric_limits<float>::infinity();
           else
              scan_msg.ranges[2*degree_90 - i] = read_value;
           scan_msg.intensities[2*degree_90 - i] = (float) (nodes[i].sync_quality >> 2);
        }
     }
}
else if (i > left_degrees)
{
  if(i>203&&i<210) scan_msg.ranges[2*degree_270 - i] = std::numeric_limits<float>::infinity();
   else
   {
     if (read_value == 0.0) scan_msg.ranges[2*degree_270 - i] = std::numeric_limits<float>::infinity();
     else
        scan_msg.ranges[2*degree_270 - i] = read_value;
     scan_msg.intensities[2*degree_270 - i] = (float) (nodes[i].sync_quality >> 2);
   }
}
else
{
   //do nothing;
}
```

The function of each piece of code:

```
const size_t degree_90 = 90;
const size_t degree_270 = 270;
const size_t left_degrees = 180;
const size_t right_degrees = 180;
```

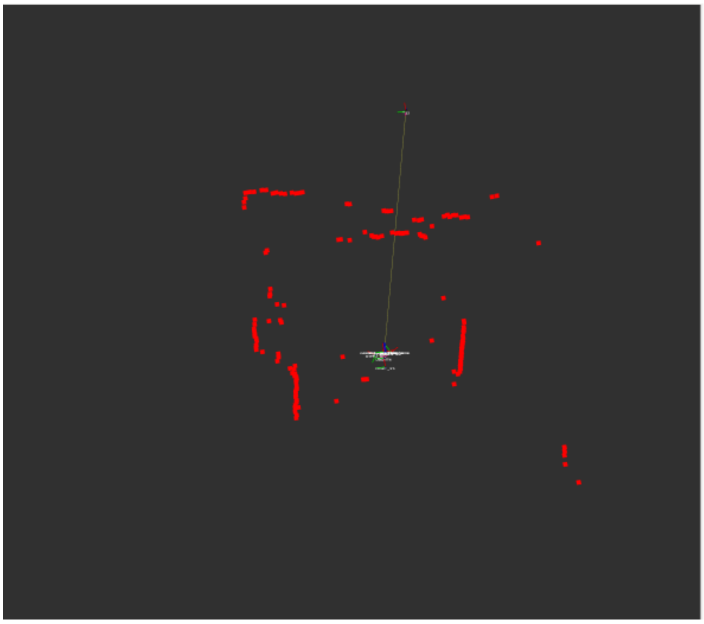
Set the scanned area to 360 degrees.

```
if (i < right_degrees)
{
   if(i>150&&i<160)scan_msg.ranges[2*degree_90 - i] = std::numeric_limits<float>::infinity();
   else
   {
      if (read_value == 0.0) scan_msg.ranges[2*degree_90 - i] = std::numeric_limits<float>::infinity();
      else
         scan_msg.ranges[2*degree_90 - i] = read_value;
      scan_msg.intensities[2*degree_90 - i] = (float) (nodes[i].sync_quality >> 2);
   }
}
```

The obstacle of the robot part around the lidar from 0 to 180 degrees is removed.

```
else if (i > left_degrees)
{
  if(i>203&&i<210) scan_msg.ranges[2*degree_270 - i] = std::numeric_limits<float>::infinity();
   else
   {
     if (read_value == 0.0) scan_msg.ranges[2*degree_270 - i] = std::numeric_limits<float>::infinity();
     else
        scan_msg.ranges[2*degree_270 - i] = read_value;
     scan_msg.intensities[2*degree_270 - i] = (float) (nodes[i].sync_quality >> 2);
   }
}
```

The obstacle of the robot part around the lidar from 180 to 360 degrees is removed.



Now robots can get almost all the obstacle information around them through the lidar.

Navigation launch & params

Most of the parameters are the same as the social robots, but to make the robots omnidirectional, we tried several different local planner.

Eband local planner:

In move_base.launch

```
<!-- move_base node -->
<node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <!--<param name ="/use_sim_time" value="true"/>-->
    <param name="controller_frequency"          value="10.0"/>
    <rosparam file="$(find nnaavvii)/parafiles/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find nnaavvii)/parafiles/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find nnaavvii)/parafiles/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/base_local_planner_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/eband_local_planner_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/dwa_local_planner_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/navfn_global_planner_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/globalplanner_params.yaml" command="load" />
<!--
    <rosparam file="$(find nnaavvii)/parafiles/move_base_params.yaml" command="load" />

-->


    <param name="global_costmap/robot_base_frame" value="base_link"/>
    <param name="global_costmap/global_frame" value="/map"/>
    <param name="local_costmap/inscribed_radius" value="0.32"/>
    <param name="local_costmap/circumscribed_radius" value="0.32"/>
    <param name="base_global_planner" value="navfn/NavfnROS"/>
    <!-- <param name="base_local_planner" value="base_local_planner/TrajectoryPlannerROS"/> -->
    <param name="base_local_planner" value="eband_local_planner/EBandPlannerROS"/>
    <remap from="cmd_vel"  to="/esther_velocity_controller/cmd_vel"/>
```

```
<rosparam  file="$(find  nnaavvii)/parafiles/eband_local_planner_params.yaml" command="load" />
```

Load the eband_local_planner_params

```
<param name="base_local_planner" value="eband_local_planner/EBandPlannerROS"/>
```

Use eband local planner become our using local planner.

In eband_local_planner_params.yaml :
*you can see how to tuning this params in:
http://wiki.ros.org/eband_local_planner

```
 xy_goal_tolerance: 0.1
 yaw_goal_tolerance: 0.05
 rot_stopped_vel: 0.01
 trans_stopped_vel: 0.01
 marker_lifetime: 0.5
 eband_min_relative_overlap: 0.7
 eband_internal_force_gain: 1.0
 eband_external_force_gain: 2.0
 num_iterations_eband_optimization: 3
 eband_equilibrium_approx_max_recursion_depth: 4
 eband_equilibrium_relative_overshoot: 0.75
 eband_significant_force_lower_bound: 0.15
 costmap_weight: 10.0

#Trajectory Controller Parameters
 max_vel_lin: 0.5
 max_vel_th: 0.2
 min_vel_lin: 0.1
 min_vel_th: 0.0
 min_in_place_vel_th: 0.0
 in_place_trans_vel: 0.0
 Ctrl_Rate: 10
 max_acceleration: 0.5
 virtual_mass: 0.75
 max_translational_acceleration: 0.5
 max_rotational_acceleration: 1.0
 rotation_correction_threshold: 0.5
 differential_drive: false
```

With this local planner, the machine can still perform small-range omnidirectional motion when the radar cannot get 360-degree scan information. This local planner has a good obstacle avoidance system that can avoidance all detected obstacles in the radar. This local plan is also good for sudden obstacles and has good velocity control.

However, since the late developers abandoned this program for the development and optimization of the omnidirectional wheel, this local planner is not fully applicable to this project. When using this local planner, you will find that most of the time it is still in a differential drive state.

DWA local planner:

In the last few days we determined that the radar could range from 340 degrees, I tried to use the full dwa local planner. Then I found that in this case, full omnidirectional movement can be achieved.

In move_base.launch:

```xml
<!-- move_base node -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <!--<param name ="/use_sim_time" value="true"/>-->
    <param name="controller_frequency"           value="5.0"/>
    <rosparam file="$(find nnaavvii)/parafiles/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find nnaavvii)/parafiles/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find nnaavvii)/parafiles/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/global_costmap_params.yaml" command="load" />
    <!--<rosparam file="$(find nnaavvii)/parafiles/base_local_planner_params.yaml" command="load" />-->
    <!--<rosparam file="$(find nnaavvii)/parafiles/eband_local_planner_params.yaml" command="load" />-->
    <rosparam file="$(find nnaavvii)/parafiles/dwa_local_planner_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/navfn_global_planner_params.yaml" command="load" />
    <rosparam file="$(find nnaavvii)/parafiles/globalplanner_params.yaml" command="load" />
<!--
    <rosparam file="$(find nnaavvii)/parafiles/move_base_params.yaml" command="load" />

-->


    <param name="global_costmap/robot_base_frame" value="base_link"/>
    <param name="global_costmap/global_frame" value="/map"/>
    <param name="local_costmap/inscribed_radius" value="0.32"/>
    <param name="local_costmap/circumscribed_radius" value="0.32"/>
    <param name="base_global_planner" value="navfn/NavfnROS"/>
    <!-- <param name="base_local_planner" value="base_local_planner/TrajectoryPlannerROS"/> -->
    <!--<param name="base_local_planner" value="eband_local_planner/EBandPlannerROS"/>-->
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS"/>
    <remap from="cmd_vel"  to="/esther_velocity_controller/cmd_vel"/>
```

We removed all other local planners, ensure that other parameters do not interfere with the DWA local planner. (In the original test, other parameters may interfere with the dwa local planner, so it is impossible to start navigation.)

in aammccll.launch:

```xml
<param name="odom_model_type"            value="omni"/>//diff

<param name="gui_publish_rate"           value="10.0"/>
<param name="laser_max_beams"            value="30"/>
<param name="laser_max_range"            value="6.0"/>
<param name="min_particles"              value="100"/>//500
<param name="max_particles"              value="1000"/>//2000
<param name="kld_err"                    value="0.05"/>
<param name="kld_z"                      value="0.99"/>
<param name="odom_alpha1"                value="0.2"/>
<param name="odom_alpha2"                value="0.2"/>
<!-- translation std dev, m -->
<param name="odom_alpha3"                value="0.8"/>
<param name="odom_alpha4"                value="0.2"/>
<param name="odom_alpha5"                value="0.2"/>
```

Make sure the position detection of the robot is in the omnidirectional state.

In dwa_local_planner_params.yaml:

```yaml
DWAPlannerROS:
  acc_lim_th: 2.0
  acc_lim_x: 2.0
  acc_lim_y: 2.0


  max_vel_x: 0.6
  min_vel_x: 0.0
  max_vel_y: 0.1
  min_vel_y: -0.1

  max_trans_vel: 1
  min_trans_vel: 0.1

  max_rot_vel: 0.8

  min_rot_vel: 0.1

  sim_time: 1.7
  sim_granularity: 0.1
  angular_sim_granularity: 0.1

  path_distance_bias: 32.0  # 32.0
  goal_distance_bias: 24.0
  occdist_scale: 0.01      # 0.01

  stop_time_buffer: 0.2
  oscillation_reset_dist: 0.05


  forward_point_distance: 0.325

  scaling_speed: 0.25
  max_scaling_factor: 0.2

  vx_samples: 3
  vy_samples: 6
  vtheta_samples: 6

  use_dwa: true


#xy_goal_tolerance: 0.2
#yaw_goal_tolerance: 0.17

  rot_stopped_vel: 0.01
  trans_stopped_vel: 0.01


  publish_traj_pc : true
  publish_cost_grid_pc: true
  global_frame_id: odom
  use_sim_time: true
```

acc_lim_th: 2.0
acc_lim_x: 2.0
acc_lim_y: 2.0

The acceleration value must be as large as possible, otherwise the machine will not move properly.

max_vel_x: 0.6
min_vel_x: 0.0
max_vel_y: 0.1
min_vel_y: -0.1

The maximum speed in the x direction must be much faster than the maximum speed in the y direction, otherwise the machine will run at the maximum speed in the middle. That would cause the robot not to face the front, but to form a 45-degree angle between the front and the front of the robot.
The minimum value in the y direction is a negative number, otherwise it cannot move in the negative direction in the y direction.

max_trans_vel: 1
min_trans_vel: 0.1

max_rot_vel: 0.8
min_rot_vel: 0.1

min_trans_vel and min_rot_vel must more then 0, otherwise the robot may not move properly.

sim_time: 1.7

sim_time need between 1.5 and 2.0.

vx_samples: 3
vy_samples: 6
vtheta_samples: 6

The sample should not be too much, otherwise the information of the map may not be transmitted fast enough, which may cause the program to not work properly.

right now, This local planner can be fully omnidirectional, but the values still need to be adjusted.

Joystick contorl

The Joystick now has new features.

Now can use RB button to stop navigation.

```
nav_stop:
  type: topic
  message_type: actionlib_msgs/GoalID
  topic_name: /move_base/cancel
  deadman_buttons: [5]
  message_value:
    -
      target: id
      value: ''
```

If you press the RB button, they will send the cancel goal massage to the move base part to stop the navigation.

Now can use start button to set the start point.

```
nav_satrt:
  type: topic
  message_type: geometry_msgs/PoseWithCovarianceStamped
  topic_name: /initialpose
  deadman_buttons: [7]
  message_value:
    -
      target: header.frame_id
      value: 'map'
    -
      target: pose.pose.position.x
      value: -5.613
    -
      target: pose.pose.position.y
      value: 1.341
    -
      target: pose.pose.orientation.z
      value: -0.005
    -
      target: pose.pose.orientation.w
      value: 0.999
    -
      target: pose.covariance
      value: [0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
```

You can find all the pose data in topic pub_pose, and you can copy the data to this file. After set all data you can use start button to set the start point. But if you start navigation you can’t press the start button, because it will reset the pose and all data will wrong.

Point to point navigation

I use a json dictionary and a python file to do the point to point navigation

I show these two files first.

nav_array.py:

```
#!/usr/bin/env python
import rospy
import actionlib
import json
import roslib
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose, PoseWithCovarianceStamped, Point,
Quaternion, Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from random import sample
from math import pow, sqrt
from std_msgs.msg import String


name = ""
class NavTest():
	def __init__(self):
		def callback(msg):
			global name
			name = msg.data
			#rospy.loginfo('%s' %name)
		rospy.init_node('nav_test', anonymous=True)
		rospy.on_shutdown(self.shutdown)

		# How long in seconds should the robot pause at each location?
		self.rest_time = rospy.get_param("~rest_time", 2)

		# Are we running in the fake simulator?
		self.fake_test = rospy.get_param("~fake_test", False)

		# Goal state return values
		goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED','SUCCEEDED',
				'ABORTED',		'REJECTED','PREEMPTING',
'RECALLING',
				'RECALLED','LOST']
```

```
        # Set up the goal locations. Poses are defined in the map frame.
        # An easy way to find the pose coordinates is to point-and-click
        # Nav Goals in RViz when running in the simulator.
        # Pose coordinates are then displayed in the terminal
        # that was used to launch RViz.
        filename =
'/home/esther_base/catkin_ws/src/rbx2/rbx2_gui/pose.json'
        with open(filename) as f:
          pose_data = json.load(f)

          locations = dict()
          datadict = dict()
          for pose_dict in pose_data:
                point_name = pose_dict['point_name']
                point_type = pose_dict['point_type']
                point_id = pose_dict['point_id']
                position_x = pose_dict['position_x']
                position_y = pose_dict['position_y']
                orientation_z = pose_dict['orientation_z']
                orientation_w = pose_dict['orientation_w']

                locations[str(point_name)] = Pose(Point(float(position_x),
float(position_y), 0.000),
                                        Quaternion(0.000, 0.000,
float(orientation_z), float(orientation_w)))
                datadict[point_id] = point_name,point_type


        # Publisher to manually control the robot (e.g. to stop it)
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=5)

        # Subscribe to the move_base action server
        self.move_base = actionlib.SimpleActionClient("move_base",
MoveBaseAction)
        rospy.loginfo("Waiting for move_base action server...")

        # Wait 60 seconds for the action server to become available
        self.move_base.wait_for_server(rospy.Duration(60))
        rospy.loginfo("Connected to move base server")

        # A variable to hold the initial pose of the robot to be set by the user in
RViz
        initial_pose = PoseWithCovarianceStamped()
```

```
        # Variables to keep track of success rate, running time, and distance
traveled
        n_locations = len(locations)
        n_successes = 0
        i = n_locations
        distance_traveled = 0
        start_time = rospy.Time.now()
        running_time = 0
        location = ""
        last_location = ""
        poselist={}
        posename = ''
        patrol = False
        action = False
        # Get the initial pose from the user
        rospy.loginfo("press    the    start  button  on  the  handle    to  start
navigation...")
        print(datadict)
        rospy.wait_for_message('initialpose', PoseWithCovarianceStamped)
        self.last_location = Pose()
        rospy.Subscriber('initialpose',               PoseWithCovarianceStamped,
self.update_initial_pose)
        # Make sure we have the initial pose
        while initial_pose.header.stamp == "":
            rospy.sleep(1)
        rospy.loginfo("Starting navigation test")

        # Begin the main loop and run through a sequence of locations
        while not rospy.is_shutdown():

            if patrol == True:
              patrol =   True
              # If we've gone through the current sequence, start with a new
random sequence

         # Increment the counters
              i += 1
              poselist = datadict.get(str(i))
              posename = poselist[0]
              action = True
              rospy.loginfo("action start")
         if i == n_locations-1:
         i = 0
         sequence = sample(locations, n_locations)
```

```
# Skip over first location if it is the same as the last location
patrol    = False
      else:
          rospy.loginfo("waiting goal1")
          rospy.wait_for_message('chatter', String)
          rospy.Subscriber('chatter', String, callback)
          rospy.loginfo('%s' %name)
          if str(name) == "":
                          rospy.loginfo("waiting goal")
          elif str(name) == "patrol" :
                      patrol    =    True
    i =1
                      poselist = datadict.get(str(i))
                      posename = poselist[0]
                      rospy.loginfo("action start1")
                      action = True
          elif locations.has_key(str(name))    ==    True:
                      posename = str(name)
                      rospy.loginfo("action start2")
                      action = True
          else:
                      rospy.loginfo("action error")

      if action == True:
          action =    False
          #location = locations.get(posename)

          # Keep track of the distance traveled.
          # Use updated initial pose if available.
          if initial_pose.header.stamp == "":
                  distance = sqrt(pow(locations[posename].position.x
                                        - locations[last_location].position.x, 2) +
                                        pow(locations[posename].position.y -
                                          locations[last_location].position.y, 2))
          else:
                  rospy.loginfo("Updating current pose.")
                  distance = sqrt(pow(locations[posename].position.x
                                        - initial_pose.pose.pose.position.x, 2) +
                                        pow(locations[posename].position.y -
                                        initial_pose.pose.pose.position.y, 2))
                  initial_pose.header.stamp = ""

          # Store the last location for distance calculations
          last_location = posename
```

```
            # Set up the next goal location
            self.goal = MoveBaseGoal()
            self.goal.target_pose.pose = locations[posename]
            self.goal.target_pose.header.frame_id = 'map'
            self.goal.target_pose.header.stamp = rospy.Time.now()

            # Let the user know where the robot is going next
            rospy.loginfo("Going to: " + str(posename))
            # Start the robot toward the next location
            self.move_base.send_goal(self.goal)

            # Allow 5 minutes to get there
            finished_within_time = self.move_base.wait_for_result(rospy.Duration(600))

            # Check for success or failure
            if not finished_within_time:
                self.move_base.cancel_goal()
                rospy.loginfo("Timed out achieving goal")
            else:
                state = self.move_base.get_state()
                if state == GoalStatus.SUCCEEDED:
                    rospy.loginfo("Goal succeeded!")
                    n_successes += 1
                    distance_traveled += distance
                else:
                    rospy.loginfo("Goal failed with error code: " + str(goal_states[state]))

            # How long have we been running?
            running_time = rospy.Time.now() - start_time
            running_time = running_time.secs / 60.0

            # Print a summary success/failure, distance traveled and time elapsed
            rospy.loginfo("Running time: " + str(trunc(running_time, 1)) +
                          " min Distance: " + str(trunc(distance_traveled, 1)) + " m")
            rospy.sleep(self.rest_time)
        else:
            rospy.loginfo("no action")
```

```python
    def update_initial_pose(self, initial_pose):
        self.initial_pose = initial_pose

    def shutdown(self):
        rospy.loginfo("Stopping the robot...")
        self.move_base.cancel_goal()
        rospy.sleep(2)
        self.cmd_vel_pub.publish(Twist())
        rospy.sleep(1)
def trunc(f, n):

    # Truncates/pads a float f to n decimal places without rounding
    slen = len('%.*f' % (n, f))
    return float(str(f)[:slen])

if __name__ == '__main__':
    try:
        NavTest()
        rospy.spin()
    except rospy.ROSInterruptException:
        rospy.loginfo(" navigation test finished.")
```

Pose.json:


```json
[
    {
        "point_name": "kitchen",
        "point_type": "startpoint",
        "point_id": "0",
        "position_x": "-5.613",
        "position_y": "1.341",
        "orientation_z": "-0.005",
        "orientation_w": "0.999"
    },
    {
        "point_name": "tableone",
        "point_type": "goalpoint",
        "point_id": "1",
        "position_x": "0.389",
        "position_y": "1.432",
        "orientation_z": "-0.713",
```

```
        "orientation_w": "0.701"
    },
    {
        "point_name": "tabletwo",
        "point_type": "goalpoint",
        "point_id": "2",
        "position_x": "6.42",
        "position_y": "1.25",
        "orientation_z": "-0.732",
        "orientation_w": "0.681"
    },
    {
        "point_name": "tablethree",
        "point_type": "goalpoint",
        "point_id": "3",
        "position_x": "12.384",
        "position_y": "1.180",
        "orientation_z": "-0.735",
        "orientation_w": "0.678"

    },
    {
        "point_name": "tablefour",
        "point_type": "goalpoint",
        "point_id": "4",
        "position_x": "11.913",
        "position_y": "-14.708",
        "orientation_z": "0.678",
        "orientation_w": "0.734"
    },
    {
        "point_name": "tablefive",
        "point_type": "goalpoint",
        "point_id": "5",
        "position_x": "6.761",
        "position_y": "-14.709",
        "orientation_z": "0.711",
        "orientation_w": "0.703"
    }
]
```
In this json dictionary, each group have seven data , we can use “point_name” to find a point, and use “point_type” know how we use the point ,and use “point_id” to set the patrol order. The last four data is used to tell the navigation system location information

```
        locations = dict()
        datadict = dict()
        for pose_dict in pose_data:
            point_name = pose_dict['point_name']
            point_type = pose_dict['point_type']
            point_id = pose_dict['point_id']
            position_x = pose_dict['position_x']
            position_y = pose_dict['position_y']
            orientation_z = pose_dict['orientation_z']
            orientation_w = pose_dict['orientation_w']

            locations[str(point_name)] = Pose(Point(float(position_x), float(position_y), 0.000),
                                              Quaternion(0.000, 0.000, float(orientation_z), float(orientation_w)))
            datadict[point_id] = point_name,point_type
```

Get all the data from the dictionary, change dictionary to list , convert the format to the format required by subsequent programs..

```
        self.move_base = actionlib.SimpleActionClient("move_base", MoveBaseAction)
        rospy.loginfo("Waiting for move_base action server...")
```

Check if move base is activated.

```
        n_locations = len(locations)
        n_successes = 0
        i = n_locations
        distance_traveled = 0
        start_time = rospy.Time.now()
        running_time = 0
        location = ""
        last_location = ""
        poselist={}
        posename = ''
        patrol = False
        action = False
```

Initialize all data

```
        rospy.wait_for_message('initialpose', PoseWithCovarianceStamped)
        self.last_location = Pose()
```

Make sure the initial position has been set.

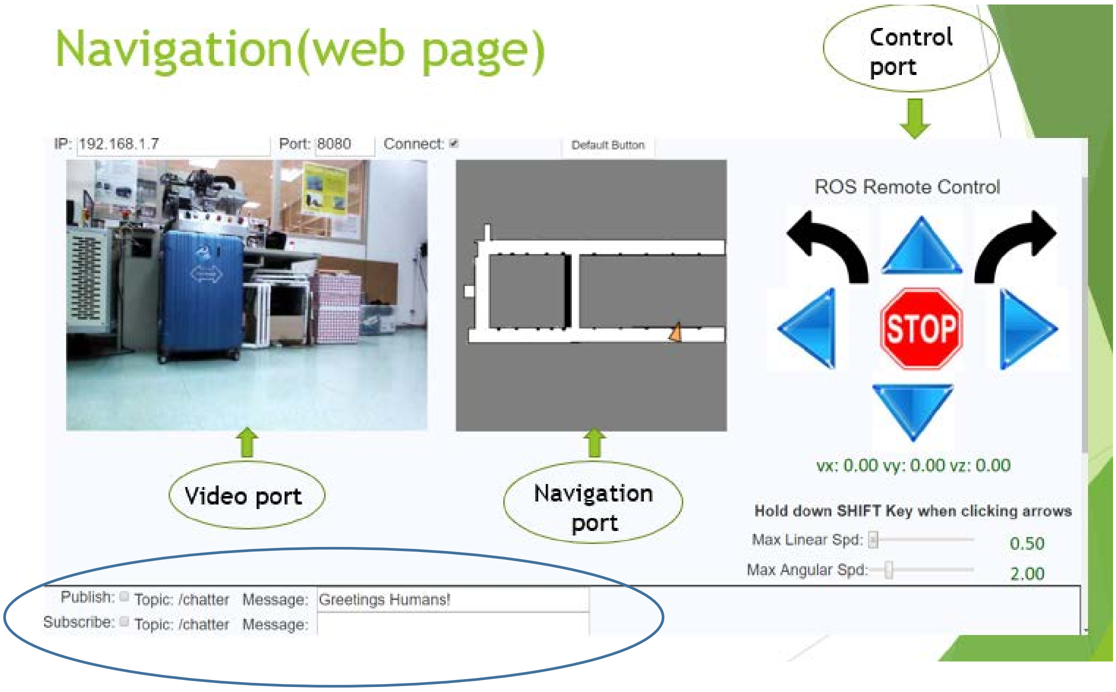
```
while initial_pose.header.stamp == "":
            rospy.sleep(1)
        rospy.loginfo("Starting navigation test")

        # Begin the main loop and run through a sequence of locations
        while not rospy.is_shutdown():

            if patrol == True:
              patrol =    True
              # If we've gone through the current sequence, start with a new
random sequence

         # Increment the counters
              i += 1
              poselist = datadict.get(str(i))
              posename = poselist[0]
              action = True
              rospy.loginfo("action start")
         if i == n_locations-1:
         i = 0
         sequence = sample(locations, n_locations)
         # Skip over first location if it is the same as the last location
         patrol    = False
            else:
              rospy.loginfo("waiting goal1")
              rospy.wait_for_message('chatter', String)
              rospy.Subscriber('chatter', String, callback)
              rospy.loginfo('%s' %name)
              if str(name) == "":
                      rospy.loginfo("waiting goal")
              elif str(name) == "patrol" :
                    patrol    =    True
           i =1
                    poselist = datadict.get(str(i))
                    posename = poselist[0]
                    rospy.loginfo("action start1")
                    action = True
              elif locations.has_key(str(name))    ==    True:
                    posename = str(name)
                    rospy.loginfo("action start2")
                    action = True
              else:
                    rospy.loginfo("action error")
```

```
if action == True:
   action =    False
```

Make sure there is an input location. If the input information is patrolling, patrol is performed. If the input information is the name of the coordinates in the dictionary, navigate to the corresponding coordinates. If the information is wrong, wait for the correct information.

Web server

Use the same web page as the social robot, but you can navigate to the specified location by entering information.



Fill in the name of the location at the input location , and check the box to control the robot to the right location.

Conclusion

Through this internship, I optimized the previous move base, studied different local planners, and learned how to adjust parameters with different robots.
I also learned how to communicate with classmates and teachers to complete a project.
As a programmer, we must be willing to learn and to be flexible with the way we think. I learnt there may be more than one solution to every problem or error. Communication and teamwork is important. Whenever face with any difficulties, we should help one Another. We must be willing to accept criticisms and be open to suggestions and opinions in order to improve.
Our supervisor was willing to help whenever I seek help from him even though he was busy with his own task and was on-site performing testing most of the time. Whenever I faced any problem, he gave advices on how to solve it. Were very friendly and helpful as well.
I am very grateful to my two supervisors for helping me during my internship.