Sim.I.am: A Robot Simulator

Coursera: Control of Mobile Robots

Jean-Pierre de la Croix

Last Updated: March 1, 2013

Contents

1 Introduction **2**
1.1 Installation . . . 2
1.2 Requirements . . . 2
1.3 Bug Reporting . . . 2

2 Mobile Robot **2**
2.1 IR Range Sensors . . . 3
2.2 Ultrasonic Range Sensors . . . 3
2.3 Differential Wheel Drive . . . 3
2.4 Wheel Encoders . . . 4

3 Simulator **4**

4 Programming Exercises **5**
4.1 Week 1 . . . 5
4.2 Week 2 . . . 5
4.3 Week 3 . . . 8
4.4 Week 4 . . . 9
4.5 Week 5 . . . 12
4.6 Week 6 . . . 16

1 Introduction

This manual is going to be your resource for using the simulator in the programming exercises for this course. It will be updated throughout the course, so make sure to check it on a regular basis. You can access it anytime from the course page on Coursera under `Programming Exercises`.

1.1 Installation

Download `simiam-coursera-week-X.zip` (where X is the corresponding week for the exercise) from the course page on Coursera under `Programming Exercises`. Make sure to download a new copy of the simulator **before** you start a new week's programming exercises, 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 exercises.

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. The simulator has been tested with MATLAB R2009a, so it is recommended that you use that version or higher. No additional toolboxes are needed to run the simulator.

1.3 Bug Reporting

If you run into a bug (issue) with the simulator, please leave a message in the discussion forums with a detailed description. The bug will get fixed and a new version of the simulator will be uploaded to the course page on Coursera.

2 Mobile Robot

The mobile robot platform you will be using in the programming exercises 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.

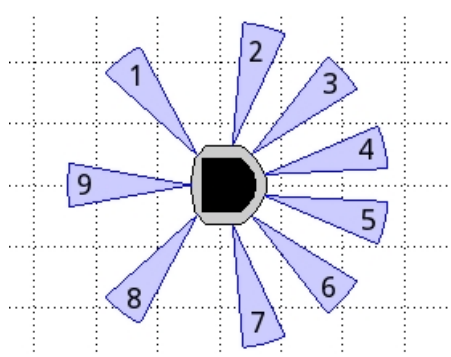


Figure 1: IR range sensor configuration

2.1 IR Range Sensors

For the purpose of the programming exercises in the course, 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.

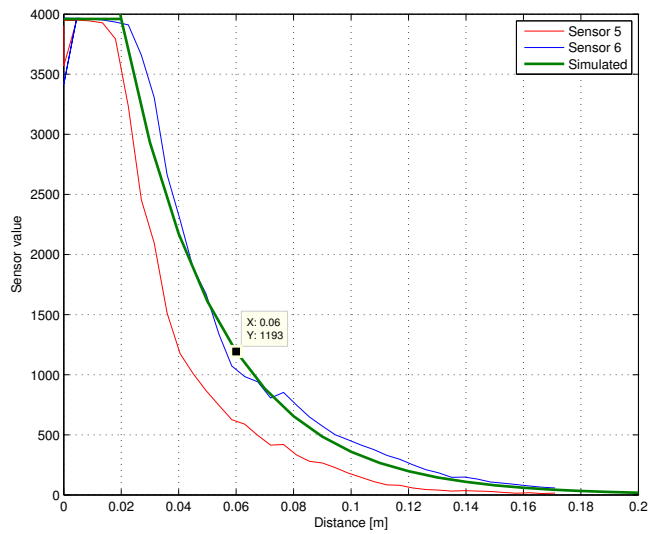


Figure 2: Sensor values vs. Measured Distance

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:

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

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.

2.2 Ultrasonic Range Sensors

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

2.3 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 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,

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

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);
```

2.4 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. 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);
```

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 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.

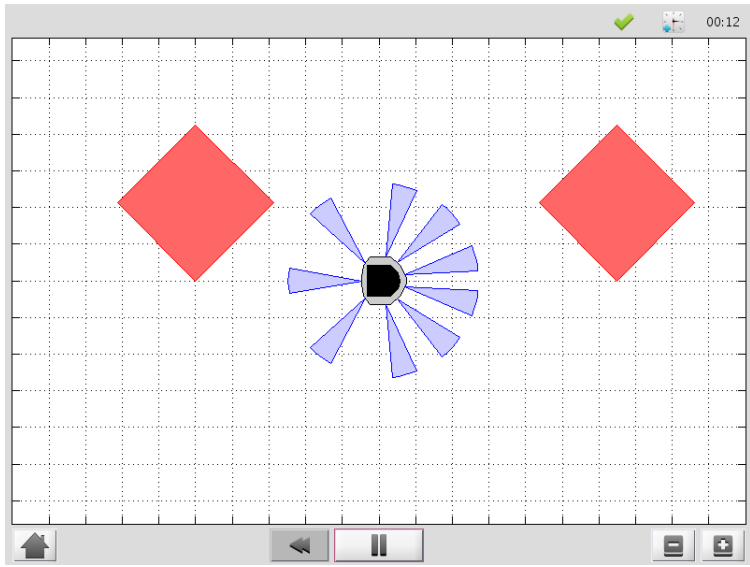


Figure 3: Simulator

4 Programming Exercises

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 programming exercises involve programming in MATLAB, you should familiarize yourself with MATLAB and its language. Point your browser to `http://www.mathworks.com/academia/student_center/tutorials`, and watch the interactive MATLAB tutorial.

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

4.2 Week 2

Start by downloading the robot simulator for this week from the "Programming Exercises" tab on Coursera course page. 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 2.3 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 video lectures and, for example the tutorial located at `www.orcboard.org/wiki/images/1/1c/OdometryTutorial.pdf`, cover 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/+khepera3/K3Supervisor.m`. This function is called every `dt` seconds, where `dt` is 0.01s (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 function $f(\Delta)$, 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/Khepera3.m`.

```
ir_distances = ir_array_values.*1;
% OR
ir_distances = zeros(1,9);
for i = 1:9
    ir_distances(i) = ir_array_values(i)*1;
end
```

The variable `ir_array_values` is an array of the IR raw values. Section 2.1 on "IR Range Sensors" defines a function $f(\Delta)$ that converts from distances to raw values. Find the inverse, so that raw values in the range $[18, 3960]$ are converted to distances in the range $[0.02, 0.2]$m. You can either do it by applying the conversion to the whole array (and thus apply it all at once), or using a `for` loop to convert each raw IR value individually. Pick one and comment the other one out.

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/+khepera3/K3Supervisor.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=1:9
   %   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) 0.2m. 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 "Programming Exercises" tab on the Coursera course page. 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.

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/+khepera3/K3Supervisor.m`.

```
obj.goal = [-1,0.5];
obj.v = 0.1;
obj.d_stop = 0.02;
```

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.

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 may overwrite `+simiam/+robot/Khepera3.m` with your own version from last week.

3. You should not overwrite `+simiam/+controller/+khepera3/K3Supervisor.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.

4.4 Week 4

Start by downloading the new robot simulator for this week from the "Programming Exercises" tab on the Coursera course page. 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 4 illustrates these points

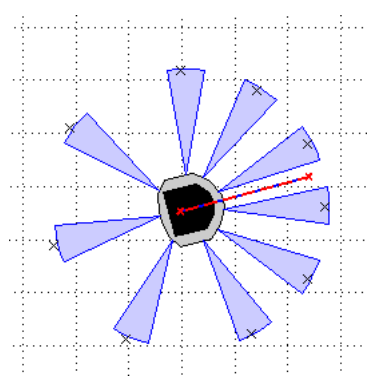


Figure 4: IR range to point transformation.

with a black cross. The 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, $u_o = \alpha_1 u_1 + \ldots + \alpha_9 u_9$.

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_sf`, 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_rf`, 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 4 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_rf` 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 not steer left or right.

(iii) Sum up the weighted vectors, $\alpha_i u_i$, into a single vector u_o.

(iv) Use u_o 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_o).

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/ +khepera3/K3Supervisor.m`.

```
obj.v = 0.1;
```

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 4. The robot should drive forward and collide with the wall. Note: The robot starts at an angle of $\frac{\pi}{12}$, instead of its usual angle of zero. 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_o) 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_o, 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/+khepera3/K3Supervisor.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.
4. You may replace the PID regulator in `+simiam/+controller/AvoidObstacles.m` with your own version from the previous week.

4.5 Week 5

Start by downloading the new robot simulator for this week from the "Programming Exercises" tab on the Coursera course page. This week you will be making a small improvement to the go-to-goal and avoid-obstacle controllers and testing two arbitration mechanisms: blending and hard switches. Arbitration between the two controllers will allow the robot to drive to a goal, while not colliding with any obstacles on the way.

1. Implement a simple control for the linear velocity, v, as a function of the angular velocity, ω. Add it to both `+simiam/+controller/GoToGoal.m` and `+simiam/+controller/AvoidObstacles.m`.

   So far, we have implemented controllers that either steer the robot towards a goal location, or steer the robot away from an obstacle. In both cases, we have set the linear velocity, v, to a constant value of 0.1 m/s. While this approach works, it certainly leave plenty of room for improvement. We will improve the performance of both the go-to-goal and avoid-obstacles behavior by dynamically adjusting the linear velocity based on the angular velocity of the robot.

   The actuator limits of the robot limit the linear velocity to a range of $[-0.3, 0.3]$ m/s and the angular velocity to a range of $[-2.765, 2.765]$ rad/s. However, it is important to remember that with a differential drive, we cannot, for example, drive the robot at the maximum linear and angular velocities. There is a trade-off between linear and angular velocities: linear velocity has to decrease for angular velocity to increase, and vice versa.

   Therefore, design and implement a function or equation for the linear velocity that depends on the angular velocity, such that the linear velocity is large when the *absolute value* of the angular velocity is small (near zero), and the linear velocity is small when the absolute value of the angular velocity is large. The linear velocity should not exceed 0.25 m/s and be no smaller than 0.075 m/s (because we want to maintain a minimum linear velocity to keep the robot moving).

   Add this function or equation to the bottom of the execute functions for both `+simiam/+controller/GoToGoal.m` and `+simiam/+controller/AvoidObstacles.m`.

   Note: This is just one way to improve the controllers. For example, one could improve the above strategy by letting the linear velocity be a function of the angular velocity *and* the distance to the goal (or distance to the nearest obstacle).

2. Combine your go-to-goal controller and avoid-obstacle controller into a single controller that blends the two behaviors. Implement it in `+simiam/+controller/AOandGTG.m`.

   It's time to implement the first type of arbitration mechanism between multiple controllers: *blending*. The solutions to the go-to-goal and avoid-obstacles controllers have been combined into a single controller, `+simiam/+controller/AOandGTG.m`. However, one important piece is missing. `u_gtg` is a vector pointing to the goal from the robot, and `u_ao` is a vector pointing from the robot to a point in space away from obstacles. These two vectors need to be combined (blended) in some way into the vector `u_ao_gtg`, which should be a vector that points the robot both away from obstacles and towards the goal.

   The combination of the two vectors into `u_ao_gtg` should result in the robot driving to a goal without colliding with any obstacles in the way. Do not use `if/else` to pick between `u_gtg` or `u_ao`, but rather think about weighing each vector according to their importance, and then linearly combining the two vectors into a single vector, `u_ao_gtg`. For example,

$$\begin{aligned} \alpha &= 0.75 \\ u_{\text{ao,gtg}} &= \alpha u_{\text{gtg}} + (1-\alpha) u_{\text{ao}} \end{aligned}$$

   In this example, the go-to-goal behavior is stronger than the avoid-obstacle behavior, but that *may* not be the best strategy. α needs to be carefully tuned (or a different weighted linear combination needs to be designed) to get the best balance between go-to-goal and avoid-obstacles.

3. Implement the switching logic that switches between the go-to-goal controller and the avoid-obstacles controller, such that the robot avoids any nearby obstacles and drives to the goal when clear of any obstacles.

   The second type of arbitration mechanism is *switching*. Instead of executing both go-to-goal and avoid-obstacles simultaneously, we will only execute one controller at a time, but switch between the two controllers whenever a certain condition is satisfied.

   In the `execute` function of `+simiam/+controller/+khepera3/K3Supervisor.m`, you will need to implement the switching logic between go-to-goal and avoid-obstacles. The supervisor has been extended since last week to support switching between different controllers (or states, where a state simply corresponds to one of the controllers being executed). In order to switch between different controllers (or states), the supervisor also defines a set of events. These events can be checked to see if they are true or false. The idea is to start of in some state (which runs a certain controller), check if a particular event has occured, and if so, switch to a new controller.

   The tools that you should will need to implement the switching logic:

   (i) Four events can be checked with the `obj.check_event(name)` function, where `name` is the name of the state:

   - `'at_obstacle'` checks to see if any of front sensors (all but the three IR sensors in the back of the robot) detect an obstacle at a distance less than `obj.d_at_obs`. Return `true` if this is the case, `false` otherwise.
   - `'at_goal'` checks to see if the robot is within `obj.d_stop` meters of the goal location.
   - `'unsafe'` checks to see if any of the front sensors detect an obstacle at a distance less than `obj.d_unsafe`.
   - `'obstacle_cleared'` checks to see if all of the front sensors report distances greater than `obj.d_at_obs` meters.

   (ii) The `obj.switch_state(name)` function switches between the states/controllers. There currently are four possible values that `name` can be:

   - `'go_to_goal'` for the go-to-goal controller.
   - `'avoid_obstacles'` for the avoid-obstacles controller.
   - `'ao_and_gtg'` for the blending controller.
   - `'stop'` for stopping the robot.

   Implement the logic for switching to `avoid_obstacles`, when `at_obstacle` is true, switching to `go_to_goal` when `obstacle_cleared` is true, and switching to `stop` when `at_goal` is true.

   Note: Running the blending controller was implemented using these switching tools as an example. In the example, `check_event('at_goal')` was used to switch from `ao_and_gtg` to `stop` once the robot reaches the goal.

4. Improve the switching arbitration by using the blended controller as an intermediary between the go-to-goal and avoid-obstacles controller.

   The blending controller's advantage is that it (hopefully) smoothly blends go-to-goal and avoid-obstacles together. However, when there are no obstacle around, it is better to purely use go-to-goal, and when the robot gets dangerously close, it is better to only use avoid-obstacles. The switching logic performs better in those kinds of situations, but jitters between go-to-goal and avoid-obstacle when close to a goal. A solution is to squeeze the blending controller in between the go-to-goal and avoid-obstacle controller.

   Implement the logic for switching to `ao_and_gtg`, when `at_obstacle` is true, switching to `go_to_goal` when `obstacle_cleared` is true, switching to `avoid_obstacles` when `unsafe` is true, and switching to `stop` when `at_goal` is true.

How to test it all

To test your code, the simulator is set up to either use the blending arbitration mechanism or the switching arbitration mechanism. If `obj.is_blending` is `true`, then blending is used, otherwise switching is used.

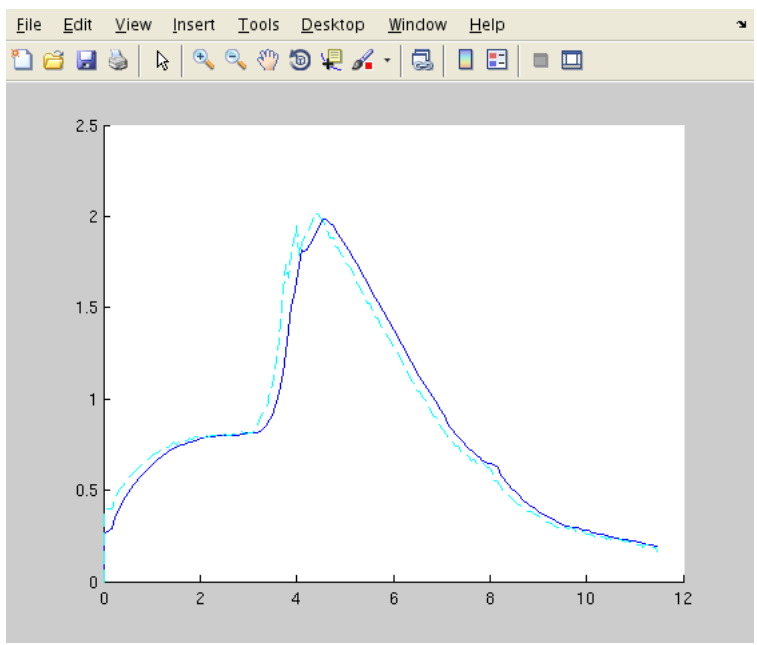
Here are some tips to the test the four parts:

1. Test the first part with the second part. Uncomment the line:

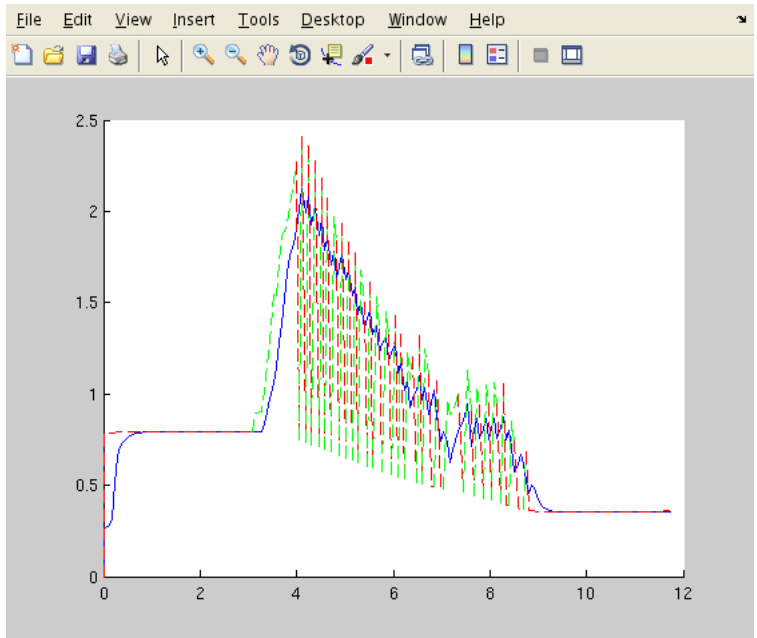
   ```
   fprintf('(v,w) = (%0.3f,%0.3f)\n', outputs.v, outputs.w);
   ```

   It is located with the code for the blending, which you will test in the next part. Watch `(v,w)` to make sure that when one increases, the other decreases.

2. Test the second part by setting `obj.is_blending` to `true`. The robot should successfully navigate to the goal location $(1, 1)$ without colliding with the obstacle that is in the way. Once the robot is near the goal, it should stop (you can adjust the stopping distance with `obj.d_stop`). The output plot will likely look something similar to:

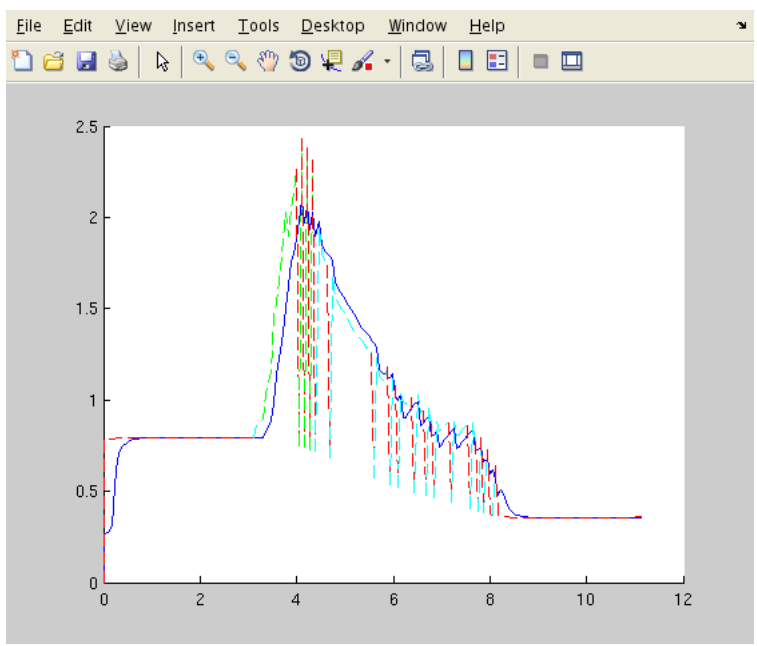


3. Test the third part by setting `obj.is_blending` to `false`. The robot should successfully navigate to the same goal location $(1, 1)$ without colliding with the obstacle that is in the way. Once the robot is near the goal, it should stop. The output plot will likely look something similar to:



Notice that the blue line is the current heading of the robot, the red line is the heading set by the go-to-goal controller, and the green line is the heading set by the avoid-obstacles controller. You should see that the two switch very quickly and often when next to the goal. Also, you will see many messages in the MATLAB window stating that a switch has occurred.

4. Test the fourth part in the same way as the third part. This time, the output plot will likely look something similar to:



Notice that the controller still switches, but less often than before. Also, it now switches to the blended controller (cyan line). Depending on how you set `obj.d_unsafe` and `obj.d_at_obs`, the

number of switches and between which controllers the supervisor switches may change. Experiment with different settings to observe their effect.

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. The simulator has seen a significant amount of changes from last week to support this week's programming exercises. It is **recommended** that you do not overwrite any of the files this week with your solutions from last week.

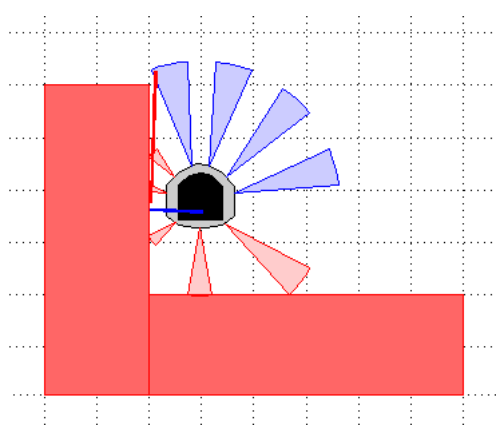
2. However, you can selectively replace the sections delimited last week (by `START/END CODE BLOCK`) in `GoToGoal.m` and `AvoidObstacles.m`, as well as the sections that were copied from each into `AOandGTG.m`.

4.6 Week 6

Start by downloading the new robot simulator for this week from the "Programming Exercises" tab on the Coursera course page. This week you will be implementing a wall following behaviour that will aid the robot in navigating around obstacles. Implement these parts in `+simiam/+controller/+FolloWall`.

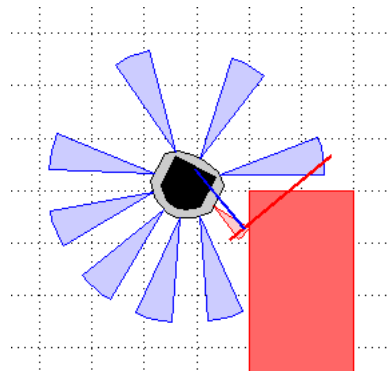
1. Compute a vector, $u_{fw,t}$, that estimates a section of the obstacle ("wall") next to the robot using the robot's right (or left) IR sensors.

   We will use the IR sensors to detect an obstacle and construct a vector that approximates a section of the obstacle ("wall"). In the figure, this vector, $u_{fw,t}$ (`u_fw_t`), is illustrated in red.



   The direction of the wall following behavior (whether it is follow obstacle on the left or right) is determined by `inputs.direction`, which can either be equal to `right` or `left`. Suppose we want to follow an obstacle to the `left` of the robot, then we would could use the left set of IR sensors (1-4). If we are following the wall, then at all times there should be at least one sensor that can detect the obstacle. So, we need to pick a second sensor and use the points corresponding to the measurements from these two sensors (see avoid-obstacles in Week 4) to form a line that estimates a section of the obstacle. In the figure above, sensors 2 and 3 are used to roughly approximate the edge of the obstacle. But what about corners?

   Corners are trickier (see figure below), because typically only a single sensor will be able to detect the wall. The estimate is off as one can see in the figure, but as long as the robot isn't following the wall too closely, it will be ok.



An example strategy for estimating a section of the wall is to pick the two sensors (from IR sensors 1-4) with the smallest reported measurement in `ir_distances`. Suppose sensor 2 and 3 returned the smallest values, then let p_1 `= ir_distances_rf(:,2)` and p_2 `= ir_distances_rf(:,3)`. A vector that estimates a section of the obstacle is $u_{fw,t} = p_2 - p_1$.

Note: It is important that the sensor with smaller ID (in the example, sensor 2) is assigned to p_1 (`p_1`) and the sensor with the larger ID (in the example, sensor 3) is assigned to p_2 (`p_2`), because we want that the vector points in the direction that robot should travel.

The figures correspond to the above example strategy, but you may want to experiment with different strategies for computing $u_{fw,t}$. A better estimate would make wall following safer and smoother when the robot navigates around the corners of obstacles.

2. Compute a vector, $u_{fw,p}$, that points from the robot to the closest point on $u_{fw,t}$.

   Now that we have the vector $u_{fw,t}$ (represented by the red line in the figures), we need to compute a vector $u_{fw,p}$ that points from the robot to the closest point on $u_{fw,t}$. This vector is visualized as blue line in the figures and can be computed using a little bit of linear algebra:

$$u'_{fw,t} = \frac{u_{fw,t}}{\|u_{fw,t}\|}, \quad u_p = \begin{bmatrix} x \\ y \end{bmatrix}, \quad u_a = p_1$$
$$u_{fw,p} = (u_a - u_p) - ((u_a - u_p) \cdot u'_{fw,t})u'_{fw,t}$$

   $u_{fw,p}$ corresponds to `u_fw_p` and $u'_{fw,t}$ corresponds to `u_fw_tp` in the code.

   Note: A small technicality is that we are computing $u_{fw,p}$ as the the vector pointing from the robot to the closest point on $u_{fw,t}$, as if $u_{fw,t}$ were infinitely long.

3. Combine the two vectors, such that it can be used as a heading vector for a PID controller that will follow the wall to the right (or left) at some distance d_{fw}.

   The last step is to combine $u_{fw,t}$ and $u_{fw,p}$ such that the robot follows the obstacle all the way around at some distance d_{fw} (`d_fw`). $u_{fw,t}$ will ensure that the robot drives in a direction that is parallel to an edge on the obstacle, while $u_{fw,p}$ needs to be used to maintain a distance d_{fw} from the obstacle.

   One way to achieve this is,

$$u'_{fw,p} = u_{fw,p} - d_{fw}\frac{u_{fw,p}}{\|u_{fw,p}|},$$

   where $u'_{fw,p}$ (`u_fw_pp`) is now a vector points towards the obstacle when the distance to the obstacle, $d > d_{fw}$, is near zero when the robot is d_{fw} away from the obstacle, and points away from the obstacle when $d < d_{fw}$.

   All that is left is to linearly combine $u'_{fw,t}$ and $u'_{fw,p}$ into a single vector u_{fw} (`u_fw`) that can be used with the PID controller to steer the robot along the obstacle at the distance d_{fw}.

   (*Hint*: Think about how this worked with u_{ao} and u_{gtg} last week).

How to test it all

To test your code, the simulator is set up to run `+simiam/+controller/FollowWall.m`. First test the follow wall behaviour by setting `inputs.direction = 'left'` in `+simiam/+controller/+khepera3/K3Supervisor.m`. This will test the robot following the obstacle to its left (like in the figures). Then set `inputs.direction = 'right'`, and changed in `settings.xml` the initial theta of the robot to π:

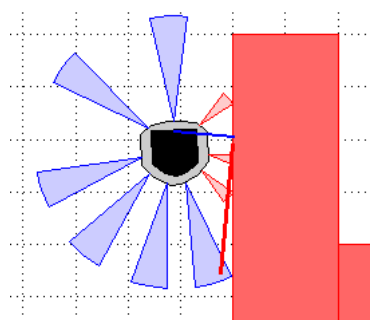
```
<pose x="0" y="0" theta="3.1416" />
```

The robot is set up near the obstacle, so that it can start following it immediately. This is a valid situation, because we are assuming another behavior (like go-to-goal) has brought us near the obstacle. Here are some tips to the test the three parts:

1. Set `u_fw = u_fw_tp`. The robot starts off next to an obstacle and you should see that the red line approximately matches up with the edge of the obstacle (like in the figures above). The robot should be able to follow the obstacle all the way around.

   Note: Depending on how the edges of the obstacle are approximated, it is possible for the robot to peel off at one of the corners. This is not the case in the example strategy provided for the first part.

2. If this part is implemented correctly, the blue line should point from the robot to the closest point on the red line.



   Note: Recall that we are computing $u_{fw,p}$ (the blue line) as the the vector pointing from the robot to the closest point on $u_{fw,t}$, as if $u_{fw,t}$ (the red line) were infinitely long. In the figure above, the red line is not drawn infinitely long, so the blue line does not touch the red line in this situation. However, if we were to extend the red line, we would see that the blue line correctly points to the closest point on the red line to the robot.

3. Set `obj.d_fw` to some distance in $[0.02, 0.2]$ m. The robot should follow the wall at approximately the distance specified by `obj.d_fw`. If the robot does not follow the wall at the specified distance, then $u'_{fw,p}$ is not given enough weight (or $u'_{fw,t}$ is given too much weight).

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. The only new addition to the simulator is `+simiam/+controller/FollowWall.m`. Everything else may be overwrite with the exception of `K3Supervisor.m`.