Lab 6 PID controller for robot navigation

In the previous Labs, you have been taught the following knowledge:

- how to use odometry data to control the robot moving from home to the charger through two gaps.
- how to collect and store odometry and laser data for generating their data graphs.

It was an open control strategy as the robot is given a number of constant velocities and turning angles at several waypoints. ***You may save your C/C++ code file which you have practiced during the last 4 lab sessions, and create a new file for this lab.***

Task 1: In this lab, you should create PID controllers to control the simulated robot moving from home to the charger in an environment shown in Figure 7.1.

- You should write C/C++ code to implement PID controllers in a ROS environment.
- Your code should drive the robot to go through two gaps and stop at the charger position.
- You should adjust the parameters of PID controllers to make the trajectory smooth.
- The collected odometry and laser data should be saved into files for generating 3 graphs, (trajectory, velocity and the laser mapping) to be included into your Assignment report.

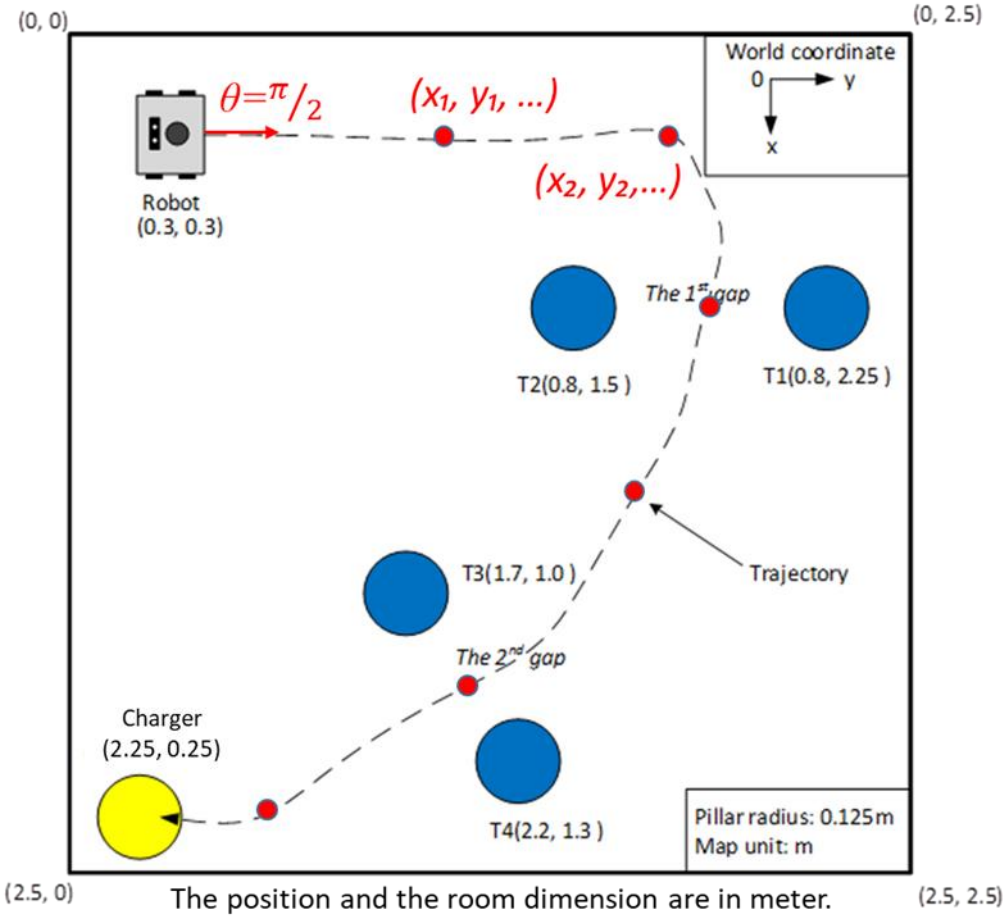
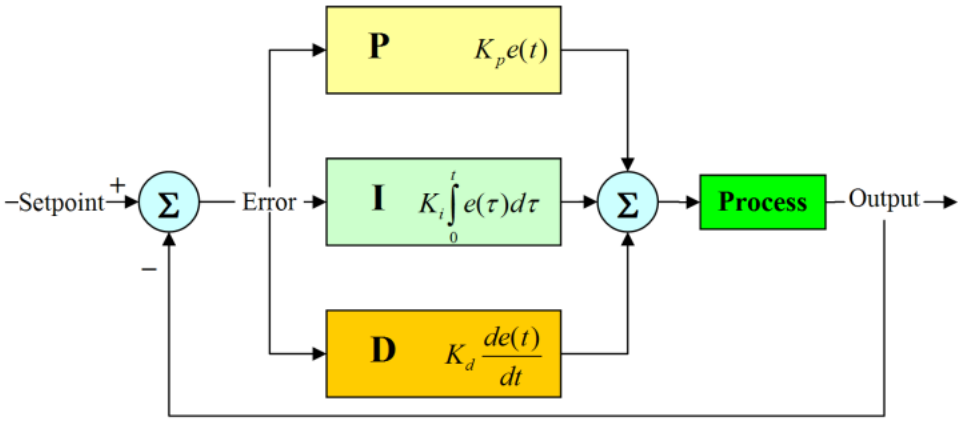


Figure 6.1 The navigation task

6.1 Basic concept

PID control algorithm is most widely used in industrial automation and commercial electronic products. It consists of three parts as shown below.



Output = kp* e + ki* ei+ kd* ed

- Set the initial PID parameters: **kp** = 0.01, **ki** = 0.001, **kd** = 0.001
- Set the initial values: **ei_pre = 0, ed_pre = 0.**
- Calculate errors of the PID controller: **err**, **ei**, **ed.**
  err = setValue – measuredValue
  ei = **ei_pre** + **err**
  ed = err – ed_pre
  ei_pre = **ei**
  ed_pre = ed

6.2 Design of PID controller

You should create a number of PID based robot behaviours for the robot to move from home to the charger, which are shown in YELLOW text below. Note that you may create different numbers of robot behaviours to implement the specified navigation tasks.

```
// The following lines of code is used to control the robot motion
void RobotMove::startMoving(){
    ros::Rate rate(20);   //Define rate for repeatable operations.
    ROS_INFO("Start moving");
    while (ros::ok()){ // keep spinning loop until user presses Ctrl+C
      switch(stage){
      case 1: // the robot is following the wall
          if (PositionY < landmark1)
              // create "the PID-based wall following behaviour"
          else stage = 2;
          break;
      case 2: // the robot turns right into the 1st gap
           if (PositionX < landmark2)
```

```
                // create "PID-based going through the 1st gap behaviour"
            else stage = 3;
            break;
        case 3: // the robot moves toward the 2nd gap
            if (PositionX < landmark3)
                // create "PID-based moving toward the 2nd gap behaviour"
            else stage = 4;
            break;
        case 4: // the robot going through the 2nd gap
            if (PositionX < landmark4)
                // create "PID-based going through the 2nd gap behaviour"
            else stage = 5;
            break;
        case 5: // the robot moves towards the charger
            if (PositionX < landmark5)
                // create "PID-based reaching goal behaviour"
            else stage = 6;
            break;
        case 6: // stop at the charger position
            moveStop();
            break;
        }
       // some existing code
        ros::spinOnce(); // Allow ROS to process incoming messages
        rate.sleep();   // Wait until defined time passes.
    }
    odomTrajFile.close();
    odomVelFile.close();
    laserFile.close();
    laserMapFile.close();
    ROS_INFO("Closing files");
}
```

6.3 Create PID based wall following behaviour

To follow the wall, you need use Laser data, i.e. *leftRange,* to monitor how far the robot is from the wall. *leftRange* will be used as the input of the controller, and compared with the targeted range, i.e. *landmark1_toWall = 0.3*, to generate the turning control value so that the robot can follow the wall well.

You should define some new variables and function in the existing code, which are highlighted in YELLOW below.

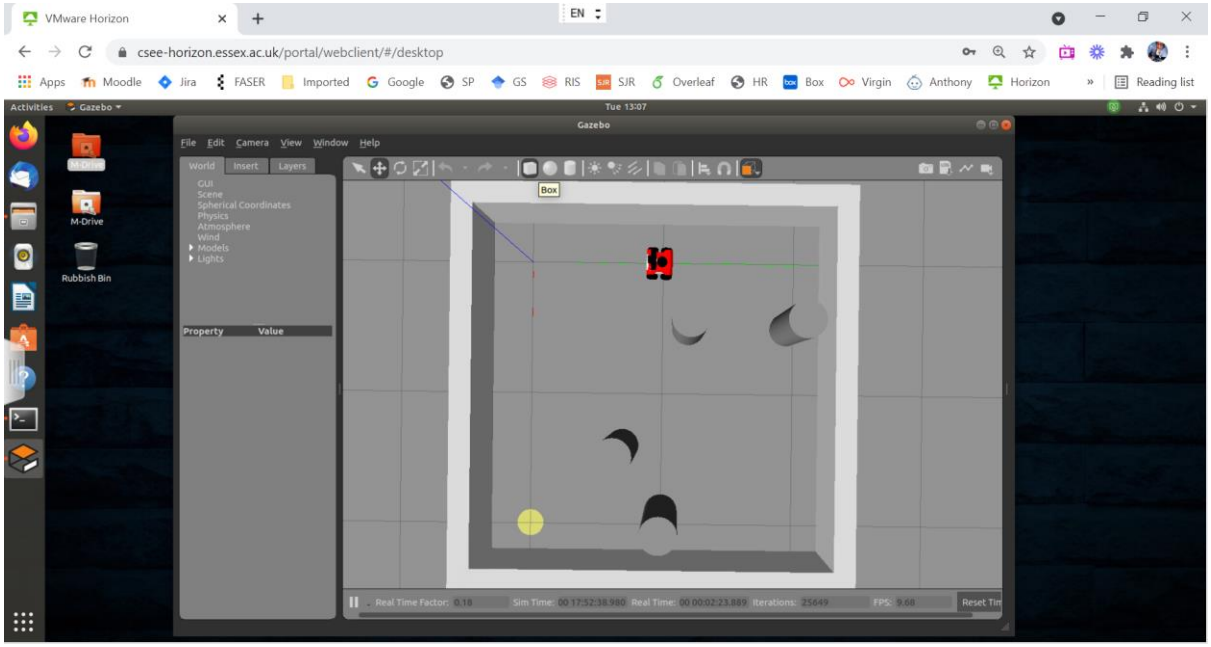
```
class RobotMove{
public:
        // some existing code……
        void PID_wallFollowing(double moveSpeed, double laserData);
private:
        // some existing code……
        double kp1 = 0.01, ki1 = 0.001, kd1 = 0.001, ei_pre1 = 0;
        double ed_pre1 = 0, landmark1_toWall = 0.3, Max_PID_outout = 0.6;
}

void RobotMove::PID_wallFollowing(double moveSpeed, double laserData)
{
        double ei, ed, err, output;
        err = laserData – landmark1_toWall;
        ei = ei_pre1 + err;   ed = err – ed_pre1;
        ei_pre1 = ei;  ed_pre1 = ed;
        output = kp1*err + ki1*ei + kd1*ed;
        if (output > Max_PID_outout )
            output = Max_PID_outout;
        else if(output < -Max_PID_outout)
            output = -Max_PID_outout;
        geometry_msgs::Twist msg;
        msg.linear.x = moveSpeed;
        msg.angular.z = output;
        commandPub.publish(msg);
}

void RobotMove::startMoving(){
        // some existing code
        while(ros::pk()){
            switch(stage){
            case 1: // wall following
                if(PositionY < landmark1)
                    PID_wallFollowing(FORWARD_SPEED_HIGH, leftRange);
                else stage = 6;
                break;
            case 2:

                // some existing code

             case 6: // stop at the charger position
                moveStop();
                break;
            }
```

```
        // some existing code
        ros::spinOnce(); // Allow ROS to process incoming messages
        rate.sleep();   // Wait until defined time passes.
    }
    odomTrajFile.close();
    odomVelFile.close();
    laserFile.close();
    laserMapFile.close();
    ROS_INFO("Closing files");
}
```

Run your code to test your PID controller and adjust its parameters: firstly **kp,** then **ki**, to improve the wall following performance until the smooth trajectory is achieved.

```
cd  ~/ros_workspace
catkin_make
source devel/setup.sh
roslaunch tutorial_pkg tutorial_rosbot.launch
```

After running your code, the robot will stop at the point you set. The display of Simulator looks like the picture below.



6.4 Create PID based going through the 1st gap behaviour

Now you need to create PID-based going through the 1st gap behaviour. Note that when the robot is in the middle of the 1st gap, its heading should be 0. Therefore, you need to set the targeted heading as *landmark2_heading=0.* By using the odometry data, the PID controller will output the turning commands based on the difference between the measured heading and the target heading *landmark2_heading*.

To make the robot pass through the 1st gap, you need to define new function and variables, which are highlighted in YELLOW below.

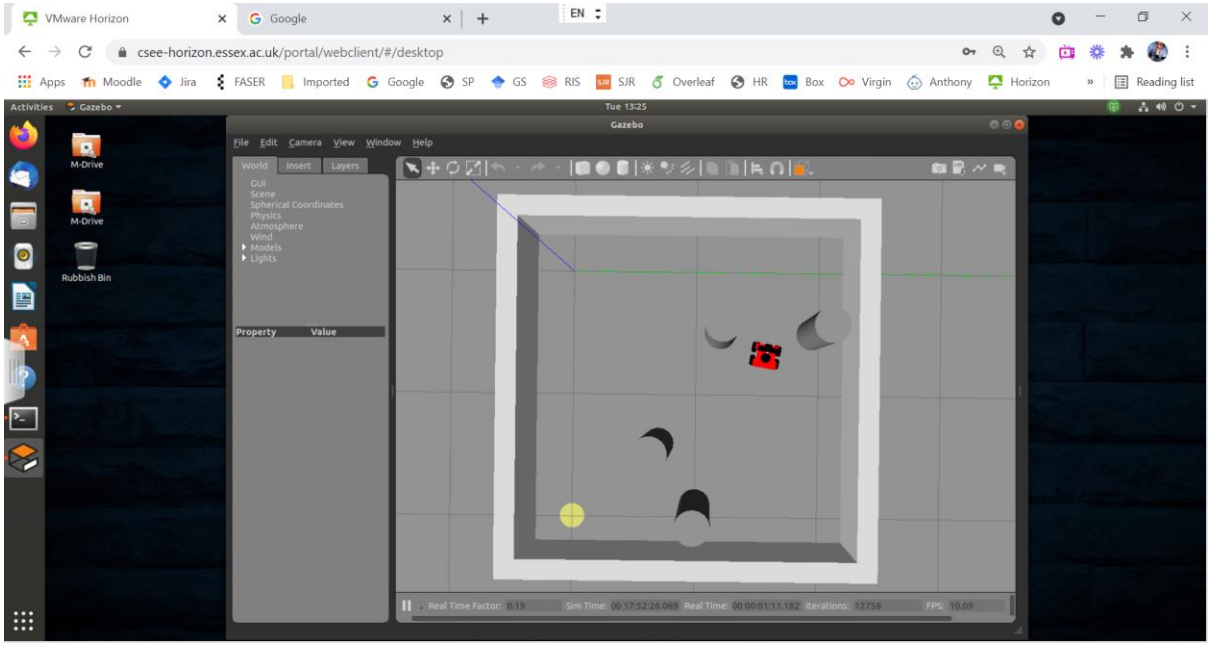
```
class RobotMove{
public:
        // some existing code……
        void PID_to1stGap(double moveSpeed, double robotHeading);
private:
        // some existing code……
        double kp2=0.01, ki2=0.001, kd2=0.001, ei_pre2 = 0;
        double ed_pre2 = 0, landmark2_heading=0;
}

void RobotMove::PID_to1stGap(double moveSpeed, double robotHeading)
{
        double ei, ed, err, output;
        err = landmark2_heading - robotHeading;
        ei = ei_pre2 + err;   ed = err – ed_pre2;
        ei_pre2 = ei;  ed_pre2 = ed;
        output = kp2*err + ki2*ei + kd2*ed;
        if (output > Max_PID_outout )
            output = Max_PID_outout;
        else if(output < -Max_PID_outout)
            output = -Max_PID_outout;
        geometry_msgs::Twist msg;
        msg.linear.x = moveSpeed;
        msg.angular.z = output;
        commandPub.publish(msg);
}

void RobotMove::startMoving(){
        // some existing code
        while(ros::pk()){
            switch(stage){
            case 1: // wall following
                if(PositionY < landmark1)
                    PID_wallFollowing(FORWARD_SPEED_HIGH, leftRange);
                else stage = 2;
                break;
             case 2: // move toward the middle of the 1st gap
                if(PositionX < landmark2)
                   PID_to1stGap(FORWARD_SPEED_MIDDLE, robotHeadAngle);
                else stage = 6;
                break;
```

```
        case 3: // move toward the 2nd gap

            // some existing code
            // ……..

        case 6: // stop at the charger position
            moveStop();
            break;
        }

        // some existing code
        ros::spinOnce(); // Allow ROS to process incoming messages
        rate.sleep();   // Wait until defined time passes.
    }
    odomTrajFile.close();
    odomVelFile.close();
    laserFile.close();
    laserMapFile.close();
    ROS_INFO("Closing files");
}
```

Run your code to test your PID controller and adjust its parameters: firstly **kp,** then **ki**, to improve the wall following performance until the smooth trajectory is achieved.

```
cd ~/ros_workspace
catkin_make
source devel/setup.sh
roslaunch tutorial_pkg tutorial_rosbot.launch
```

After running your code, the robot should be stopped at the point you set, as shown below.



6.5 Create remaining robot behaviours

Now, you should create two remaining robot behaviours so that the mobile robot can be controlled to move toward the 2nd gap, go through it, and finally reach the charger position to complete the navigation task.

To make the mobile robot pass through the 2nd gap smoothly and not too close to the pillars, you should adjust PID parameters and the *landmark3_heading* value to improve the robot performance.

After the mobile robot has reached to the charger position, you should use "Control^C" to stop the program running. Then, you can use the saved odometry and laser data to create the velocity and trajectory graphs, as well as the laser map for including into your Assignment report.