Sim.I.am: A Robot Simulator

ECE4555 Embedded and Hybrid Control

Jean-Pierre de la Croix

Last Updated: March 7, 2013

Manual

This manual is going to be your resource for using the simulator in the homeworks and projects for this course. It will be updated throughout the semester, so make sure to check it on a regular basis. You can access it anytime from T-Square under `ECE4555/Resources`.

Installation

Download `simiam.zip` from T-Square under `ECE4555/Resources`. Make sure to download a new copy of the simulator **before** you start a new homework, 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 homeworks and projects.

Unzip the `.zip` file to any directory.

Requirements

You will need a reasonably modern computer to run the K3 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. The simulator has been tested with MATLAB R2011a, so it is recommended that you use that version or higher.

Bug Reporting

If you run into a bug (issue) with the simulator, please contact `jdelacroix@gatech.edu` with a detailed description. The bug will get fixed and an announcement will be made that a new version of the simulator is available on T-Square.

Mobile Robot

The mobile robot platform you will be using in the homeworks and projects is the Khepera III (K3) mobile robot. The K3 is equipped with 11 infrared (IR) range sensors, of which nine are located in a ring around it and two are located on the underside of the robot. The IR sensors are complemented by a set of five ultrasonic sensors. The K3 has a two-wheel differential drive with a wheel encoder for each wheel. It is powered by a single battery on the underside and can be controlled via software on its embedded Linux computer.

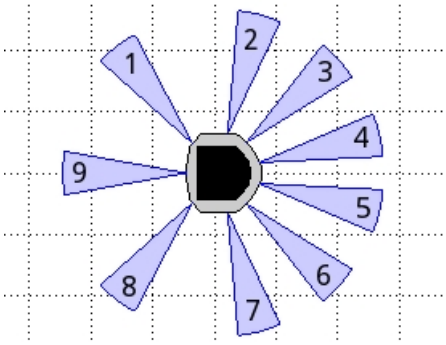

Figure 1: IR range sensor configuration

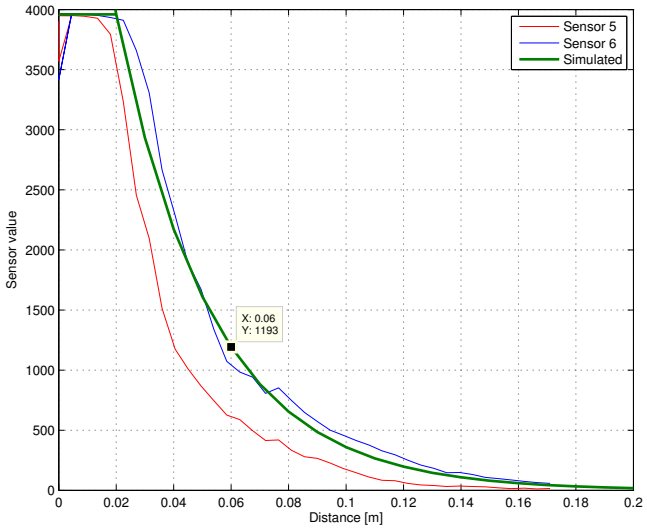

Figure 2: Sensor values vs. Measured Distance

IR Range Sensors

For the purpose of the homeworks and projects in the class, you will have access to the array of nine IR sensors that encompass the K3. IR range sensors are effective in the range 0.02 m to 0.2 m only. However, the IR sensors return raw values in the range of $[18, 3960]$ instead of the measured distances. Figure 2 demonstrates the function that maps these sensors values to distances.

The green plot represents the sensor model used in the simulator, while the blue and red plots show the sensor response of two different IR sensors (under different ambient lighting levels). The effect of ambient lighting (and other sources of noise) are **not** modelled in the simulator, but will be apparent on the actual hardware.

The function that maps distances, denoted by Δ, to sensor values is the following piecewise function:

$$f(\Delta) = \begin{cases} 3960, & \text{if } 0\text{m} \leq \Delta \leq 0.02\text{m} \\ \lfloor 3960e^{-30(\Delta - 0.02)} \rfloor, & \text{if } 0.02\text{m} \leq \Delta \leq 0.2\text{m} \end{cases} \tag{1}$$

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

```
for i=1:9
   fprintf('IR #%d has a value of %d.\n', i, robot.ir_array(i).get_range());
end
```

The orientation (relative to the body of the K3, as shown in figure 1) of IR sensors 1 through 9 is $128°, 75°, 42°$, $13°, -13°, -42°, -75°, -128°$, and $180°$, respectively.

Ultrasonic Range Sensors

The ultrasonice range sensors have a sensing range of 0.2m to 4m, but are not available in the simulator.

Differential Wheel Drive

Since the K3 has a differential wheel drive (i.e., is not a unicyle), it has to be controlled by specifying the rotational velocities of the right and left wheel. 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,

$$\begin{aligned} \dot{x} &= vcos(\theta) \\ \dot{y} &= vsin(\theta) \\ \dot{\theta} &= \omega. \end{aligned} \tag{2}$$

The dynamics of the differential drive are defined as,

$$\begin{aligned} \dot{x} &= \frac{R}{2}(v_r + v_\ell)cos(\theta) \\ \dot{y} &= \frac{R}{2}(v_r + v_\ell)sin(\theta) \\ \dot{\theta} &= \frac{R}{L}(v_r - v_\ell), \end{aligned} \tag{3}$$

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

The speed of the K3 can be set in the following way assuming that you have implemented the `uni_to_diff` function, 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);
```

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, the distance between the wheels, and the number of ticks per revolution of the wheel.

```
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);
```

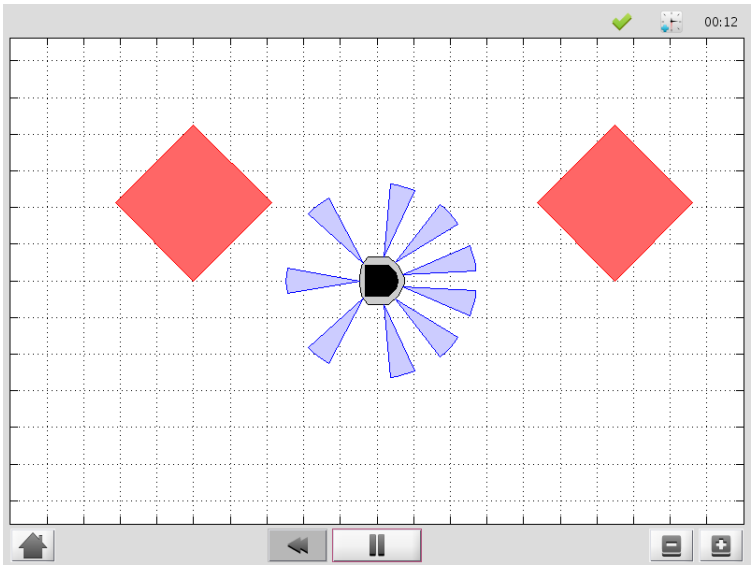

Figure 3: Simulator

Simulator

Start the simulator with the `launch` command in MATLAB.

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 (or their equivalent keyboard shortcuts). The first button is the *Home* button `[h]` 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 `[p]`, 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. The set of *Pan* buttons `[left,right,up,down]` can be used to pan around the environment, or alternatively, Clicking, holding, and moving the mouse allows you to pan too. *Track* button `[c]` can be used to toggle between a fixed camera view and a view that tracks the movement of the robot. You can also simply click on a robot to follow it.

Homeworks

The following sections serve as a tutorial for getting through the simulator portions of the homeworks.

Homework 0

The first homework asks you to implement odometry in the `+simiam/+controller/+khepera3/K3Supervisor.m` file, as well as, the transformation from (v, ω) to (v_r, v_ℓ) in the `+simiam/+robot/+dynamics/DifferentialDrive.m` file. Since odometry and this particular transformation will be used in all of your controllers, make sure to implement these correctly.

To test odometry and your transformation, you will likely want to edit `+simiam/+controller/GoToGoal.m` and implement a P-regulator. Similarily, for converting raw IR values to distances, edit `+simiam/+controller/AvoidObstacle.m`.

Project 1

The first project requires you to steer the robot to some desired angle (`theta_d`). You will need to implement a PID regulator that achieves this quickly and smoothly. Edit `+simiam/+controller/GoToAngle.m`

and implement the PID controller there. Remember, `execute` is called once every timestep `dt`, so will probably want to create variable under `properties` to save a previous error or soemthing else you want to use during the next iteration. In `+simiam/+controller/+khepera3/K3Supervisor.m`, adjust the desired angle `inputs.theta_d = pi/4;` to make sure it works more than just the default. Also, remember to implement homework 0 before you start with this project!

Project 2

The second project requires you to drive the robot to a series of given locations `x_g` and `y_g`. These locations are specified in `K3Supervisor.m` as a matrix, where each row corresponds to a goal and the columns correspond to the x and y location, see: `obj.goal_list`. `obj.goal_index` should keep track of which goal is being driven to. Use both of these variable, plus the current location of the robot to implement some simple logic that will cause the robot to drive to all three goals and then stop. `K3Supervisor.m` already has some **pseudocode** to give you an idea how this could be achieved. Feel free to implement your own logic!

The actual go-to-goal behavior needs to be implement in `GoToGoal.m`. It is recommended that you implement a PID regulator that steers the robot to the goal with a constant linear velocity. If you'd like, you may experiment with regulating the linear velocity too. The objective is for the robot to drive to all goal points and stop as close as possible to the final goal point.

Project 3

The third project requires you to drive the robot away from obstacles in the environment. The robot should just drive straight when there are no obstacles, and once it detects an obstacle (in the simulator it will be a wall) it should avoid colliding with it. Compute (v, ω) that can achieve this obstacle avoiding behavior.

Hint: You may want to think about obstacle avoidance in terms of computing a vector that points away from the obstacle, and using this vector in a similar way as the go-to-goal vector to control the robot.

You may also want to think about filtering the IR measurements and/or adding a threshold for obstacle avoidance as you design this behavior.

The obstacle avoidance behavior should be implemented in `+simiam/+controller/AvoidObstacles.m`. This requires you to also properly implement the IR raw values to meters conversion in `get_ir_distances` in `+simiam/+robot/Khepera3.m` (refer back to homework 0 for more details).

Project 4

The fourth project requires you to drive the robot to a series of goal locations without colliding with any of the obstacles in the environment. You will test two strategies:

1. In `+simiam/+controller/AOandGTG.m`, implement a blend of the go-to-goal and avoid-obstacles behaviors by combining the two controllers in a meaningful way.

2. In `+simiam/+controller/+khepera3/K3Supervisor.m`, implement switching logic that **only** switches the robot between the go-to-goal controller defined in `GoToGoal.m` and the avoid-obstacles controller defined in `AvoidObstacles.m`. Use the `switch_controller` function to switch between two controllers.

Think about the advantage and disadvantages of both strategies and come up with a switching logic that takes advantage of all three controllers.

Project 5

The fifth project requires you to implement a wall following behavior that can be toggled (or dynamically decides) to follow walls to either the right or left. The robot needs to be able to follow the contour of any obstacle it follows indefinitely (i.e. neither collide or break away from the obstacle).

`+simiam/+controller/FollowWall.m` should contain your implementation. Edit `settings.xml` to add interesting obstacles into the environment for testing. I suggest that you place (initialize) the robot near an obstacle, since you can assume that a go-to-goal obstacle has already "navigated" the robot to an obstacle.

Project 6

The sixth project requires to combine all of the controllers that you have implemented in the previous projects into a full navigation system. The navigation system should transition between the different behaviors (go-to-goal, obstacle avoidance, ao-and-gtg, wall-following) in order to navigate a complex obstacle course. Up until now obstacles have been convex, but concave obstacles (for example, a U shaped obstacle) pose new problems. The supervisor needs to be able to reason about whether the robot is stuck inside a concave obstacle, and if so initiate a strategy that allows it to navigate out of such obstacles.

The skeleton code for `K3Supervisor.m` includes new code to create states and events, which can be used to construct a finite state machine (FSM) that represents the logic that is used by the navigation system. You may want to extend the set of states and events, such as, creating an event that detects whether the robot is making any forward progress towards the goal. If not, it may be stuck inside of a concave obstacle and the FSM can trigger on that event to switch to another controller (e.g. follow-wall).