Sim.I.am: A Robot Simulator

Coursera: Control of Mobile Robots

Jean-Pierre de la Croix

Last Updated: February 2, 2014

Contents

1 Introduction

This manual is going to be your resource for using the simulator with the programming assignments featured in the Coursera course, *Control of Mobile Robots* (and included at the end of this manual). It will be updated from time to time whenever new features are added to the simulator or any corrections to the course material are made.

1.1 Installation

Download `simiam-coursera-week-X.zip` (where X is the corresponding week number for the assignment) from the course page on Coursera under *Programming Assignments*. Make sure to download a new copy of the simulator **before** you start a new week's programming assignment, or whenever an announcement is made that a new version is available. It is important to stay up-to-date, since new versions may contain important bug fixes or features required for the programming assignment.

Unzip the `.zip` file to any directory.

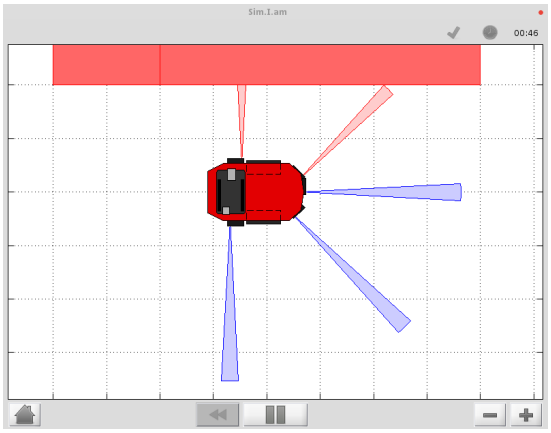
1.2 Requirements

You will need a reasonably modern computer to run the robot simulator. While the simulator will run on hardware older than a Pentium 4, it will probably be a very slow experience. You will also need a copy of MATLAB.

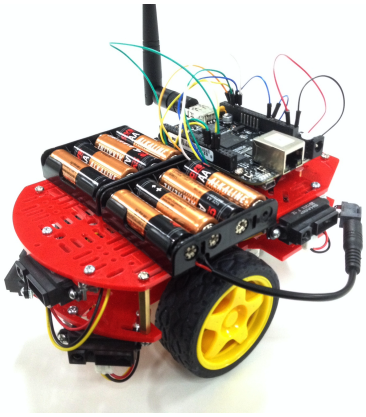
Thanks to support from MathWorks, a license for MATLAB and all required toolboxes is available to all students for the duration of the course. Check the *Getting Started with MATLAB* section under *Programming Assignments* on the course page for detailed instructions on how to download and install MATLAB on your computer.

1.3 Bug Reporting

If you run into a bug (issue) with the simulator, please create a post on the discussion forums in the *Programming Assignments* section. Make sure to leave a detailed description of the bug. Any questions or issues with MATLAB itself should be posted on the discussion forums in the *MATLAB* section.



(a) Simulated QuickBot



(b) Actual QuickBot

Figure 1: The QuickBot mobile robot in and outside of the simulator.

2 Mobile Robot

The mobile robot you will be working with in the programming exercises is the QuickBot. The QuickBot is equipped with five infrared (IR) range sensors, of which three are located in the front and two are located on its sides. The QuickBot has a two-wheel differential drive system (two wheels, two motors) with a wheel encoder for each wheel. It is powered by two 4x AA battery packs on top and can be controlled via software on its embedded Linux computer, the BeagleBone Black. You can build the QuickBot yourself by following the hardware lectures in this course.

Figure 2 shows the simulated and actual QuickBot mobile robot. The robot simulator recreates the QuickBot as faithfully as possible. For example, the range, output, and field of view of the simulated IR range sensors match the specifications in the datasheet for the actual Sharp GP2D120XJ00F infrared proximity sensors on the QuickBot.

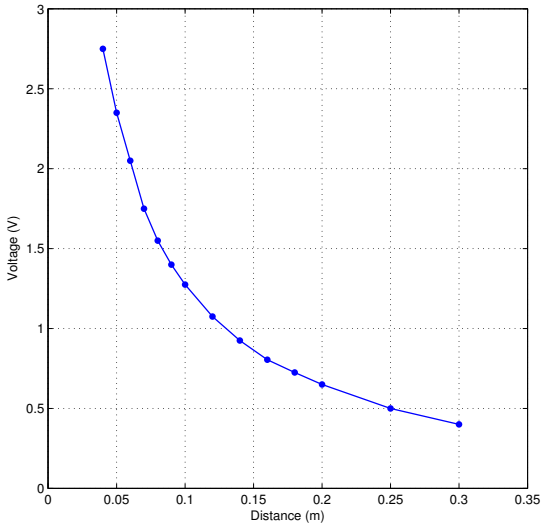
2.1 IR Range Sensors

You will have access to the array of five IR sensors that encompass the QuickBot. The orientation (relative to the body of the QuickBot, as shown in figure 1a) of IR sensors 1 through 5 is $90°, 45°, 0°, -45°, -90°$, respectively.I R range sensors are effective in the range 0.04 m to 0.3 m only. However, the IR sensors return raw values in the range of $[0.4, 2.75]V$ instead of the measured distances. Figure 2a demonstrates the function that maps these sensors values to distances. To complicate matters slightly, the BeagleBone Black digitizes the analog output voltage using a voltage divider and a 12-bit, 1.8V analog-to-digital converter (ADC). Figure 2b is a look-up table to demonstrate the relationship between the ADC output, the analog voltage from the IR proximity sensor, and the approximate distance that corresponds to this voltage.

Any controller can access the IR array through the `robot` object that is passed into its `execute` function. For example,

```
ir_distances = robot.get_ir_distances();
for i=1:numel(robot.ir_array)
  fprintf('IR #%d has a value of %d', i, robot.ir_array(i).get_range());
  fprintf('or %0.3f meters.\n', ir_distances(i));
end
```

It is assumed that the function `get_ir_distances` properly converts from the ADC output to an analog output voltage, and then from the analog output voltage to a distance in meters. The conversion from ADC output to analog output voltage is simply,



(a) Analog voltage output when an object is between 0.04m and 0.3m in the IR proximity sensor's field of view.

Distance (m)	Voltage (V)	ADC Out
0.04	2.750	917
0.05	2.350	783
0.06	2.050	683
0.07	1.750	583
0.08	1.550	517
0.09	1.400	467
0.10	1.275	425
0.12	1.075	358
0.14	0.925	308
0.16	0.805	268
0.18	0.725	242
0.20	0.650	217
0.25	0.500	167
0.30	0.400	133

(b) A look-up table for interpolating a distance (m) from the analog (and digital) output voltages.

Figure 2: A graph and a table illustrating the relationship between the distance of an object within the field of view of an infrared proximity sensor and the analog (and digital) ouptut voltage of the sensor.

$$V_{\text{ADC}} = \left\lfloor \frac{1000 \cdot V_{\text{analog}}}{3} \right\rceil$$

Converting from the the analog output voltage to a distance is a little bit more complicated, because a) the relationships between analog output voltage and distance is not linear, and b) the look-up table provides a coarse sample of points on the curve in Figure 2a. MATLAB has a `polyfit` function to fit a curve to the values in the look-up table, and a `polyval` function to interpolate a point on that fitted curve. The combination of the these two functions can be use to approximate a distance based on the analog output voltage. For more information, see Section 4.2.

It is important to note that the IR proximity sensor on the actual QuickBot will be influenced by ambient lighting and other sources of interference. For example, under different ambient lighting conditions, the same analog output voltage may correspond to different distances of an object from the IR proximity sensor. This effect of ambient lighting (and other sources of noise) is **not** modelled in the simulator, but will be apparent on the actual hardware.

2.2 Differential Wheel Drive

Since the QuickBot has a differential wheel drive (i.e., is not a unicyle), it has to be controlled by specifying the angular velocities of the right and left wheel (v_r, v_l), instead of the linear and angular velocities of a unicycle (v, ω). These velocities are computed by a transformation from (v, ω) to (v_r, v_ℓ). Recall that the dynamics of the unicycle are defined as,

$$\begin{aligned} \dot{x} &= v cos(\theta) \\ \dot{y} &= v sin(\theta) \\ \dot{\theta} &= \omega. \end{aligned} \tag{1}$$

The dynamics of the differential drive are defined as,

$$\begin{aligned}\dot{x} &= \frac{R}{2}(v_r + v_\ell)cos(\theta) \\ \dot{y} &= \frac{R}{2}(v_r + v_\ell)sin(\theta) \\ \dot{\theta} &= \frac{R}{L}(v_r - v_\ell),\end{aligned} \tag{2}$$

where R is the radius of the wheels and L is the distance between the wheels.

The speed of the QuickBot can be set in the following way assuming that the `uni_to_diff` function has been implemented, which transforms (v, ω) to (v_r, v_ℓ):

```
v = 0.15; % m/s
w = pi/4; % rad/s
% Transform from v,w to v_r,v_l and set the speed of the robot
[vel_r, vel_l] = obj.robot.dynamics.uni_to_diff(robot,v,w);
obj.robot.set_speeds(vel_r, vel_l);
```

The maximum angular wheel velocity for the QuickBot is approximately 80 RPM or 8.37 rad/s. It is important to note that if the QuickBot is controlled ot move at maximum linear velocity, it is not possible to achieve any angular velocity, because the angular velocity of the wheel will have been maximized. Therefore, there exists a tradeoff between the linear and angular velocity of the QuickBot: *the faster the robot should turn, the slower it has to move forward.*

2.3 Wheel Encoders

Each of the wheels is outfitted with a wheel encoder that increments or decrements a tick counter depending on whether the wheel is moving forward or backwards, respectively. Wheel encoders may be used to infer the relative pose of the robot. This inference is called **odometry**. The relevant information needed for odometry is the radius of the wheel (32.5mm), the distance between the wheels (99.25mm), and the number of ticks per revolution of the wheel (16 ticks/rev). For example,

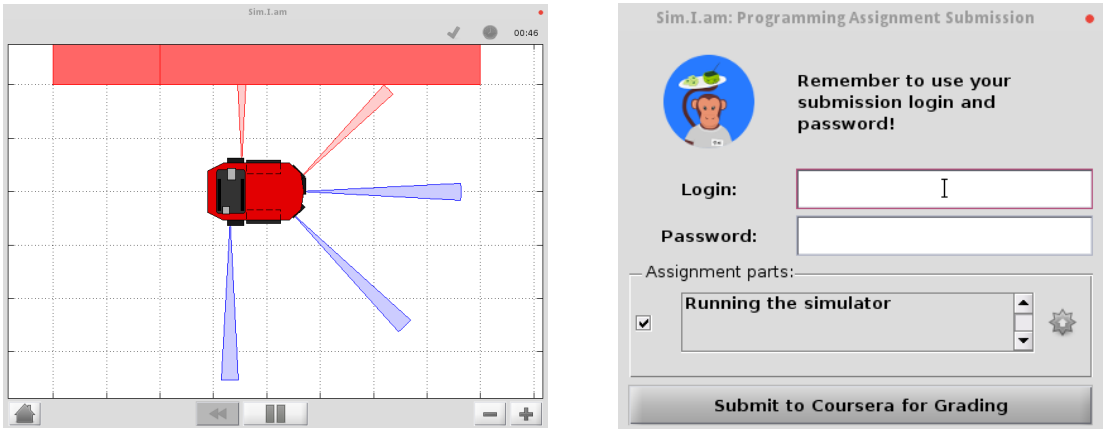
```
R = robot.wheel_radius; % radius of the wheel
L = robot.wheel_base_length; % distance between the wheels
tpr = robot.encoders(1).ticks_per_rev; % ticks per revolution for the right wheel

fprintf('The right wheel has a tick count of %d\n', robot.encoders(1).state);
fprintf('The left wheel has a tick count of %d\n', robot.encoders(2).state);
```

For more information about odometry, see Section 4.2.

3 Simulator

Start the simulator with the `launch` command in MATLAB from the command window. It is important that this command is executed inside the unzipped folder (but not inside any of its subdirectories).



(a) Simulator

(b) Submission screen

Figure 3: `launch` starts the simulator, while `submit` brings up the submission tool.

Figure 3a is a screenshot of the graphical user interface (GUI) of the simulator. The GUI can be controlled by the bottom row of buttons. The first button is the *Home* button and returns you to the home screen. The second button is the *Rewind* button and resets the simulation. The third button is the *Play* button, which can be used to play and pause the simulation. The set of *Zoom* buttons or the mouse scroll wheel allows you to zoom in and out to get a better view of the simulation. Clicking, holding, and moving the mouse allows you to pan around the environment. You can click on a robot to follow it as it moves through the environment.

Figure 3b is a screenshot of the submission screen. Each assignment can be submitted to Coursera for automatic grading and feedback. Start the submission tool by typing `submit` into the MATLAB command window. Use your login and password from the *Assignments* page. Your Coursera login and password will **not** work. Select which parts of the assignments in the list you would like to submit, then click *Submit to Coursera for Grading*. You will receive feedback, either a green checkmark for pass, or a red checkmark for fail. If you receive a red checkmark, check the MATLAB command window for a helpful message.

4 Programming Assignments

The following sections serve as a tutorial for getting through the simulator portions of the programming exercises. Places where you need to either edit or add code is marked off by a set of comments. For example,

```
%% START CODE BLOCK %%
  [edit or add code here]
%% END CODE BLOCK %%
```

To start the simulator with the `launch` command from the command window, it is important that this command is executed inside the unzipped folder (but not inside any of its subdirectories).

4.1 Week 1

This week's exercises will help you learn about MATLAB and robot simulator:

1. Since the assignments in this course involve programming in MATLAB, you should familiarize yourself with MATLAB (both the environment and the language). Review the resources posted in the "Getting Started with MATLAB" section on the *Programming Assignments* page.

2. Familiarize yourself with the simulator by reading this manual and downloading the robot simulator posted on the *Programming Assignments* section on the Coursera page.

4.2 Week 2

Start by downloading the robot simulator for this week from the Week 2 programming assignment. Before you can design and test controllers in the simulator, you will need to implement three components of the simulator:

1. Implement the transformation from unicycle dynamics to differential drive dynamics, i.e. convert from (v, ω) to the right and left **angular** wheel speeds (v_r, v_l).

   In the simulator, (v, ω) corresponds to the variables `v` and `w`, while (v_r, v_l) correspond to the variables `vel_r` and `vel_l`. The function used by the controllers to convert from unicycle dynamics to differential drive dynamics is located in `+simiam/+robot/+dynamics/DifferentialDrive.m`. The function is named `uni_to_diff`, and inside of this function you will need to define `vel_r` (v_r) and `vel_l` (v_l) in terms of `v`, `w`, `R`, and `L`. `R` is the radius of a wheel, and `L` is the distance separating the two wheels. Make sure to refer to Section 2.2 on "Differential Wheel Drive" for the dynamics.

2. Implement odometry for the robot, such that as the robot moves around, its pose (x, y, θ) is estimated based on how far each of the wheels have turned. Assume that the robot starts at (0,0,0).

   The tutorial located at `www.orcboard.org/wiki/images/1/1c/OdometryTutorial.pdf` covers how odometry is computed. The general idea behind odometry is to use wheel encoders to measure the distance the wheels have turned over a small period of time, and use this information to approximate the change in pose of the robot.

   The pose of the robot is composed of its position (x, y) and its orientation θ on a 2 dimensional plane (**note**: the video lecture may refer to robot's orientation as ϕ). The currently estimated pose is stored in the variable `state_estimate`, which bundles `x` (x), `y` (y), and `theta` (θ). The robot updates the estimate of its pose by calling the `update_odometry` function, which is located in `+simiam/+controller/+quickbot/QBSupervisor.m`. This function is called every `dt` seconds, where `dt` is 0.033s (or a little more if the simulation is running slower).

```
% Get wheel encoder ticks from the robot
right_ticks = obj.robot.encoders(1).ticks;
left_ticks = obj.robot.encoders(2).ticks;

% Recall the wheel encoder ticks from the last estimate
prev_right_ticks = obj.prev_ticks.right;
prev_left_ticks = obj.prev_ticks.left;

% Previous estimate
[x, y, theta] = obj.state_estimate.unpack();

% Compute odometry here
R = obj.robot.wheel_radius;
L = obj.robot.wheel_base_length;
m_per_tick = (2*pi*R)/obj.robot.encoders(1).ticks_per_rev;
```

The above code is already provided so that you have all of the information needed to estimate the change in pose of the robot. `right_ticks` and `left_ticks` are the accumulated wheel encoder ticks of the right and left wheel. `prev_right_ticks` and `prev_left_ticks` are the wheel encoder ticks of the right and left wheel saved during the last call to `update_odometry`. `R` is the radius of each wheel, and `L` is the distance separating the two wheels. `m_per_tick` is a constant that tells you how many meters a wheel covers with each tick of the wheel encoder. So, if you were to multiply `m_per_tick` by (`right_ticks-prev_right_ticks`), you would get the distance travelled by the right wheel since the last estimate.

Once you have computed the change in (x, y, θ) (let us denote the changes as `x_dt`, `y_dt`, and `theta_dt`) , you need to update the estimate of the pose:

```
theta_new = theta + theta_d;
x_new = x + x_dt;
y_new = y + y_dt;
```

3. Read the "IR Range Sensors" section in the manual and take note of the table in Figure 2b, which maps distances (in meters) to raw IR values. Implement code that converts raw IR values to distances (in meters).

   To retrieve the distances (in meters) measured by the IR proximity sensor, you will need to implement a conversion from the raw IR values to distances in the `get_ir_distances` function located in `+simiam/+robot/Quickbot.m`.

```
function ir_distances = get_ir_distances(obj)
    ir_array_values = obj.ir_array.get_range();
    ir_voltages = ir_array_values;
    coeff = [];
    ir_distances = polyval(coeff, ir_voltages);
end
```

   The variable `ir_array_values` is an array of the IR raw values. Divide this array by 500 to compute the `ir_voltages` array. The `coeff` should be the coefficients returned by

```
coeff = polyfit(ir_voltages_from_table, ir_distances_from_table, 5);
```

   where the first input argument is an array of IR voltages from the table in Figure 2b and the second argument is an array of the corresponding distances from the table in Figure 2b. The third argument specifies that we will use a fifth-order polynomial to fit to the data. Instead of running this fit every

time, execute the polyfit once in the MATLAB command line, and enter them manually on the third line, i.e. `coeff = [ ... ];`. If the coefficients are properly computed, then the last line will use polyval to convert from IR voltages to distances using a fifth-order polynomial using the coefficients in `coeff`.

How to test it all

To test your code, the simulator will is set to run a single P-regulator that will steer the robot to a particular angle (denoted θ_d or, in code, `theta_d`). This P-regulator is implemented in `+simiam/+controller/GoToAngle.m`. If you want to change the linear velocity of the robot, or the angle to which it steers, edit the following two lines in `+simiam/+controller/+quickbot/QBSupervisor.m`

```
obj.theta_d = pi/4;
obj.v = 0.1; %m/s
```

1. To test the transformation from unicycle to differential drive, first set `obj.theta_d=0`. The robot should drive straight forward. Now, set `obj.theta_d` to positive or negative $\frac{\pi}{4}$. If positive, the robot should start off by turning to its left, if negative it should start off by turning to its right. **Note**: If you haven't implemented odometry yet, the robot will just keep on turning in that direction.

2. To test the odometry, first make sure that the transformation from unicycle to differential drive works correctly. If so, set `obj.theta_d` to some value, for example $\frac{\pi}{4}$, and the robot's P-regulator should steer the robot to that angle. You may also want to uncomment the `fprintf` statement in the `update_odometry` function to print out the current estimate position to see if it make sense. Remember, the robot starts at $(x, y, \theta) = (0, 0, 0)$.

3. To test the IR raw to distances conversion, edit `+simiam/+controller/GoToAngle.m` and uncomment the following section:

   ```
   % for i=1:numel(ir_distances)
   %   fprintf('IR %d: %0.3fm\n', i, ir_distances(i));
   % end
   ```

   This `for` loop will print out the IR distances. If there are no obstacles (for example, walls) around the robot, these values should be close (if not equal to) 0.3m. Once the robot gets within range of a wall, these values should decrease for some of the IR sensors (depending on which ones can sense the obstacle). `Note:` The robot will eventually collide with the wall, because we have not designed an obstacle avoidance controller yet!

4.3 Week 3

Start by downloading the new robot simulator for this week from the Week 3 programming assignment. This week you will be implementing the different parts of a PID regulator that steers the robot successfully to some goal location. This is known as the go-to-goal behavior:

1. Calculate the heading (angle), θ_g, to the goal location (x_g, y_g). Let u be the vector from the robot located at (x, y) to the goal located at (x_g, y_g), then θ_g is the angle u makes with the x-axis (positive θ_g is in the counterclockwise direction).

   All parts of the PID regulator will be implemented in the file `+simiam/+controller/GoToGoal.m`. Take note that each of the three parts is commented to help you figure out where to code each part. The vector u can be expressed in terms of its x-component, u_x, and its y-component, u_y. u_x should be assigned to `u_x` and u_y to `u_y` in the code. Use these two components and the `atan2` function to compute the angle to the goal, θ_g (`theta_g` in the code).

2. Calculate the error between θ_g and the current heading of the robot, θ.

   The error `e_k` should represent the error between the heading to the goal `theta_g` and the current heading of the robot `theta`. Make sure to use `atan2` and/or other functions to keep the error between $[-\pi, \pi]$.

3. Calculate the proportional, integral, and derivative terms for the PID regulator that steers the robot to the goal.

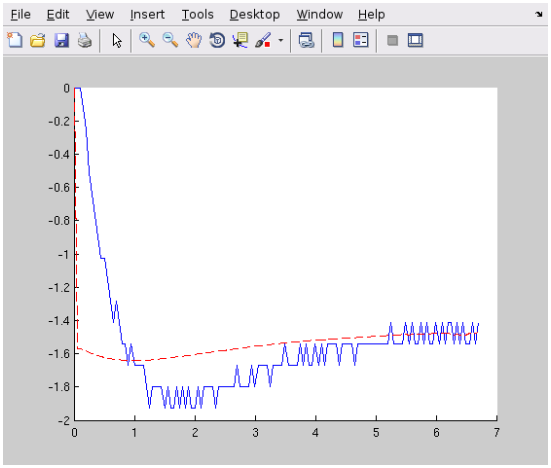
   As before, the robot will drive at a constant linear velocity `v`, but it is up to the PID regulator to steer the robot to the goal, i.e compute the correct angular velocity `w`. The PID regulator needs three parts implemented:

   (i) The first part is the proportional term `e_P`. It is simply the current error `e_k`. `e_P` is multiplied by the proportional gain `obj.Kp` when computing `w`.

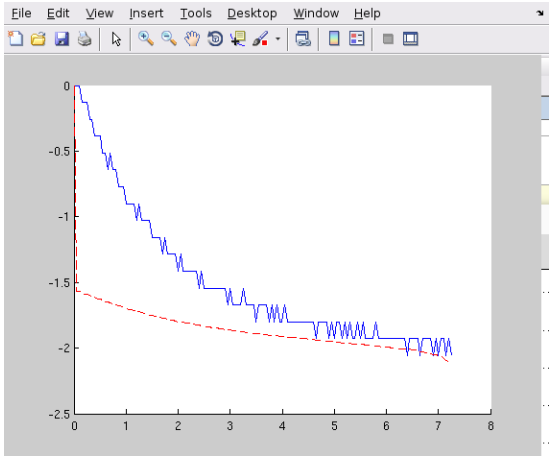
   (ii) The second part is the integral term `e_I`. The integral needs to be approximated in discrete time using the total accumulated error `obj.E_k`, the current error `e_k`, and the time step `dt`. `e_I` is multiplied by the integral gain `obj.Ki` when computing `w`, and is also saved as `obj.E_k` for the next time step.

   (iii) The third part is the derivative term `e_D`. The derivative needs to be approximated in discrete time using the current error `e_k`, the previous error `obj.e_k_1`, and the the time step `dt`. `e_D` is multiplied by the derivative gain `obj.Kd` when computing `w`, and the current error `e_k` is saved as the previous error `obj.e_k_1` for the next time step.

   Now, you need to tune your PID gains to get a fast settle time (θ matches θ_g within 10% in three seconds or less) and there should be little overshoot (maximum θ should not increase beyond 10% of the reference value θ_g). What you don't want to see are the following two graphs when the robot tries to reach goal location $(x_g, y_g) = (0, -1)$:



(a) Overshoot



(b) Undershoot (slow settle time)

Figure 4: PID gains were picked poorly, which lead to overshoot and poor settling times.

Figure 4b demonstrates undershoot, which could be fixed by increasing the proportional gain or adding some integral gain for better tracking. Picking better gains leads to the graph in Figure 5.

4. Ensure that the robot steers with an angular velocity ω, even if the combination of v and ω exceeds the maximum angular velocity of the robot's motors.

   This week we'll tackle the first of two limitations of the motors on the QuickBot. The first limitation is that the robot's motors have a maximum angular velocity, and the second limitation is that the

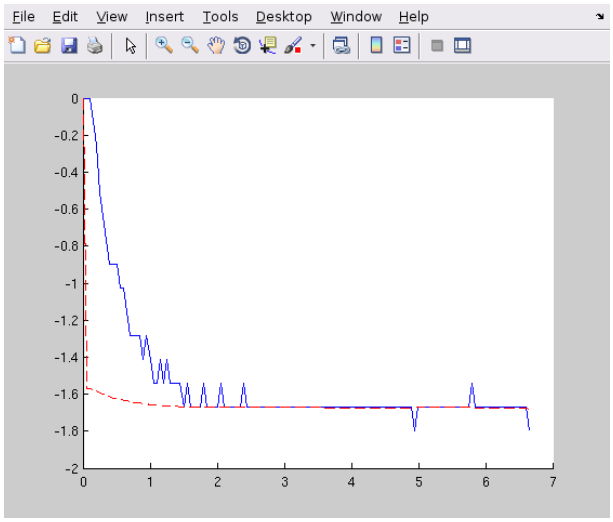


Figure 5: Faster settle time and good tracking with little overshoot.

motors stall at low speeds. We will discuss the latter limitation in a later week and focus our attention on the first limitation. Suppose that we pick a linear velocity v that requires the motors to spin at 90% power. Then, we want to change ω from 0 to some value that requires 20% more power from the right motor, and 20% less power from the left motor. This is not an issue for the left motor, but the right motor cannot turn at a capacity greater than 100%. The results is that the robot cannot turn with the ω specified by our controller.

Since our PID controllers focus more on steering than on controlling the linear velocity, we want to prioritize ω over v in situations, where we cannot satisfy ω with the motors. In fact, we will simply reduce v until we have sufficient headroom to achieve ω with the robot. The function `ensure_w` in `+simiam/+controller/+quickbot/QBSupervisor.m` is designed ensure that ω is achieved even if the original combination of v and ω exceeds the maximum v_r and v_l.

Complete `ensure_w`. Suppose $v_{r,d}$ and $v_{l,d}$ are the angular wheel velocities needed to achieve ω. Then `vel_rl_max` is $\max(v_{r,d}, v_{l,d})$ and `vel_rl_min` is $\min(v_{r,d}, v_{l,d})$. A motor's maximum forward angular velocity is `obj.robot.max_vel` (or $\text{vel}_{\max}$). So, for example, the equation that represents the `if/else` statement for the right motors is:

$$v_r = \begin{cases} v_{r,d} - (\max(v_{r,d}, v_{l,d}) - vel_{\max}) & \text{if } \max(v_{r,d}, v_{l,d}) > vel_{\max} \\ v_{r,d} - (\min(v_{r,d}, v_{l,d}) + vel_{\max}) & \text{if } \min(v_{r,d}, v_{l,d}) < -vel_{\max} \\ v_{r,d}, & \text{otherwise,} \end{cases}$$

which defines the appropriate v_r (or `vel_r`) needed to achieve ω. This equation also applies to computing a new v_l. The results of `ensures_w` is that if v and ω are so large that v_r and/or v_l exceed $\text{vel}_{\max}$, then v is scaled back to ensure ω is achieved (Note: ω is precapped at the beginnging of `ensure_w` to the maximum ω possible if the robot is stationary).

How to test it all

To test your code, the simulator is set up to use the PID regulator in `GoToGoal.m` to drive the robot to a goal location and stop. If you want to change the linear velocity of the robot, the goal location, or the distance from the goal the robot will stop, then edit the following three lines in `+simiam/+controller/+quickbot/QBSupervisor.m`.

```
obj.goal = [-1,1];
```

```
obj.v = 0.2;
obj.d_stop = 0.05;
```

Make sure the goal is located inside the walls, i.e. the x and y coordinates of the goal should be in the range $[-1, 1]$. Otherwise the robot will crash into a wall on its way to the goal!

1. To test the heading to the goal, set the goal location to `obj.goal = [1,1]`. `theta_g` should be approximately $\frac{\pi}{4} \approx 0.785$ initially, and as the robot moves forward (since $v = 0.1$ and $\omega = 0$) `theta_g` should increase. Check it using a `fprintf` statment or the plot that pops up. `theta_g` corresponds to the red dashed line (i.e., it is the reference signal for the PID regulator).

2. Test this part with the implementation of the third part.

3. To test the third part, run the simulator and check if the robot drives to the goal location and stops. In the plot, the blue solid line (`theta`) should match up with the red dashed line (`theta_g`). You may also use `fprintf` statements to verify that the robot stops within `obj.d_stop` meters of the goal location.

4. To test the fourth part, set `obj.v=10`. Then add the following two lines of code after the call to `ensure_w` in the `execute` function of `QBSupervisor.m`.

   ```
   [v_limited, w_limited] = obj.robot.dynamics.diff_to_uni(vel_r, vel_l);
   fprintf('(v,w) = (%0.3f,%0.3f), (v_limited,w_limited) = (%0.3f, %0.3f)\n', ...
                                       outputs.v, outputs.w, v_limited, w_limited);
   ```

   If $\omega \neq \omega_{\text{limited}}$, then ω is not ensured by `ensure_w`. This function should scale back v, such that it is possible for the robot to turn with the ω output by the controller (unless $|\omega| > 5.48$ rad/s).

How to migrate your solutions from last week.

Here are a few pointers to help you migrate your own solutions from last week to this week's simulator code. You only need to pay attention to this section if you want to use your own solutions, otherwise you can use what is provided for this week and skip this section.

1. You may overwrite `+simiam/+robot/+dynamics/DifferentialDrive.m` with your own version from last week.

2. You should not overwrite `+simiam/+robot/QuickBot.m` with your own version from last week! Many changes were made to this file for this week.

3. You should not overwrite `+simiam/+controller/+quickbot/QBSupervisor.m`! However, to use your own solution to the odometry, you can replace the provided `update_odometry` function in `K3Supervisor.m` with your own version from last week.