TECHIN 516: Robotic Sensing And Mobility

Lab 3: Physical Implementations and Introduction to Control

DATE: Jan 26th. 2023

Lecturer: Queenie Qiu, John Raiti. Student: ***Shucheng Guo, Harry Tung, Joshua Sanchez***

1 Sensor inaccuracies in the physical robot

1.1 Connecting to the physical robot

Connecting to the physical robot according to the instructions in the slides.

1.2 Errors in odometry in real world operation

In this section, we will use the patrol example that we have explored in the Lab1 again so as to compare the odometry performance between simulation and real world implementation. We will use the nodes and code created in Lab1 to visualize data from the /odom topic.

Deliverables:

1. Run the patrol example with the following parameters: square length 0.8m and 1 iteration. In an additional terminal run the plot_odom.py file.

```
$ roslaunch turtlebot3_example turtlebot3_client.launch
$ rosrun turtlebot3_example turtlebot3_server
```

   (a) Make notes about robot behaviors: do the wheels turn at the same rate when moving forward? Were the 90 degree turns accurate? Does the robot return to the starting position?

   Answer: The wheels would rotate at the same rate when moving forward without the impact of other factors, e.g. low battery, facing obstacles.
   The turns, however, were far from accurate. The robot wouldn't make perpendicular turns in square patrols, not to mention gradual ones in circular turns.
   As expected, the robot would never return to its original position with only the joint states from odometry in real world.

   (b) Plot the x-y position of the robot against the "ground truth" and take a screenshot.

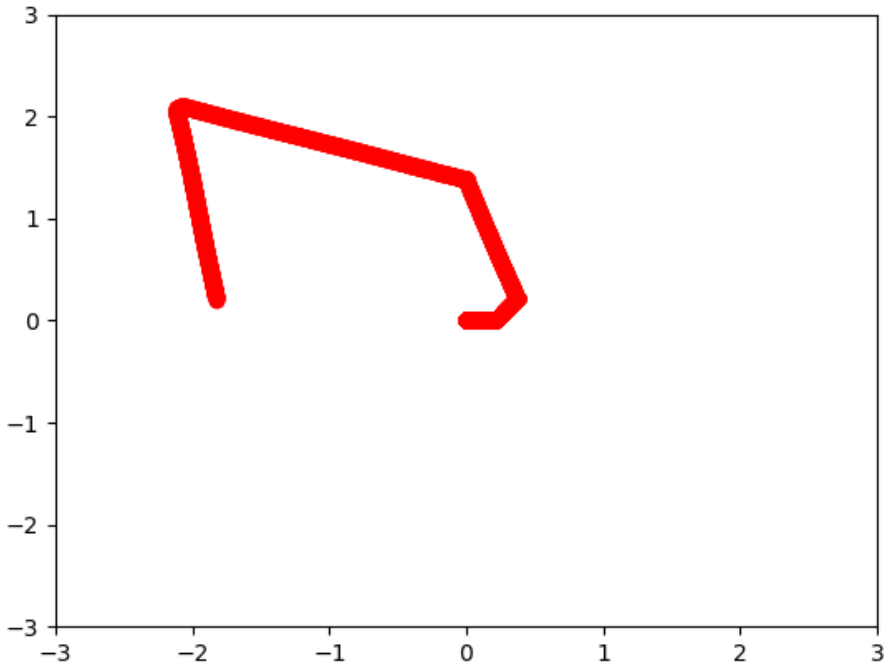


Figure 1: Plotting a square robotic patrol with the command `s 0.8 1`.

(c) Compare the graph to the one obtained in Lab1. Mention differences. Are odom errors larger or smaller? Mention what can be influencing the results.

Answer: Compared to the graph from Lab1, which is based on simulation, this real-world odometry-supported robot obviously performed much worse. The underperforming areas include the errors in its turns, differences between start and stop points, and the ability in walking at an even pace.
It's assumed that the context made a huge difference. When simulated, the environment, in which the route is planned, is nearly ideal. But in the real world, more things factor in when the odometry is sensoring information. It needs more input from more types of sensors to jointly plan for the route.

2. What can be done to improve the accuracy of sensor data or to get better estimation of robots' positions and orientations?

Answer: We can calibrate the sensors against ground truth, combine passive sensors and active ones, and prime the robot with a static map.

1.3 Errors in laser readings in real world operation

In this section, we will go through the mapping procedure followed in Lab2, and store the /scan topic data into a rosbag. We will compare the gmapping results in the real world against the ones obtained in the simulation.

1. Make sure the physical turtlebot is operational and placed correctly, in the environment you want to map. Move the robot to the right bottom corner of the map.

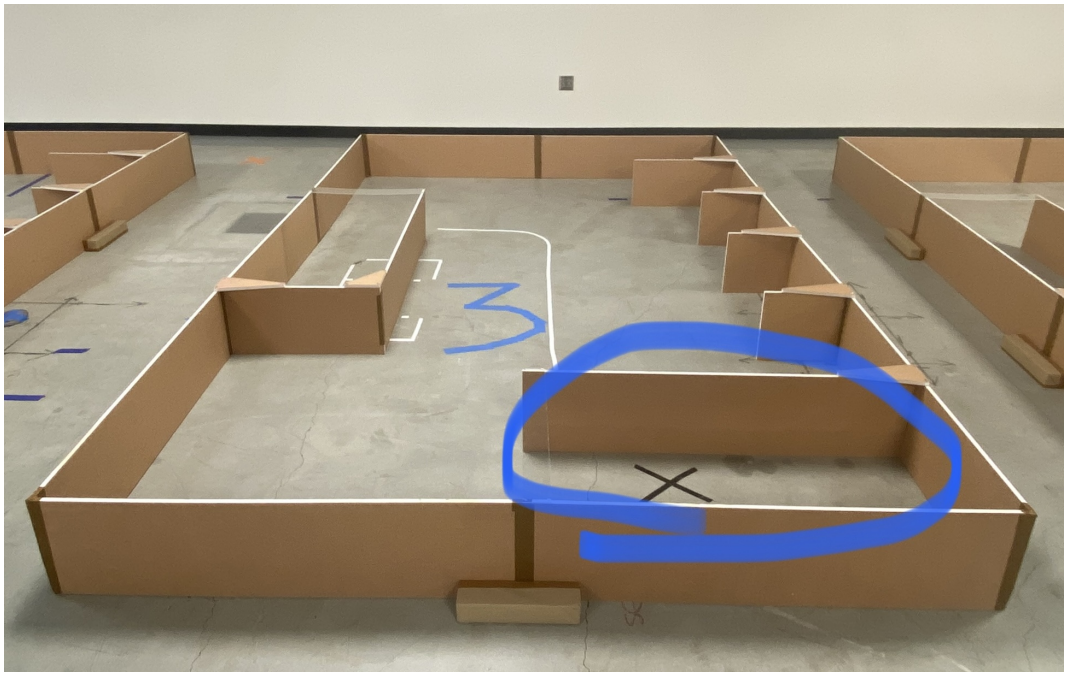


Figure 2: Start Position Indication

As a reminder the command line to launch the mapping is as follows:

```
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

Note: you will need to run the teleoperation program to go through the map.

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

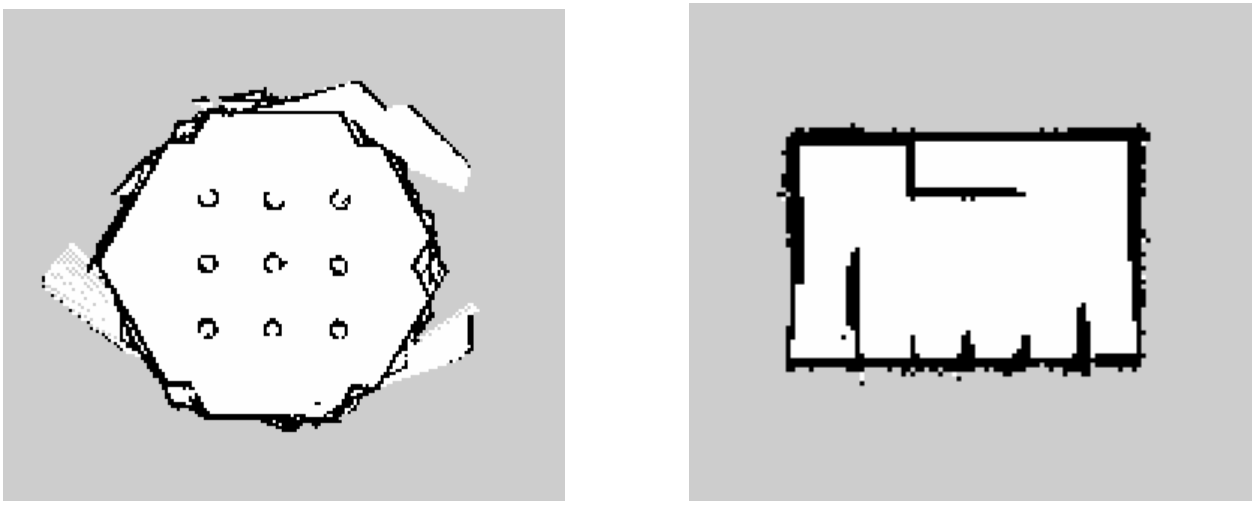
2. Additionally, before you start teleoperating the robot, record a rosbag file with the following topics.

```
$ rosbag record /odom /scan /cmd_vel /tf -O physicaltb3_map.bag
```

3. When you have finished tracing the surroundings to complete the map, you can stop the bag file recording and save the resulting map by running the following command in the terminal before finishing the gmapping launch.

```
$ rosrun map_server map_saver -f ~/gix_map
```

Deliverables:



(a) Digitally simulated map of the hexagon area.

(b) Physically implemented map of the GIX setup.

Figure 3: Comparing two maps achieved in different contexts.

1. Inspect the resulting files of the maps: the one obtained in the simulation and the one from the physical implementation. Attach screenshots of both pgm files. Mention differences between the results obtained in the simulation and the real world in terms of: thickness of the edges, additional shapes outside of the intended map area, differences in parameter values found in the .yaml file.

   Answer: Between the two maps sensored and generated by the robot, there are similarities and differences. The edges detected in the physical map of GIX are significantly thicker than the ones in the hexagon. There are more extraneous shapes detected and marked in the simulated map, while the physical implementation is more clear-cut. This could be personal, since not everyone has the messy simulation like mine. The parameter values, especially the occupancy and free thresholds, are identical in both .yaml files.

2. Use that map you just created and launch the navigation example in Lab2. Return the robot to the start position on the physical map. You may want to use that spot to give a 2DPoseEstimate as your starting position.

   ```
   $ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file
   :=$HOME/gix_map.yaml
   ```

   (a) Set a 2DNavGoal on the map that is close to this area.

Figure 4: 2DNavGoal Position Indication

What happens when a 2D navigation goal is provided? How long does it take for your robot to plan and reach the goal?

Answer: After a goal is set, the robot takes some time to plan its path, adjusts its direction, walks the route, makes turns and avoids the obstacles along the way.
If it bumps into obstacles, both in map and in reality, the path will be replanned for a couple of times before an official failure due to possible collisions.
For our robot, it took less than one minute to arrive at the goal on average. The time could be, however, significantly longer if the destination is chosen poorly.

(b) How close did the robot get to the goal (distance from the goal)?

Answer: The robot was very close to the goal, with an error of several inches, if there was any.

(c) Discuss your observations. These should include quantitative and qualitative components. You may rely on comparisons based on the concentric rings shown around the goal, or you may reference your comparisons to the robot's interpretation of the world.

Answer: Based on my observations of 10 robotic patrols, I found the alignment between maps essential. Since there is an error in the starting position, the robot has to be manually put in a position away from the cross that aligns both maps, or it will fail. The failure rates accounted for roughly 50% of the attempts, for reasons such as misaligned maps, dynamic obstacles and tough destinations. To avoid such failures, it's recommended to place the robot and select a target relatively in the middle of the lane to save enough space for route planning.

The time for the robot to reach the target also varied a lot, from as fast as 45s to as slow as 2min. When it detected obstacles at the start or end points, it would replan the route over and over before it could work, or just not move at all. Around 20% of the times, when the destination was poorly selected, the program would report too many errors until it declared a failure of potential collisions.

3. Submit your recorded rosbag with your lab assignment.

   Answer: The recorded bag file is attached in the assignment.

2 Introduction to control strategies in robot navigation

In this section, we will explore different paradigms to control the movements of a robotic platform. In this case, we will focus on the changes in position and orientation of a wheeled robot. For convenience, we will return to the simulation setting. This means that we will set our environment variables to its previous values of localhost in the .bashrc file.

2.1 Executing robot motions in open-loop control

Open-loop refers to the robot's operation created by the input signal does not depend on the system's output. Specifically, we will create commands for the robot to move forward 1.5 meters.

1. Launch a simulated turtlebot3 in the stage_1 world.

   ```
   $ roslaunch turtlebot3_gazebo turtlebot3_stage_1.launch
   ```

2. By knowing the distance to be traveled, we can determine the constant speed and the amount of time that constant speed needs to be held to cover said distance. Use the command-tool rostopic to publish velocity commands to the robot:

   ```
   $ rostopic pub -1 /cmd_vel geometry_msgs/
   Twist  '{linear:  {x: <value>, y: 0.0, z: 0.0}, angular
   : {x: 0.0, y: 0.0, z: 0.0}}'
   ```

   After a certain amount of time (pos = vel * time), you will publish a twist message to effectively stop the robot from moving:

   ```
   $ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{linear
   :  {x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}'
   ```

   For your convenience, we have prepared a simple python script that submits the commands to the terminal by using the os library. You will need to complete your selected values for time and speed.

Deliverables:

1. Report the chosen strategy (published values) to complete the task of moving 1.5m forward in open-loop.

Answer: For the command to publish velocity, I chose to apply `0.15` to the linear velocity of `x`, and stopped right after the topic published for 10 seconds, which made the expected distance to travel $1.5m$ (0.15×10.0).

2. Inspect the turtlebot3's position in Gazebo. (World -> Models -> turtlebot3_burger -> pose -> x). Did the robot move 1.5m accurately?

   Answer: It didn't move exactly $1.5m$ forward, but a bit shorter than that, around $1.48m$, and slightly to the left or right. Depending on the speed or linear velocity, the error between expected and realistic distances could be smaller or greater.

3. Compare your desired traveled value of 1.5m with the actual position of the robot in gazebo, and the reported value in the /odom topic (use rostopic echo /odom). Discuss differences.

   Answer: As mentioned in the last question, the actual travel of the robot is slightly longer than expected and tilted to the left or right.
   The pose information as reported in `Gazebo` and actual location as published in `/odom` mostly agree with each other. However, since the messages are published more precisely and more frequently in `/odom`, the pose information remains dynamic and decreases with time, though very slightly.

4. Mention the challenges of operating the robot in open-loop, particularly when the motions increase in complexity.

   Answer: Compared to the closed-loop control system, open-loop lacks accuracy and versatility. Due to a lack of feedback structure, they produce inaccurate outputs, and they can't correct the output automatically. As a result, they cannot adapt to the variations in environmental conditions or external disturbances.
   Open-loop can be used in simple applications, since they are low in cost and easy to implement. But with the increase of motion complexity, there is a need for human monitoring for better outputs.

5. How would the sequence of commands look if you wanted to complete a patrol pattern (e.g. a triangle) in open-loop? Report the solution in pseudocode, commands should include GoForward(distance), WaitTime(time), Rotate(angle in degrees).

   ```
   Rotate(30)
   GoForward(0.5)
   WaitTime(3s)
   Rotate(120)
   GoForward(0.5)
   WaitTime(3s)
   Rotate(120)
   GoForward(0.5)
   WaitTime(3s)
   ```

   Answer: Above is the pseudocode for a patrol of a equilateral triangle in the first quadrant with the bottom side sitting on the x-axis. The argument passed to the `GoForward` function

is the velocity, times `WaitTime` to get the distance to walk. The argument passed to `Rotate` dictates the degrees to turn right.

6. Modify the given open loop file to achieve this patrol motion.

   Answer: The modified python script is attached in the submission, and agrees with the above pseudocode.

7. How close to the start location did your robot finish? Use values from both: odometry and gazebo pose.

Pose	Attempt 1		Attempt 2	
	Gazebo	*Odometry*	*Gazebo*	*Odometry*
x	1.089103	1.0875463	-1.592979	-1.5919701
y	-1.068893	-1.0695119	-0.358039	-0.3571131
z	-0.001002	-0.0010022	-0.001002	-0.0010016

   Answer: In all of my attempts, the robots ended up very far from the designated target. The ending poses didn't follow a pattern that could be induced, but rather were randomly scattered in the whole space. In the table, two of the representative attempt were chosen and recorded.

2.2 Executing robot motions in closed-loop control

In closed-loop operation, the robot's input signal depends on a reference value that we want the output to be, and the comparison against the robot's current output through sensor feedback. We will create commands for the robot to move forward 1.5 meters by using the readings of some of its sensors, namely odometry and laser.

1. Create a node that subscribes to the /odom topic and publishes to the /cmd_vel topic. It will read the initial /odom value and only stop publishing commands to the velocity topic when the current value increases by 1.5m. Use the file close_loop_odom.py as a starting point.

2. Create a node that subscribes to the /scan topic and publishes to the /cmd_vel topic. It will read the initial /scan value corresponding to the front of the robot and only stop publishing commands to the velocity topic when the current scan value decreases by 1.5m. Use the file close_loop_laser.py as a starting point.

Deliverables: Run your closed-loop nodes to move the robot 1.5 meters forward and compare the performance of using /odom vs. /scan. Also check the robot's position in Gazebo through the plotting utility and compare to answer following questions:

1. How do each closed-loop control compare to the open-loop performance?

Pose	*Open: Time + Vel*		*Closed: Odometry*		*Closed: Laser Scan*	
x	1.465127	1.472632	1.559412	1.588417	1.528828	1.556617
y	-0.013769	0.075753	0.001973	0.007014	0.005801	0.005815
z	-0.001002	-0.001002	-0.001002	-0.001002	-0.001002	-0.001002

Answer: Compared to the open-loop robot patrol, the closed-loop ones utilizing sensored data were slightly better in performance, but quite inconsistent. Based on the limited data, controlling with laser scanned data was more accurate in path planning, but controlling with merely time and velocity wasn't good for nothing. It remains unknown why sometimes performances of closed-loops would be worse, and it calls for us to conduct more experiments on whether obstacles or robot types makes a difference.
From the table, it can be seen that using sensors to move $1.5m$ forward, whether `/odom` or `/scan`, yields better results in maintaining the original angle – walking in straight line. Determining only time and velocity for the robot, without the use of sensors, in fact controls better the distance to walk. It's worth mentioning choosing different linear velocities lead to distinct results, since the faster speed, the more inertia.

2. Record a video of the robot's performance for each closed-loop control node. Attach your video links.

   Google Drive link to the robotics motions by closed-loop and open-loop controls

3 Extra credit: Simulated Wall-following behavior

Now that we have completed motion using open and closed-loop control, we will go further into closed-loop by adding a controller step. Instead of using the error between the difference of an output measurement of our system and a specified reference value, the controller will use this error as input and output values to be used directly by the system. In our particular case of the simulated turtlebot3, we will use the readings from the laser sensor and a reference value of 0.5m to keep the robot driving parallel to a wall in the environment. We will provide code that you will need to complete, where a controller will use the error in the distance to the wall to output values from a PID controller that will modify the /cmd_vel values

1. Download the wall_follower and PID python files from the course materials and add them to your workspace (e.g. turtlebot3/turtlebot3_example/nodes folder). You will need to use the chmod +x <filename> command to make both files executables.

   ```
   $ rosrun turtlebot3_example wall_follower.py
   ```

2. Complete the code by selecting the correct topics to subscribe and publish, making sure that the readings from the lidar sensor are used correctly (you are interested in keeping a wall to the right side of the robot at a 0.5 distance at all times, this maps to a specific position in the ranges array).

3. Launch a simulated turtlebot3 burger in the empty world in Gazebo. Change the initial position of the turtlebot3.

```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch x_pos
:=-19.0 y_pos:=0.0
```

4. Go to the Insert model tab in Gazebo's left panel and add a "Grey Wall". Go into the Model Editor to modify the geometry of the link in both visual and collisions tab (X = 20.0). Set the pose of the new long grey wall to x=-0.8, y=0.0

5. Run (rosrun) the wall_follower node. We will change the values in the main function of the file to modify the PID controller parameters and observe performance. Every time you terminate the launch you need to publish a /cmd_vel message with zeros to stop the turtlebot3. You also need to reset the world poses in Gazebo (Ctrl+Shift+R). This step may have to be done several times before you see the robot in its original pose.

Deliverables:

1. Run the plotting node (plot_odom.py) to visualize the change of the error value over time with the different PID parameters values and save the plots for each case. You can also use the Gazebo plotting Utility and export the resulting plot.

   Answer: Refer to the 12 subplots in Figure 5 of the robotic responses to various PID parameter settings. The x-axis represents the robot's x coordinate, or horizontal position in the 2D space. The y-axis represents the difference between the expected and actual distances between its right side and the wall.

   NOTICE: The pose settings of the models in the Gazebo world are changed for successful execution of the program and analysis of the plots. Specifically, the original position of the robot is `(-10, 0, 0)`, while the wall is `(0, -0.5, 0)`, with a default difference of $0.5m$, exactly the distance to keep.

2. Describe what is happening in each of the 6 plots you create. For example: time for error to go to zero, is it oscillating, is there stationary error?

Case	Kp	Ki	Kd
No PID	1	0	0
Only P	1.2	0	0
PI	1.2	0.1	0
PD	1.1	0.0	1.5
PID	1.2	0.25	1.5
Custom (find your own PID values)			

Figure 6: Parameters for PID.

For the custom values of the PID, you should aim to have a response that goes fast to the

desired reference, has little or no oscillation in the response and its response is centered on the desired response (no stationary error)

Answer: For the required sets of PID values, with my pose settings, the robot is consistently unstable as the oscillations become larger with time. Despite no stationary errors, the existing errors don't go away, creating growing oscillations, and the robot may even be out of control before the path ends. This applies to plots a through f, along with h and j. Alternatively speaking, tuning only P and I doesn't make a difference.

After raising D to 10, with different PI settings, the robot gets more stable, and it takes only 10 seconds to clear the errors. As can be seen in g, i and k, tuning P and I with a constant D doesn't make a noticeable difference.

Based on the observations, increasing D significantly is expected to boost the performance and bring down the time it takes to rule out the errors. On the contrary, it creates a stationary error of around 1, after some instability like drifting.

In conclusion, for this specific case, the derivative term has more control over the behavior of the robot. Increasing it to a reasonable range dampens the oscillations and eliminates the errors, but a stationary error will be introduced beyond that range.

3. Compare the performance of the six controllers mentioned in the previous table in terms of: a. Time it takes to reach the desired goal (error=0). b. Peak oscillation value. c. Stationary error (value at which the robot settles and moves forward).

   Answer: In general, the three parameters in PID (Proportional, Integral, Derivative) impacts on the performance of the robot in walking in a straight line. Take the oscillation: P is related to its amplitude, I to its frequency and D to its damping.

   (a) For subplots other than g, i and k where D equals `10`, the oscillations get larger, or unstable, which means they never reach the desired goal of zero error.

   (b) The absolute values of the peaks in the oscillations are the errors, which mostly range from `0.3` to `0.5`. However, in better performances, the errors are kept close to zero.

   (c) For stable controllers,

4. Describe some of the challenges you faced with this wall following task and the steps you took to overcome them.

   Answer: A couple of challenges came up in the experiments, the root causes of which can be categorized and analyzed as follows:

   - Configuring the Gazebo models. To be honest, the difficulties that took most of the time were actually all related to the more insignificant issues. For example, it took me two days to find out if the robot was walking forward or backward; it took me even more time to correct the original pose settings of both models, which didn't work in the beginning. Some of these issues could be resolved by consulting others who are more

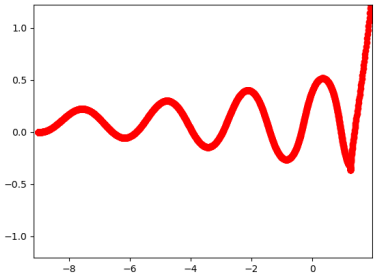
familiar with using Gazebo, while some others by reading some online documentations and discussions, or experimenting with diverse options.

- Implementation of the wall following algorithm. Understanding how the algorithm works was not easy either, especially when the idea isn't taught in class. One of the examples is finding the correct indices of `msg.ranges` at different angles, including `right` and `right_ahead_theta`, for distance calculation. Part of the reason for the failure is not knowing the robot's default orientation, leading to completely opposite results; another part is figuring out what the theta angle refers to here, and its difference with other wall follower solutions. The ways I tackle this issue include printing debug messages and testing different options, and comparing the codes with other algorithms.
- Tuning of the PID controller. Since the auto-tuning technique isn't available for this assignment, it has to be done by hand on a trial and error basis. At first, with the required sets of parameters, my robot is consistently unstable, as can be seen in the oscillations getting larger. The steps I took to obtain the correct damping are:
  (a) pinpoint the issue by consulting the TAs about what could've cause the problem;
  (b) understand the principles of PID by reading the theories and hypothesize possible combinations that could work
  (c) perform initial tuning by controling the variables and setting extreme values, e.g. $D = 100$
  (d) analyze the plots and generalize the impacts of each parameter in PID
  (e) perform fine tuning by selecting better sets of values that maximizes the trend

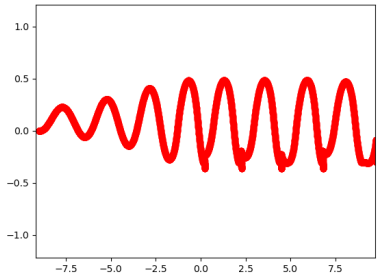
5. Mention at least 2 applications where PID controllers would be useful to a robotic implementation (e.g. navigation)

   Answer: A proportional–integral–derivative controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. The controller continuously calculates an error value $e(t)$ as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction.

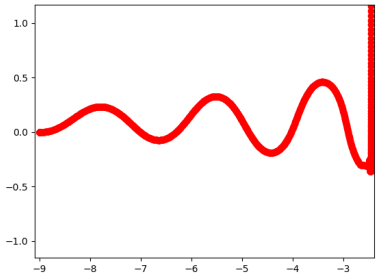
   In theory, a PID controller can be used to control any process that has a measurable output (PV), a known ideal value for that output (SP), and an input to the process (MV) that will affect the relevant PV. Controllers are used in industry to regulate temperature, pressure, force, feed rate, flow rate, chemical composition, weight, position, speed, and practically every other variable for which a measurement exists.



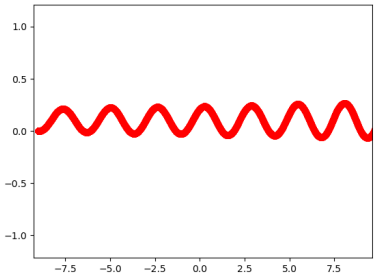
(a) No PID: `P=1, I=0, D=0`



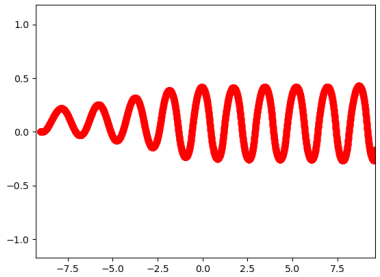
(b) Only P: `P=1.2, I=0, D=0`



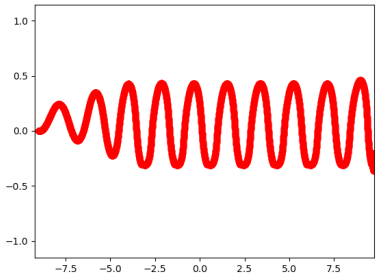
(c) PI: `P=1.2, I=0.1, D=0`



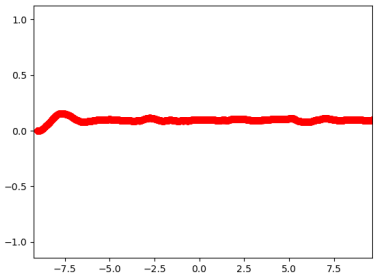
(d) PD: `P=1.1, I=0, D=1.5`



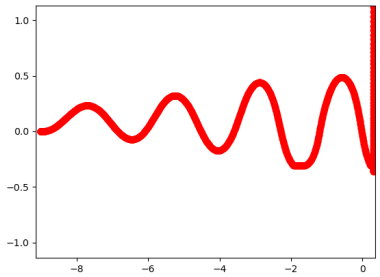
(e) PID: `P=1.2, I=0.25, D=1.5`



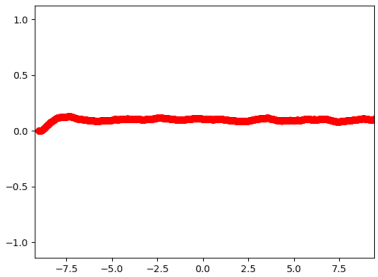
(f) PI:`P=1.2, I=0.25, D=0`



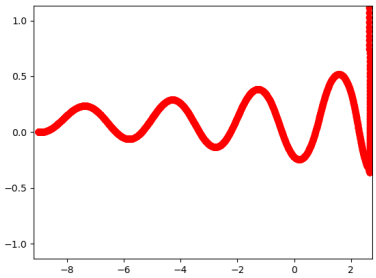
(g) ID: `P=1, I=0.1, D=10`



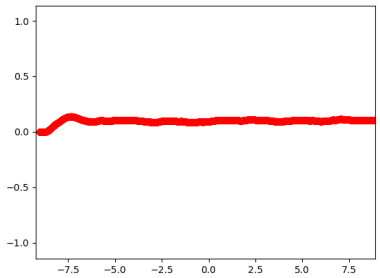
(h) I: `P=1, I=0.1, D=0`



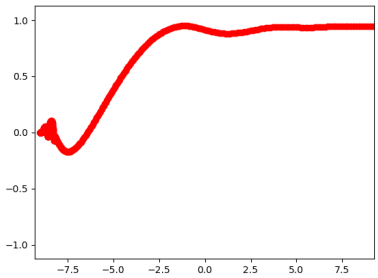
(i) D:`P=1, I=0, D=10`



(j) Neg P: `P=0.8, I=0, D=0`



(k) PID: `P=0.8, I=0.1, D=10`



(l) Inf D: `P=1, I=0, D=100`

Figure 5: Plotting robot positional changes with different sets of PID parameters.