Sim.I.am: A Robot Simulator

Coursera: Control of Mobile Robots

Jean-Pierre de la Croix and Matthew Hale

Last Updated: March 9, 2016

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 *Optional Programming Assignment* page under *Week X*. 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.

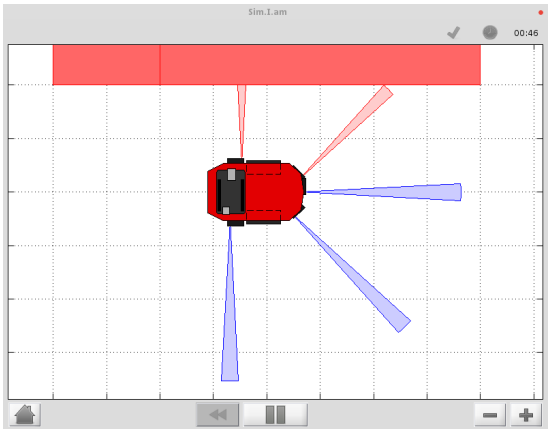
1.2 Requirements

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.

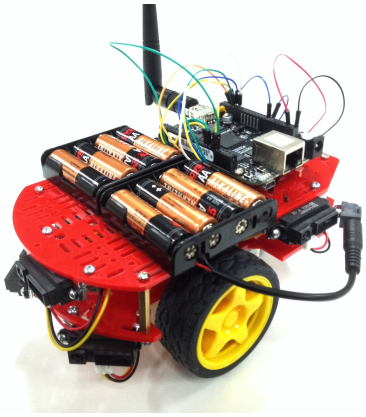
Thanks to support from MathWorks, a license for MATLAB and all required toolboxes are available to all students beginning in Week 2 and lasting until the end of the course. **You can easily put off the optional Week 1 MATLAB assignment until Week 2 without falling behind.** Check the *Installing MATLAB* page accessible from the *Week 2* page for a link that will let you install MATLAB on your computer.

1.3 MATLAB Help

Any questions about using MATLAB for the programming assignments should be posted to the course discussion forums.



(a) Simulated QuickBot



(b) Actual QuickBot

Figure 1: The simulated QuickBot and the mobile robot it is based upon.

2 Mobile Robot Simulator

The programming exercises use a simulated version of a robot called the QuickBot. The simulated 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 simulated QuickBot has a two-wheel differential drive system (two wheels, two motors) with a wheel encoder for each wheel. You can build the QuickBot yourself by following the hardware videos located under the *Course Resources* tab. **These hardware videos are not part of the course and their content is therefore unsupported by course staff. We provide them as a courtesy to learners in the course who may be interested in them.**

Figure 1 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.

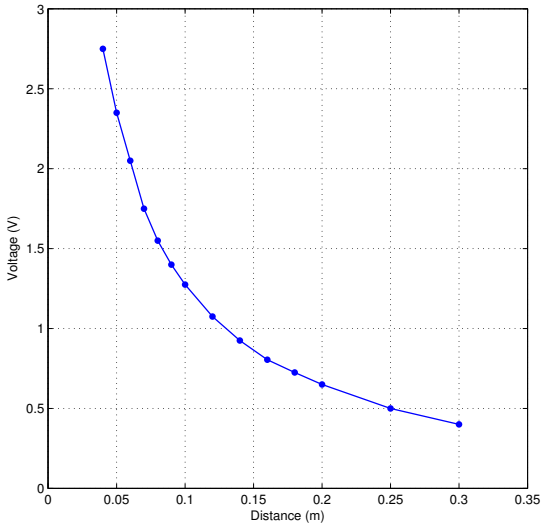
2.1 IR Range Sensors

In this section we cover some of the details pertaining to the five simulated IR sensors onboard the simulated QuickBot. The orientations (relative to the body of the QuickBot, as shown in Figure 1a) of IR sensors 1 through 5 are $90°, 45°, 0°$, $-45°$, and $-90°$, respectively. IR 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 onboard the physical QuickBot digitizes the analog output voltage using a voltage divider and a 12-bit, 1.8V analog-to-digital converter (ADC). To faithfully recreate the QuickBot in simulation, we simulate the effect of this digitization. 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



(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.

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

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

NOTE: For Week 2, the simulator uses a different voltage divider on the ADC; therefore, $V_{\textbf{ADC}} = V_{\textbf{analog}} \cdot 1000/2$. This has been fixed in subsequent weeks!

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, and this will be done as part of the programming assignment in Week 2.

2.2 Differential Wheel Drive

Since the simulated 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 simulated 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 physical QuickBot is approximately 80 RPM or 8.37 rad/s and this value is reflected in the simulator. It is therefore important to note that if the simulated QuickBot is controlled to 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 simulated 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 the Week 2 programming assignment.

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).

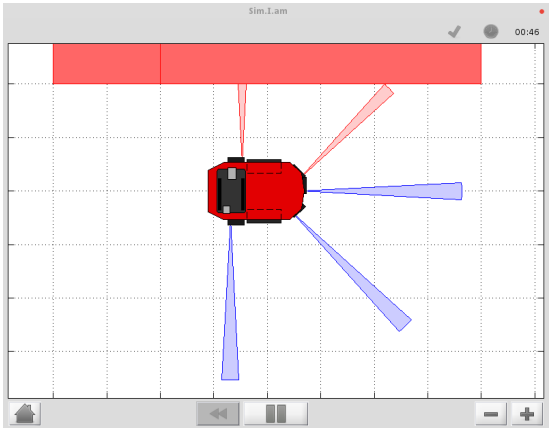


Figure 3: `launch` starts the simulator.

Figure 3 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.

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 are 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. **If you do not already have access to MATLAB, a student license will be made available to registered participants of this course starting in Week 2. You can easily put off the optional Week 1 MATLAB assignment until Week 2 without falling behind.**

1. Install the GUI Layout Toolbox using the instructions provided in the *Preparing for Programming Assignments* page under *Week 1*.

2. Familiarize yourself with the simulator by reading this manual and downloading the robot simulator posted on the *Optional Programming Assignment 1: Instructions* section under *Week 1*.

3. Start the simulator by running the `launch` command from the command window. If the GUI Layout Toolbox has been installed correctly (instructions for this can be found in the *Preparing for Programming Assignments* page under *Week 1*), the home screen should appear. Pressing the Play button will then show a robot which is stationary (it will move once we design controllers for it next week!).

4.2 Week 2

If you are installing MATLAB this week, follow the instructions on the page titled *Installing MATLAB* under Week 2 first. Then return to Week 1's assignment (above) before completing this week.

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_dt;
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 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.

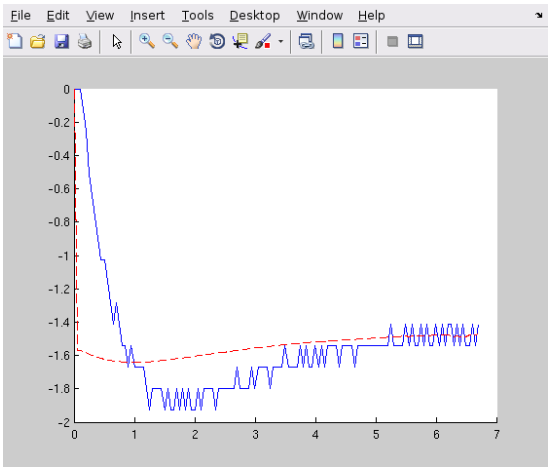
   (iii) The third part is the derivative term `e_D`. The derivative needs to be approximated in discrete time using the current error `e_k`, the previous error `obj.e_k_1`, and the the time step `dt`. `e_D` is multiplied by the derivative gain `obj.Kd` when computing `w`, and the current error `e_k` is 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 $(x_g, y_g) = (0, -1)$:

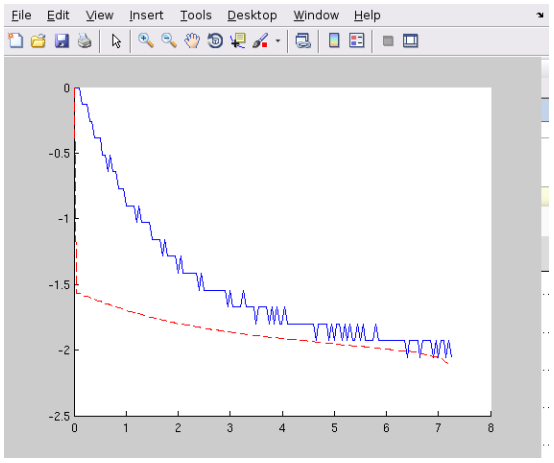
   Figure 4b demonstrates undershoot, which could be fixed by increasing the proportional gain or adding some integral gain for better tracking. Picking better gains leads to the graph in Figure 5.

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 physical QuickBot (which are simulated on the QuickBot we use in simulation). The first limitation is that the robot's motors have a maximum angular velocity, and the second limitation is that the 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



(a) Overshoot



(b) Undershoot (slow settle time)

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

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 `ensure_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!

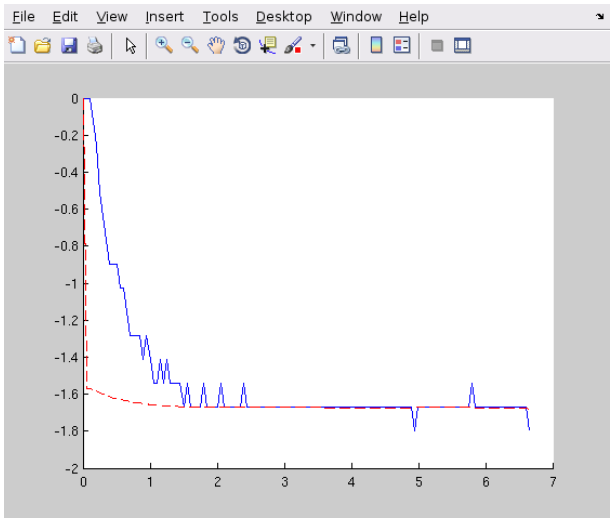


Figure 5: Faster settle time and good tracking with little overshoot.

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 `QBSupervisor.m` with your own version from last week.

4.4 Week 4

Start by downloading the new robot simulator for this week from the Week 4 programming assignment. This week you will be implementing the different parts of a controller that steers the robot successfully away from obstacles to avoid a collision. This is known as the avoid-obstacles behavior. The IR sensors allow us to measure the distance to obstacles in the environment, but we need to compute the points in the world to which these distances correspond. Figure 6 illustrates these points with a black cross. The

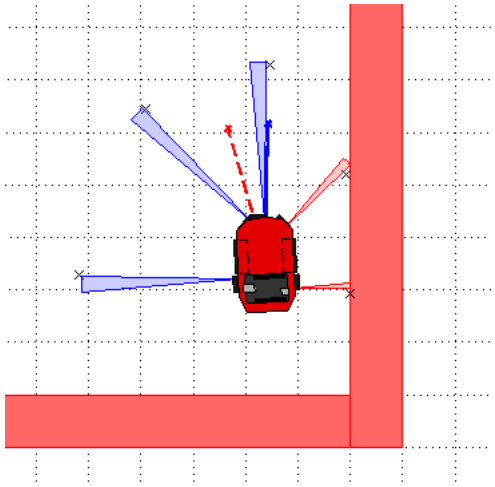


Figure 6: IR range to point transformation.

strategy for obstacle avoidance that we will use is as follows:

1. Transform the IR distances to points in the world.
2. Compute a vector to each point from the robot, $u_1, u_2, \ldots, u_9$.
3. Weigh each vector according to their importance, $\alpha_1 u_1, \alpha_2 u_2, \ldots, \alpha_9 u_9$. For example, the front and side sensors are typically more important for obstacle avoidance while moving forward.
4. Sum the weighted vectors to form a single vector, $u_{ao} = \alpha_1 u_1 + \ldots + \alpha_9 u_9$.
5. Use this vector to compute a heading and steer the robot to this angle.

This strategy will steer the robot in a direction with the most free space (i.e., it is a direction *away* from obstacles). For this strategy to work, you will need to implement three crucial parts of the strategy for the obstacle avoidance behavior:

1. Transform the IR distance (which you converted from the raw IR values in Week 2) measured by each sensor to a point in the reference frame of the robot.

   A point p_i that is measured to be d_i meters away by sensor i can be written as the vector (coordinate) $v_i = \begin{bmatrix} d_i \\ 0 \end{bmatrix}$ in the reference frame of sensor i. We first need to transform this point to be in the reference frame of the robot. To do this transformation, we need to use the pose (location and orientation) of the sensor in the reference frame of the robot: $(x_{s_i}, y_{s_i}, \theta_{s_i})$ or in code, `(x_s,y_s,theta_s)`. The transformation is defined as:

$$v_i' = R(x_{s_i}, y_{s_i}, \theta_{s_i}) \begin{bmatrix} v_i \\ 1 \end{bmatrix},$$

where R is known as the transformation matrix that applies a translation by (x, y) and a rotation by θ:

$$R(x, y, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix},$$

which you need to implement in the function `obj.get_transformation_matrix`.

In `+simiam/+controller/+AvoidObstacles.m`, implement the transformation in the `apply_sensor_geometry` function. The objective is to store the transformed points in `ir_distances_rf`, such that this matrix has v'_1 as its first column, v'_2 as its second column, and so on.

2. Transform the point in the robot's reference frame to the world's reference frame.

   A second transformation is needed to determine where a point p_i is located in the world that is measured by sensor i. We need to use the pose of the robot, (x, y, θ), to transform the robot from the robot's reference frame to the world's reference frame. This transformation is defined as:

   $$v''_i = R(x, y, \theta)v'_i$$

   In `+simiam/+controller/+AvoidObstacles.m`, implement this transformation in the `apply_sensor_geometry` function. The objective is to store the transformed points in `ir_distances_wf`, such that this matrix has v''_1 as its first column, v''_2 as its second column, and so on. This matrix now contains the coordinates of the points illustrated in Figure 6 by the black crosses. Note how these points *approximately* correspond to the distances measured by each sensor (Note: *approximately*, because of how we converted from raw IR values to meters in Week 2).

3. Use the set of transformed points to compute a vector that points away from the obstacle. The robot will steer in the direction of this vector and successfully avoid the obstacle.

   In the function `execute` implement parts 2.-4. of the obstacle avoidance strategy.

   (i) Compute a vector u_i to each point (corresponding to a particular sensor) from the robot. Use a point's coordinate from `ir_distances_wf` and the robot's location (`x`,`y`) for this computation.

   (ii) Pick a weight α_i for each vector according to how important you think a particular sensor is for obstacle avoidance. For example, if you were to multiply the vector from the robot to point i (corresponding to sensor i) by a small value (e.g., 0.1), then sensor i will not impact obstacle avoidance significantly. Set the weights in `sensor_gains`. **Note**: Make sure to that the weights are symmetric with respect to the left and right sides of the robot. Without any obstacles around, the robot should only steer slightly right (due to a small asymmetry in the how the IR sensors are mounted on the robot).

   (iii) Sum up the weighted vectors, $\alpha_i u_i$, into a single vector u_{ao}.

   (iv) Use u_{ao} and the pose of the robot to compute a heading that steers the robot away from obstacles (i.e., in a direction with free space, because the vectors that correspond to directions with large IR distances will contribute the most to u_{ao}).

QuickBot Motor Limitations

Last week we implemented a function, `ensure_w`, which was responsible for respecting ω from the controller as best as possible by scaling v if necessary. This implementation assumed that it was possible to control the angular velocity in the range $[-\text{vel}_{\max}, \text{vel}_{\max}]$. This range was implemented in the simulation to reflect the fact that the motors on the physical QuickBot have a maximum rotational speed. However, it is also true that the motors have a minimum speed before the robot starts moving. If not enough power is applied to the motors, the angular velocity of a wheel remains at 0. Once enough power is applied, the wheels spin at a speed $\text{vel}_{\min}$.

The `ensure_w` function has been updated this week to take this limitation into account. For example, small (v, ω) may not be achievable on the QuickBot, so `ensure_w` scales up v to make ω possible. Similarily, if (v, ω) are both large, `ensure_w` scales down v to ensure ω (as was the case last week). You can uncomment the two `fprintf` statements to see (v, ω) before and after.

There is nothing that needs to be added or implemented for this week in `ensure_w`, but you may find it interesting how one deals with physical limitations on a mobile robot, like the QuickBot. This particular approach has an interesting consequence, which is that if $v > 0$, then v_r and v_l are both positive (and vice versa, if $v < 0$). Therefore, we often have to increase or decrease v significantly to ensure ω even if it were better to make small adjustments to both ω and v. As with most of the components in these programming assignments, there are alternative designs with their own advantages and disadvantages. Feel free to share your designs with everyone on the discussion forums!

How to test it all

To test your code, the simulator is set up to use load the `AvoidObstacles.m` controller to drive the robot around the environment without colliding with any of the walls. If you want to change the linear velocity of the robot, then edit the following line in `+simiam/+controller/+quickbot/QBSupervisor.m`.

```
obj.v = 0.2;
```

Here are some tips on how to test the three parts:

1. Test the first part with the second part.
2. Once you have implemented the second part, one black cross should match up with each sensor as shown in Figure 6. The robot should drive forward and collide with the wall. The blue line indicates the direction that the robot is currently heading (θ).
3. Once you have implemented the third part, the robot should be able to successfully navigate the world without colliding with the walls (obstacles). If no obstacles are in range of the sensors, the red line (representing u_{ao}) should just point forward (i.e., in the direction the robot is driving). In the presence of obstacles, the red line should point away from the obstacles in the direction of free space.

You can also tune the parameters of the PID regulator for ω by editing `obj.Kp`, `obj.Ki`, and `obj.Kd` in `AvoidObstacles.m`. The PID regulator should steer the robot in the direction of u_{ao}, so you should see that the blue line tracks the red line. **Note:** The red and blue lines (as well as, the black crosses) will likely deviate from their positions on the robot. The reason is that they are drawn with information derived from the odometry of the robot. The odometry of the robot accumulates error over time as the robot drives around the world. This odometric drift can be seen when information based on odometry is visualized via the lines and crosses.

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 the same files as listed for Week 3.
2. You may overwrite `+simiam/+controller/GoToGoal.m` with your own version from last 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 `QBSupervisor.m` with your own version from last week.
4. You may replace the PID regulator in `+simiam/+controller/AvoidObstacles.m` with your own version from the previous week (i.e., use the PID code from `GoToGoal.m`).