MATLAB Simulator for the iRobot Create

Code Documentation

Author: Cameron Salzberger
Advisors: Dr. Hadas Kress-Gazit
Dr. K-Y Daisy Fan
Supported By: The MathWorks
Updated: 2/18/2013 by Kevin Wyffels

CORNELL UNIVERSITY

Contents

Disclaimer

Introduction

This document is intended to help users who want a more in-depth understanding of the coding behind the simulator toolbox. This may be used to help modify the code to fit your specific robot setup, or to add or modify other functionalities. This document will describe the process of the simulation, and the algorithms within individual functions. This document is intended to supplement, but not replace, the comments contained within the code. Therefore, if a function is very simple or clearly explained by the comments, it will not be mentioned in this document. This is the case for many functions that simply 'get' or 'set' various robot object properties.

This document assumes that you have read both the User Guide and the documentation for the MATLAB Toolbox for the iRobot Create. It also assumes a more in-depth knowledge of MATLAB than is necessary to simply use the simulator. If you wish to modify all or part of the simulator, you should first have background knowledge of what you want to change. This may include knowledge of how to create and program GUIs, create and use timers, define and manipulate user-defined objects (object oriented programming), open and manipulate data and text files, and a background of basic physics (mechanics).

External resources will be handy even for experienced users. These are some that were referenced during the creation of the simulator.
GUIs:
http://www.mathworks.com/access/helpdesk/help/techdoc/ref/f16-40727.html
http://blinkdagger.com/matlab/
Timers:
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f9-38012.html
Object Oriented Programming:
http://www.mathworks.com/access/helpdesk/help/techdoc/ref/brk7uzk.html

Design Intent

Some of the user-defined requirements for this simulator are listed here so you can understand the design intent behind the programming choices.

- Must simulate the robot movement relatively close to reality
- Physics need only be simulated to the extent of collision, corners, and wall friction
- Autonomous control program must work on both the real Create and the simulator without modification
- One input argument to the control program is allowed
- Simulator must be started first, before starting autonomous program
- Must have a GUI that is intuitive to use with minimal MATLAB knowledge
- 'Helper' GUIs (input file builders and debugger) must be independent of the simulator and the robot object definition

Simplifications and Known Issues

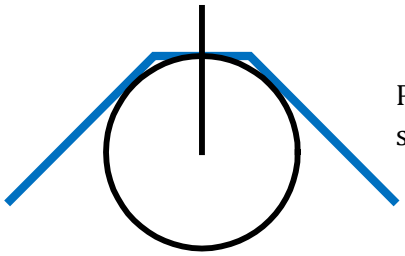
Physics Engine

The physics engine treats all walls and lines on the ground as linear, not rectangular. This is done to simplify the computations by not worrying about thickness of the wall. This assumption is only valid if the walls are thin compared to the radius of the robot (about 1-2 cm thick). If the walls are any thicker they will have to be modeled as rectangles.

The simulator only allows for straight lines to be used in creating all shapes for walls and lines. Multiple lines may be used to make a fairly accurate representation of a curved surface, but straight surfaces would probably fit the needs of the robot better.

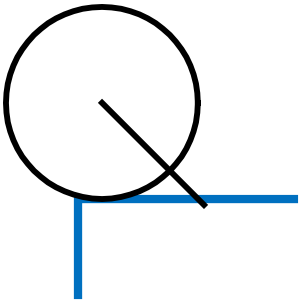
There are no 'out-of-bounds' areas in the simulator maps. Therefore, the user is able to place the robot over a wall or in an area that would normally be inaccessible. The behavior exhibited by the robot will be irregular in these instances, and not representative of reality. Note that the user is unable to place obstacles or the robot off of the axes by clicking elsewhere in the figure window. However, they are able to use the toolbar to pan and zoom to different areas of the map, and the robot is able to drive off the map, so this does not constitute an 'out-of-bounds'.

If colliding with more than two walls, the simulation only takes into account the two closest walls. The simulator should still work when it's hitting more walls since if the robot starts to move towards the undetected wall, it will become one of the two walls that is detected. However there is the possibility that the robot could inch its way through a corner by bouncing off of different walls but always heading towards them. This should not be a problem though, since most maps should have corners between two walls that are larger than the robot.

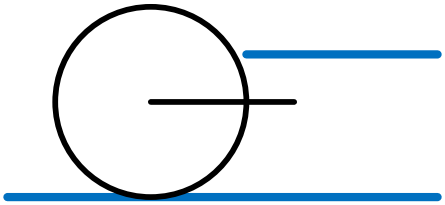


Problematic situation

If colliding with the side of a wall and a corner, the simulator only takes into account the wall. This is to avoid confusion when contacting something like a right-angle corner, so that the robot doesn't try to rotate around the corner while also sliding along the wall. However, this can cause problems when the corner is not connected to the wall.

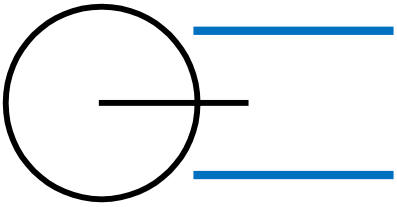


Intended situation



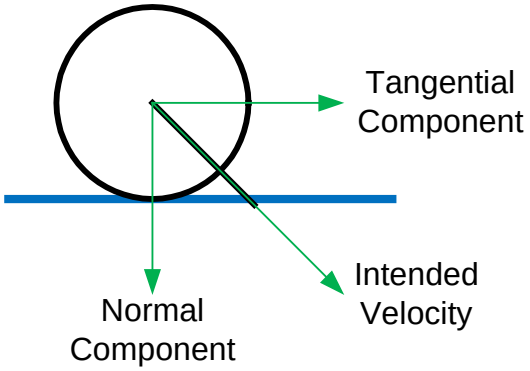
Problematic situation

If colliding between multiple corners, the simulator only takes into account the closest corner. This is to simplify the computation, but can run into similar problems as with the multiple walls case. If there are multiple corners of walls the robot can enter between the walls and get stuck.



Problematic situation

The simulator assumes force is proportional to intended velocity. However, the actual force produced by the robot is unknown, so this assumption may be completely invalid. This is done to calculate the reduction in velocity due to friction while sliding along a wall. An example calculation is below.



$\vec{v}_t - tangential\ velocity$
$\vec{v}_{int} - intended\ velocity$
$\hat{n} - unit\ vector\ normal\ to\ the\ wall$
$\hat{t} - unit\ vector\ tangential\ to\ wall$
$\mu_k - kinetic\ friction\ coefficient$
$\vec{v}_t = \vec{v}_{int} \cdot \hat{t} - (\vec{v}_{int} \cdot \hat{n}) * \mu_k$

The simulator assumes there is only friction from the walls, not from the wheels on the ground. Therefore if the robot is not contacting any walls, and is told to go forward at a certain velocity, the robot will move forward at that velocity.

The simulator also assumes that the wheels never get stuck, and that they will never slip except when running into a wall. This is not realistic, but it is difficult to model random wheel slips or catching. However, the odometry is not assumed to be perfect, since the noise added should be relative to the errors in odometry.

The simulator assumes infinite acceleration and constant velocity. If a change velocity command is sent to the robot, it changes to that velocity instantly and remains at that velocity until told otherwise. This makes simulating sensors that use acceleration difficult. Therefore, no IMU functionality was added to the simulator.

The simulator assumes that there is no bouncing from collisions into walls. This is somewhat valid due to the robot moving at low speeds, but there is still always some recoil that may affect the position or direction of the real Create in action.

The simulator assumes that beacons are not obstacles and can be driven over. Though colored golf balls are probably the optimal beacons to use for the real Create, the simulator assumes that they are more akin to circles of colored tape on the ground. The simulator also assumes that virtual wall emitters can be driven through. This is not the case at all, but it would be too complicated to model reality. If the real Create hits a virtual wall emitter, it will probably knock it over or at least move it. To promote reality in the simulator and avoid damage with the real Create, the robot should probably be programmed to move away from virtual walls.

Sensors

The simulator assumes that all sensor noise can be modeled as normally distributed about a mean. For many sensors, this mean will be 0, but this is more fully explained in the User Guide on page 14. An exception to this is that the sonar noise values are limited such that they do not cause the sensor to saturate. That is, if the true distance is within sensor range, the noisy measurement will also be within sensor range. This was done to avoid sparse data for noisy sensors, and is not necessarily a reflection of true sonar behavior. This can result in a deviation from Gaussian noise for sensors whose noise parameters are relatively large compared to their range.

The virtual wall field is assumed to be triangular and without noise. Through experimentation, it was determined that the virtual wall field is not actually triangular, but more diamond-shaped or even pentagonal. However, this would be even more difficult to model and unnecessarily complicated. In reality, the effective linear and angular range of the virtual wall beam varies widely, but these can be adjusted to the closest observed parameters as explained in this document on pages 25-26.

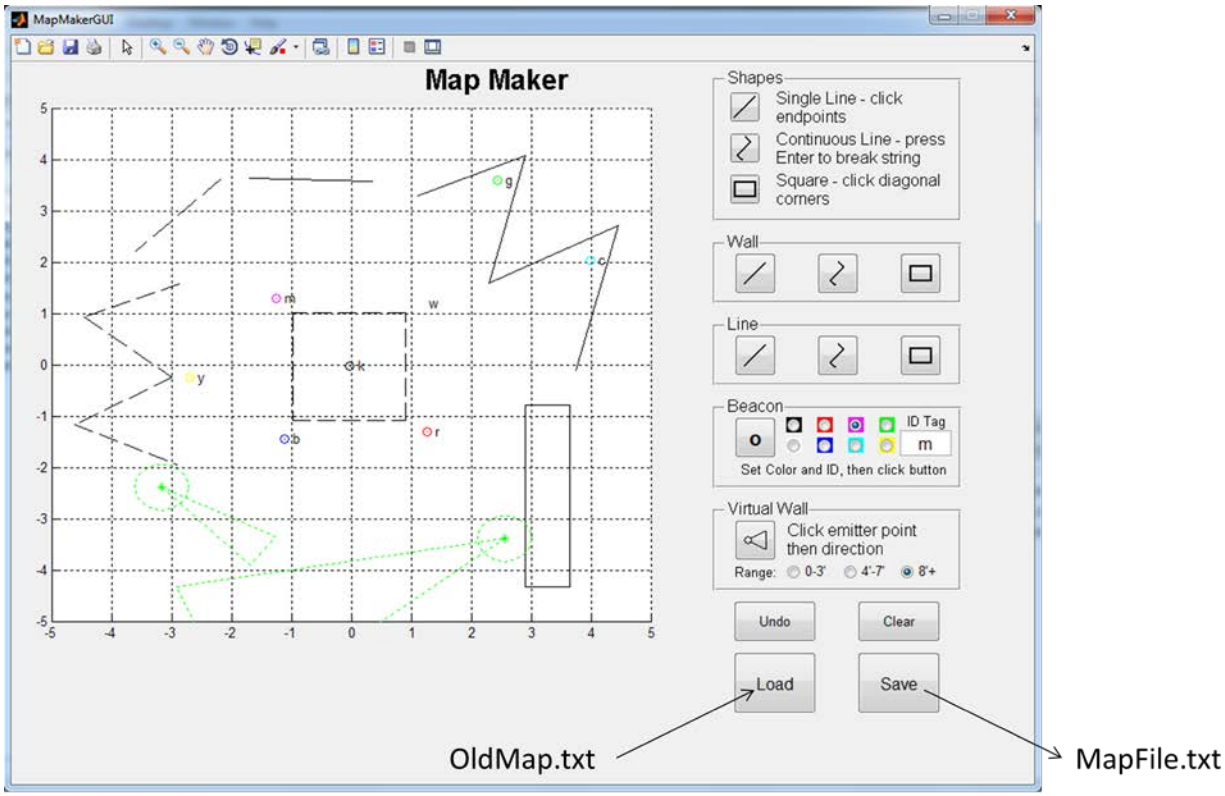
The virtual wall halo is modeled as being emitted from same location as the beam. In reality (at least with the Virtual Wall Scheduler product) the halo emitter is a few cm behind the beam emitter. However, this is small enough of a difference that it is justified as being modeled at the same location.

The bump sensors and overhead localization system are modeled as having no noise. The bump sensors may or may not be activated on a glancing hit with the wall, and also may activate either the front or one of the side sensors in the same location depending on how hard the bumper is pushed. However, they tend to perform consistently enough to model the sensor as noiseless. The overhead localization system is supposed to be a very accurate method of determining the position and orientation of the robot. This means that the noise should be low enough to be negligible. More information about the system and modifying the code relating to it can be found on page 28.

The overhead localization system is assumed to be equally accurate everywhere on the map. This is not necessarily a realistic assumption, but the variations in accuracy based on location and orientation of the robot can be difficult to calculate and very dependent on the system. It is also assumed to be able to detect the robot anywhere on the infinite plane, which is impossible for any real system. To improve the usage of the robot in reality and realism in the simulator, a wall should probably be constructed around the border of the region of valid readings from the overhead localization system.

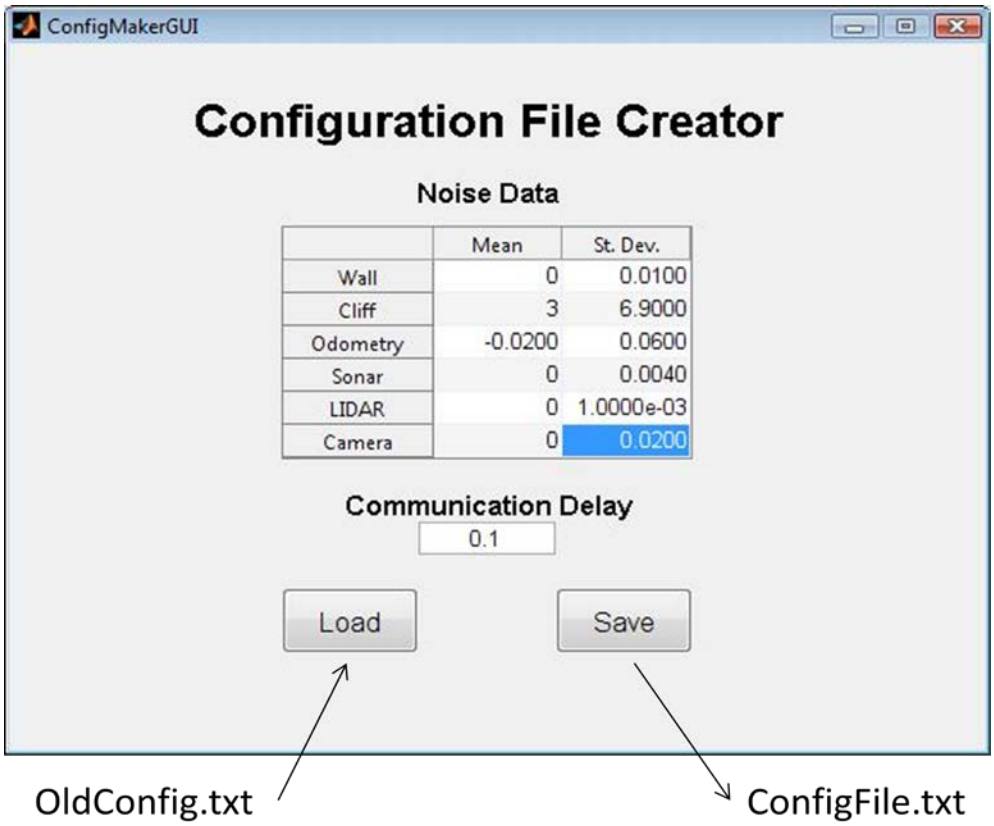
Structure of the Program

Map File Generation



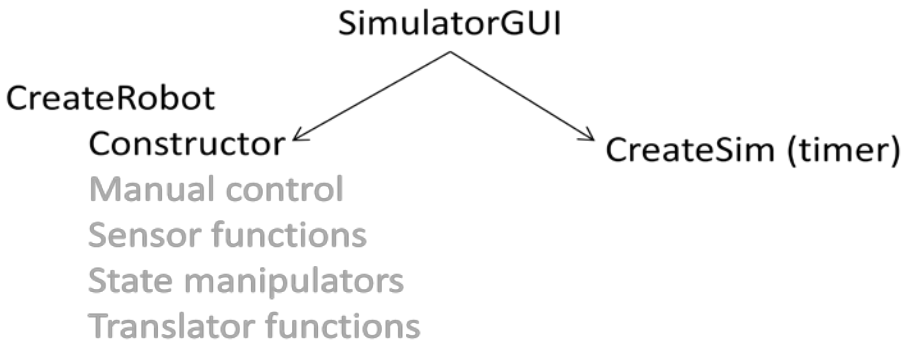
One of the first steps is to generate the map file for input into the simulator. This can be done using `MapMakerGUI`, or any text file editor. The generation process is explained more in the User Guide on pages 11-13.

Configuration File Generation



The other first step is to generate the configuration file for input into the simulator. This can be done using `ConfigMakerGUI`, or any text file editor. The generation process is explained more in the User Guide on pages 14-16.

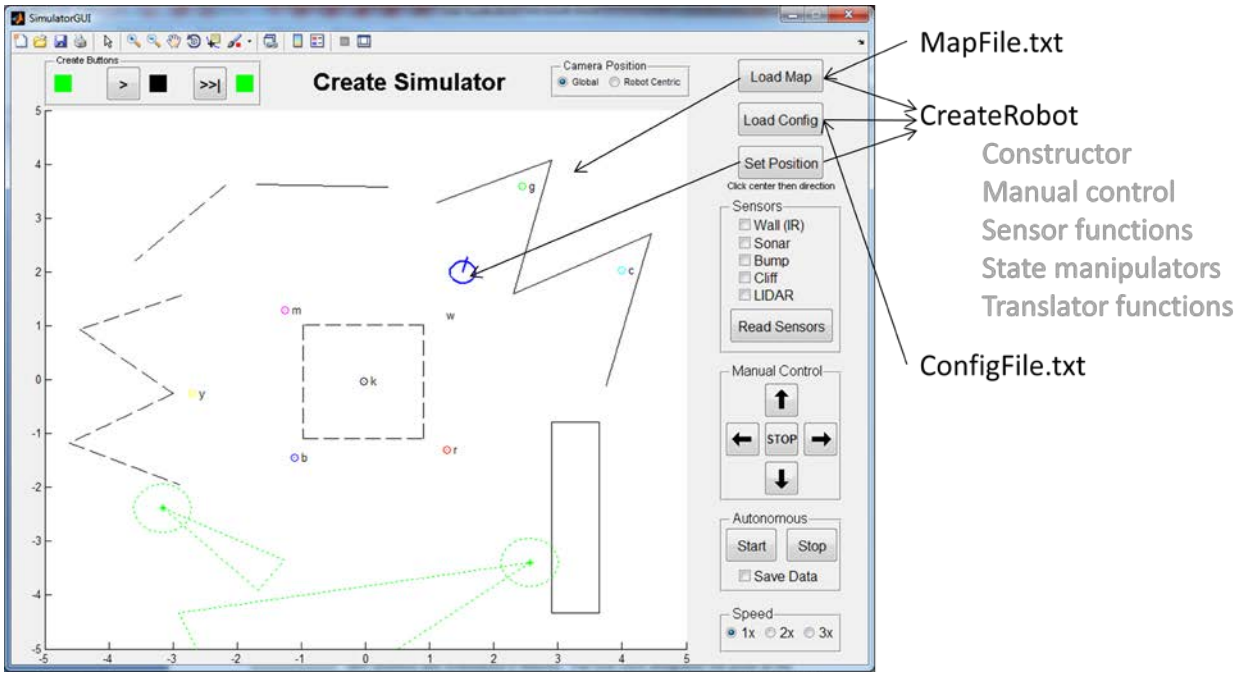
Simulator Initialization



When `SimulatorGUI` is called, it performs several tasks during the initialization process. Two of the most important are creating the robot object, and setting up the timer object. The robot object is an instance of the user-defined class `CreateRobot`, which contains a great many properties and non-static methods. The timer object has the name `CreateSim`, and the task to update the simulator at a regular interval.

`SimulatorGUI` will pass the handles to all the GUI controls to the robot object during the construction method. These handles will be stored in the property `handlesGUI`. `SimulatorGUI` will also set up the timer function `updateSim` so that it is passed the robot object and the handles to the GUI controls as input arguments.

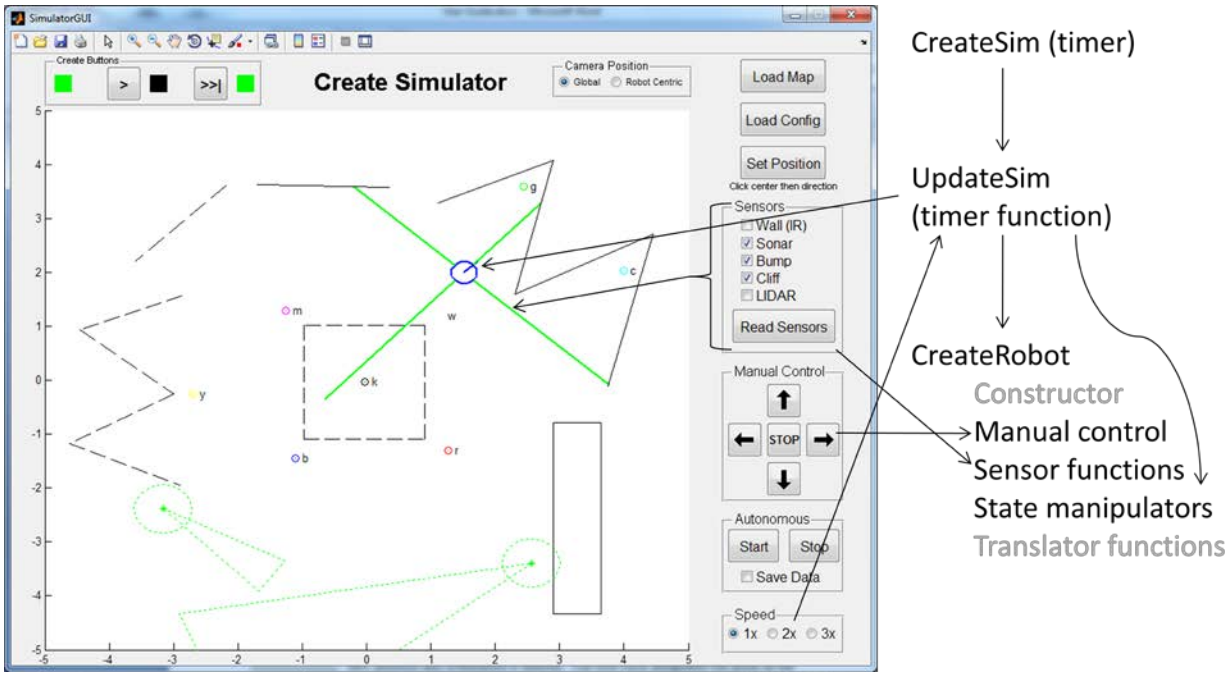
Environment Setup



Once `SimulatorGUI` is open, the user will need to set up the environment and other parameters of the robot. The simulator is immediately usable without executing these steps though. The map will default to blank, the noise and communication delay parameters will default to zero, and the position of the robot will default to the origin facing right. To adjust these values the user will load the map and configuration files, and set the robot's position and orientation manually.

When the map or configuration file is loaded, the data is imported from the text file, parsed in the `SimulatorGUI` program, and stored in the robot object. The map data will be stored in the properties `mapWalls`, `mapLines`, `mapBeacs`, and `mapVWalls`. The noise data from the configuration file will be stored in the property `noise`, and the communication delay will be stored in the property `comDelay`.

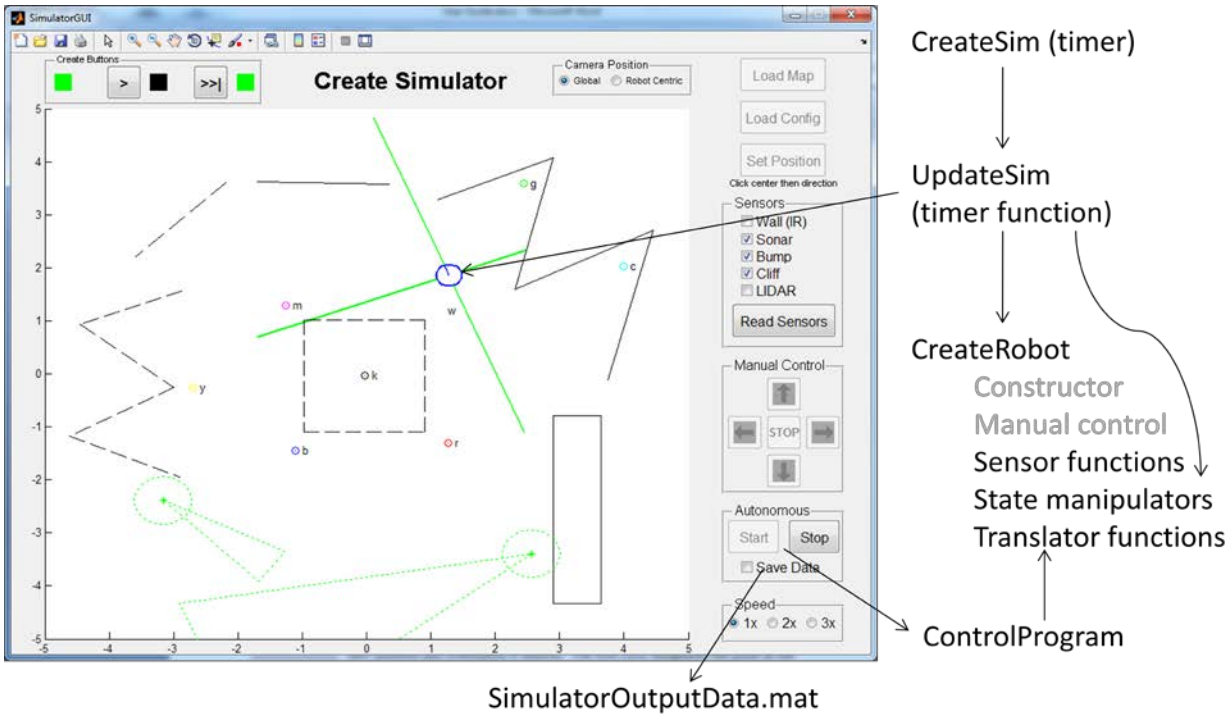
Manual Mode



Manual mode in the simulator is automatically active after initialization and while autonomous code is not being executed. This allows the user to drive the robot with the keyboard or the GUI controls, as well as visualize the sensors and reading their values at any point in time. The use of these functionalities is explained more fully in the User Guide on pages 18-19.

During both manual and autonomous mode, the timer object (`CreateSim`) will periodically call the function `updateSim` to recalculate the position of the robot and update the plot accordingly. If the viewpoint control is set to Robot Centric, `updateSim` will shift the plot so that it is centered at the robot. Also, `updateSim` will call a function in `CreateRobot` to replot the positions of the sensors that are being visualized. There are a few more function calls and purposes in updateSim that will be explained in more detail on pages 45-46. The keyboard and manual control buttons on the simulator will call a function in `CreateRobot` that changes the intended velocities of the robot. The property values that are changed are `velInt` and `wInt`, for the intended linear and angular velocities respectively.

Autonomous Mode



When the user starts the autonomous control program, all manual movement controls are disabled. The simulator setup buttons are also disabled, along with the autonomous start button. If enabled and pressed during autonomous code execution, none of these would cause errors. They are disabled merely to ensure pure autonomous execution with no user input beyond what is possible on the real Create.

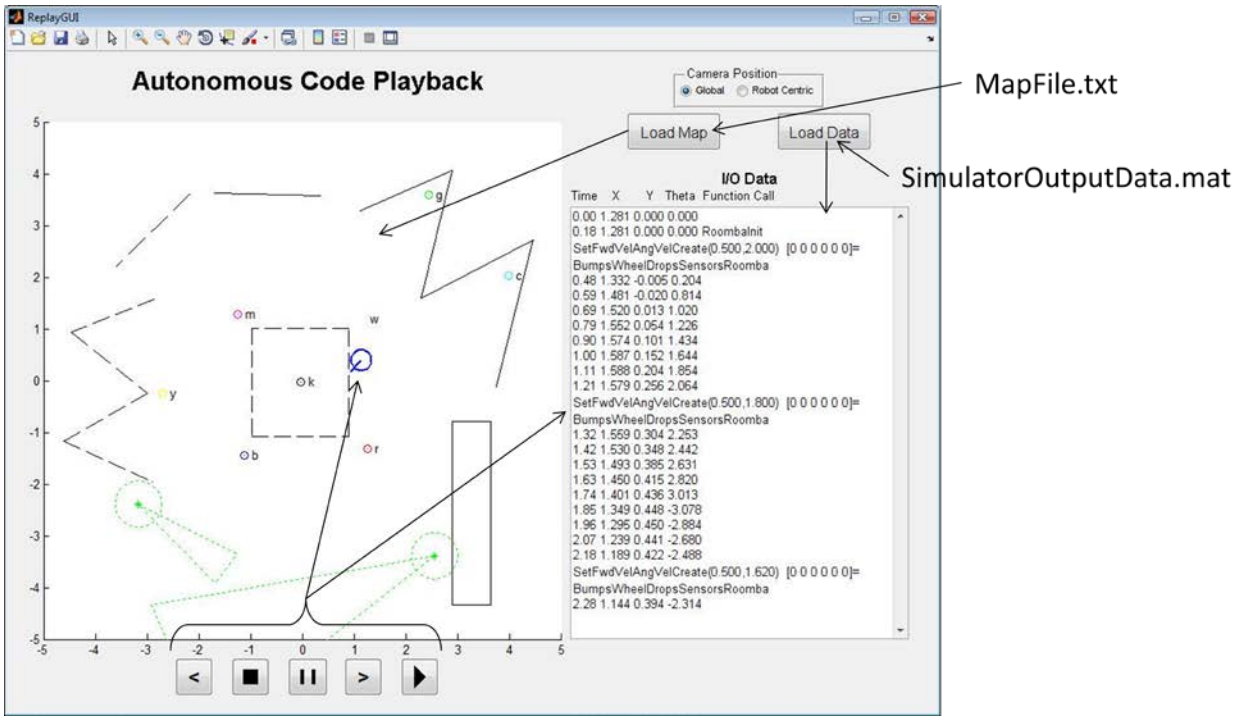
Other than the disabled functionalities, other processes still work mostly the same as in manual mode. The sensor visualization options still work. The timer still calls `updateSim`, which updates the robot position according to the intended velocities, among other things. Also, the Read Sensors button is still active, allowing users to get sensor data at any point in time. Note that this will temporarily stall the autonomous code execution while the sensor data is computed.

The main difference between autonomous and manual mode is that the autonomous control program sets the intended velocities and reads sensors. This is filtered through the Translator Functions in `CreateRobot`, that all have the same name and functionality of those in the MATLAB Toolbox for the iRobot Create.

If the Save Data checkbox is marked, when the autonomous code finishes executing (either by pushing the stop button or reaching the end of the code, but not by closing the GUI) a data file will be created for debugging purposes. The robot object property `dataHist` will be used to make this data file. More information about this data file can be found in the User Guide on page 21. Note that the data will be recorded to `dataHist` independent of whether the box is marked, so the box need only be marked by the end of the autonomous code execution to save the data. However, this may be considered inefficient and should possibly be changed.

Autonomous code execution is halted when the control program finishes execution, the autonomous Stop button is pressed, or the GUI is closed. When stopping before the code is done executing, `SimulatorGUI` will set a property (`autoEnable`) in the robot object so that the next Translator Function to be called will throw an exception (error) and the autonomous code will quit. The exception will be caught back in the simulator though, so no message will display in the command window. More information about this can be found on pages 20-21.

Replay and Debugging



The output data file from autonomous code execution can be used for debugging in two ways. It can be inspected by importing it into the workspace and looking at `datahistory` in the MATLAB variable editor. The alternative is to open `ReplayGUI` and import the data file. `ReplayGUI` usage is explained more in the User Guide on pages 20.

Function Descriptions

Conventions

The GUI controls are tagged with the convention of the uicontrol object style, underscore, and then name (e.g. `push_fwd` is the tag for the manual control button to drive forward in `SimulatorGUI`). Pushbuttons have the prefix '`push`', toggle buttons have '`toggle`', radio buttons – '`radio`', checkboxes – '`chkbx`', edit text boxes – '`edit`', static text boxes – '`text`', tables – '`table`', axes – '`axes`', and GUI figure – '`figure`'. Only uicontrol objects that will be used are specifically tagged, so some static text boxes and uibutton groups retain their default tags. The callback functions and other functions retain their default names and input arguments. This usually is in some form of `uicontroltag_Callback(hObject,eventdata,handles)`.

Many important values or matrices that are frequently used by the GUIs are stored in the User Data property of the various uicontrol objects. Usually the data will be related to the uicontrol object, but in some cases the logical connection is not there. Check the following function descriptions for the relevant GUI to determine where particular values are stored.

Sensor names are intended to remain constant throughout the toolbox to improve readability and reduce errors. These names are used in the function and variable names that pertain to that sensor (e.g. `genSensor` or `rangeSensor`). The designation for the infrared wall detector is '`IR`', for the bump sensors is '`Bump`', cliff sensors (signal strength) – '`Cliff`', wheel encoders (odometry) – '`Odom`', sonar devices – '`Sonar`', LIDAR device – '`Lidar`', camera (for detecting beacons) – '`Camera`', overhead localization system – '`Overhead`'.

In general, variable names follow a few rules with a couple exceptions. Commonly used variables have short names; rarely used variables have longer ones. Index or count variables are '`i`' or '`j`'. The variables '`x`' and '`y`' only ever refer to Cartesian positions, and '`th`' refers to angle relative to the positive x-axis (positive counter-clockwise). Variable names are lowercase except when made of multiple words (e.g. `twoWords`). Underscores are only used in local variables that require a great deal of clarity to distinguish between similar variable names.

All units are in meters, radians, and seconds unless otherwise specified.

MapMakerGUI

Important UserData
`figure_mapmaker` – Cell array of strings to be outputted to the map text file
`edit_beacon` – Boolean value of whether user edited the text box since starting the GUI

Map File Parsing Algorithm
This method of loading the map file is used in all three of `MapMakerGUI`, `SimulatorGUI`, and `ReplayGUI`. When the Load button is pressed, the user will be able to choose a file to input. Each

line is extracted as a string from the map file and processed. The rules for creating allowable map files can be found in the User Guide on page 13.

Each line is first processed through the built in functions `lower` and `strtrim` to account for uppercase characters and whitespace at the beginning and end of the string. Unfortunately this also makes beacon IDs convert to lowercase, but this is probably not a big deal. Next it is check to see if the line is empty or a comment (signified by '%'). If it is not, the first grouping of characters is extracted and compared to valid words. If it is recognized to be a wall, line, or virtual wall, the rest of the line is converted to numbers and stored in the appropriate matrix. If it is instead recognized as a beacon, the position and color values are converted to numbers and the ID is converted to a string, and all the information is stored in the cell array. If the first word is unrecognized, then the parser will display a warning to the command window.

The elements are then plotted to the map using simple plot-lines for walls, lines, and the edges of the virtual wall fields. A line approximation of a circle (20 lines) is used to show the halo radius of the virtual wall field. Different markers are used to plot the virtual wall emitter point, as well as the beacons. The beacons are also labeled by text next to the marker. Different colors and line styles are used to differentiate between the different map elements, although the beacon marker color is set by the color of the actual beacon.

Clear Functionality

The map and map data are cleared before a new map is loaded, or when the Clear button is pushed. This process involves deleting all the children on the plot (the children property of the axes contains handles to all plotted items). The User Data property that contains the map information is reset to contain only the comments that tell the user about formatting.

Undo Functionality

The Undo function erases the last action taken in most instances. When the Undo button is pressed, the plot is checked to see if it has any items plotted (children), verifying that there is actually something to undo. If there are plotted items, the element that was last plotted is deleted and the last string in the cell array containing the map data is deleted. It must first be checked that children is not empty to prevent errors, and to make sure the lines of the cell array that are the beginning comments are not deleted. Note that beacons involve two plotted items and virtual walls involve three, so multiple entries in children will need to be deleted for those cases.

Because of this method of implementation, the Undo button can be pressed until all elements are deleted. There is no 'redo' functionality. The Undo button will not be able to undo a Clear action, which is the reason for the confirmation pop-up. Undo will delete entries one by one if a Load action has just been performed. It also will not undo saving a file, but will undo normally instead.

Save Functionality
The Save button will open a dialogue box allowing the user to enter their filename. The file will only be saved if there is no file by the same name in the current directory, or if the user confirms that they wish to overwrite it. The save is executed by simply writing the map data to the file, one line per cell.

Map Size
The default size and placement of the map is 10 m x 10 m, centered at the origin. The user is able to pan and zoom to control their viewing of the map. Oddly though, MATLAB does not allow zooming out further than the initial zoom. The available area to place obstacles and such though is theoretically infinite. If you wish to change the default viewing size, enter `guide` into the MATLAB command prompt. Navigate to `MapMakerGUI.fig` and open it. Double click on `axes_map` to bring up the properties inspector, and change the values in the `XLim` and `YLim` properties to match the desired map size.

ConfigMakerGUI

Configuration File Parsing Algorithm
This method of loading the configuration file is used by both `ConfigMakerGUI` and `SimulatorGUI`. The initial loading and processing is done the exact same way as the map file parser in `MapMakerGUI`. The difference is once the first word is recognized as a valid sensor name; the rest of the line is converted to numeric values and stored in the table for noise. The first word could also be recognized as the communication delay indicator, in which case the edit text box for setting that is updated.

Save Functionality
The set up of the method of saving the data is similar to `MapMakerGUI`. The only difference is that the noise data must be extracted from the table and printed to the line. The precision of the numbers is adjustable inside the `fprintf` statement. The communication delay is extracted from the edit text box and printed last. That precision is similarly adjustable.

SimulatorGUI

Important UserData

`text_title` – Robot object of class `CreateRobot` (commonly stored in variable `obj`)

`figure_simulator` – Array of two handles for the plot of the robot

1. Circle
2. Direction line

`axes_map` – Array of handles to the visualization plots of the sensors

1. IR wall
2. Front sonar
3. Left sonar
4. Back sonar
5. Right sonar
6. Right bump
7. Front bump
8. Left bump
9. Right cliff
10. Front right cliff
11. Front left cliff
12. Left cliff
13. Right LIDAR
14. Front right LIDAR
15. Front LIDAR
16. Front left LIDAR
17. Left LIDAR

Enabling the Simulator

In previous versions of the simulator there were measures in place to ensure that the autonomous control program called the simulator translator functions rather than the functions from the MATLAB Toolbox for the iRobot Create. The first version set a flag in the autonomous control program to indicate a run with the simulator or real robot. This was undesired as the control program should be able to work without modification on both the simulator and real robot. In beta versions, the simulator set the MATLAB search path so that when `SimulatorGUI` ran, the simulator functions would be prioritized. When `SimulatorGUI` was closed, the path was reset.

The path setting was determined to be unnecessary as MATLAB automatically detects the class of the input arguments when calling a function. Since the input argument for all of the Translator Functions will be of the user-defined class `CreateRobot`, MATLAB will only make function calls to the public methods in there. This necessitates that the autonomous control program receives the instance of `CreateRobot` as its input argument so that it can access the correct `RoombaInit` and other functions.

Updating the Simulator

One other difficulty with the simulator is ensuring that updating of the visualization of the robot occurs independently of processing or function calls made by the control program. The first program that was used as inspiration for the simulator, updates the visualization at every Translator Function call. However, if there is heavy processing going on with the autonomous code, calls to functions `travelDist` or `turnAngle`, or simple pauses, this will enormously delay the visualization. To avoid these problems, a timer is used. This timer (`CreateSim`) regularly interrupts other processing and calls a function to update the simulator (by calling function `updateSim`). This interruption is done infrequently enough, and the update finishes fast enough, to avoid appearing to slow down the control program execution. This timer is created upon loading the simulator, and deleted upon exit to avoid excessive memory usage.

Visualization Efficiency

In order to speed up the updating of the map and robot visualization, the plotted items are only updated, not replotted. It would take far more time to clear the axes and replot the map, robot, and sensor visualization than to simply update the data of the currently plotted object. For this purpose, the handles to the robot lines and the sensor visualizations are stored in the robot object. At every update the robot representation is updated, and the sensor visualizations that are marked are updated as well.

Only a few issues come out of this method. One is that both the robot and the sensor visualization representations must be plotted upon the opening of the simulator. The sensor visualization checkboxes, though, default to being unmarked, so the plotted objects have their visibility property turned off. When the sensor is marked, the plot is turned visible and the position will be corrected at the next update. This means that when a sensor is marked, it is immediately plotted in the incorrect position, but this is fixed fast enough not to matter too much. The data itself coming from the sensor is correct at all times.

Input File Parsing Algorithms

The parsing algorithms for the map and configuration files are exactly the same as in `MapMakerGUI` and `ConfigMakerGUI`. See the appropriate section for details.

Create Button Functionality

There are three buttons on the real Create: Power, Play, and Advance. Obviously the Power button functionality does not need to be added, but the control program could use the Play or Advance buttons as input. However, the function `ButtonsSensorsRoomba` on the real Create, will only return true for the buttons while they are depressed. MATLAB GUIs on the other hand, do not have a function for holding and releasing a button. `ButtonsSensorRoomba` needs to read a value, so the callback function of a push button would not exactly work, since there is nothing that would indicate if the button is being held down or just released. Using the WindowsButtonDownFcn property would increase inefficiency because it would be called on every mouse click, and it is only active on a figure anyway, not over a uicontrol. The best available option seemed to be to make the Create button representations on the simulator into toggle buttons. This would allow the buttons to be

pushed and held, and multiple buttons to be depressed at one time. It does not exactly match the user interface on the real Create, but it seems to be close enough.

Autonomous Code Execution

In order to allow an arbitrarily named control program to execute, the built-in function `feval` needs to be used. When the Start button is pressed, the simulator will allow the user to select the control program file and then will run it with the input argument of the robot object. The autonomous code is then allowed to execute unhindered until either the simulator is closed or the Stop button is pushed. In both cases, the property `autoEnable` of the robot object is set to 0, meaning autonomous mode is disabled. The next Translator Function that tries to execute will throw an error.

Unless stopped by a try-catch statement, errors will always force-quit the function that throws the error, and the error will propagate up through all calling functions. When the error is thrown in the function `autoCheck`, it quits that function, as well as the Translator Function that called it, then the autonomous control program, then is received back at the `feval` statement. This will end all code execution in the `try` block that surrounds the `feval` statement, and continue execution in the `catch` block. This is done mostly to avoid outputting the error message to the command window, since even without the `catch` block the simulator will continue to work. If the error is not recognized as the one that is intended to terminate autonomous mode, the error is rethrown so it propagates out to the command window for debugging the control program.

The main problem with this method (besides poor programming practice) is delay. The autonomous code will continue to execute until the next call to a Translator Function. If there are frequent `pause` commands in the code, or a time-consuming computation, the code may not stop execution for some time. Also, if the autonomous code gets stuck in an infinite loop or computation that does not involve calls to Translator Functions, the Stop button will do nothing to break the cycle and the program may freeze. Fortunately, Ctrl+C works just as well as normal, and should also allow the simulator to keep running, at which point pushing the Stop button will re-enable manual mode.

CreateRobot

Constant Properties

These properties cannot be changed by any of the functions. They are used for setting the parameters relevant to the robot and the sensors that are not set in the configuration file. These values may be modified manually to adjust or exaggerate certain features in the simulator (e.g. friction). The use of these properties is explained sufficiently by the comments.

Other Properties

These properties can be set by any of the methods contained within `CreateRobot`, but the 'private' permissions make it impossible to extract or set them from external methods. If at least some of them were set to 'public', many of the get… or set… methods (e.g. `getMap`) would not need to be written. However, it seems to be better programming practice not to have 'public' properties.

The input-related properties `noise` and `comDelay`, as well as all of the `map` properties, will contain information from the input files. Also, `handlesGUI` is the handles structure for `SimulatorGUI`, mostly used for the LED and Play/Advance button functions.

The output-related properties are `autoEnable`, `timeElap`, and `dataHist`. `autoEnable` is used to determine if autonomous code is still running. `timeElap` is used purely to store the timestamp of the step in the data history output. The timer property `InstantPeriod` could also be used here, but the `tic-toc` functions are used for convenience. `dataHist` stores the output data, to possibly be saved at the end of autonomous mode execution.

The rest of the properties vary as the simulator runs. The `odom` properties store the odometry data, obviously. The `Int` properties (`velInt`, `wInt`) are set by the user or the control program to specify the intended velocities for input to the physics engine. Finally, the `Abs` properties give the absolute position and velocities of the robot are calculated by the physics engine and used to determine the new pose of the robot.

Constructor Function

`CreateRobot:`

This function follows the rules for user-defined object constructor functions that can be found at:
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/brd2m9e-1.html
It sets default values to all the properties that have not already been defined. The default is an empty map with no noise or communication delay. The robot starts stationary, at the origin, facing right, with the simulator in manual mode.

The reason for using `varargin` as the input parameter is so that the `handles` structure from `SimulatorGUI` can be passed in and stored in the object, but if the robot object is created from the command window instead of through `SimulatorGUI`, no input argument need be passed. This is to allow object creation for simple testing and debugging purposes. Not all of the simulator functionalities will work without running the GUI, but some methods can be tested.

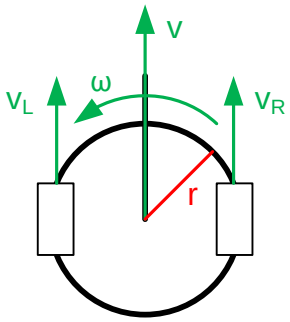
Simulator Control Functions

`manualKeyboard:`

This function is used to control the robot's movement during manual mode. The arguments are the intended increase or decrease in the robot's velocities, rather than the new intended velocities. This is done to avoid requiring that `SimulatorGUI` knows the robot's current velocities to set the new ones. However, that means that a different type of input is required to stop the robot's movement entirely, since `manualKeyboard(obj,0,0)` would just leave the velocities how they are. Thus, the stop commands will send `NaN` as the input argument.

The velocities are limited based on the physical restriction on the individual wheel speeds of the Create. Each wheel is only able to move at 0.5 m/s, so if the velocity combination results in exceed-

ing this, each wheel speed will be reduced to within the limits and the total velocities will be recalculated from that. The wheel speeds are calculated as shown below:



Wheel speeds from total speeds:

$$v_R = v + \omega r$$
$$v_L = v - \omega r$$

Total speeds from wheel speeds:

$$v = \frac{v_R + v_L}{2}$$
$$\omega = \frac{v_R - v_L}{2r}$$

`updateSensorVisualization:`

This function is called on every call to the timer function `updateSim`. Because the sensor functions take a bit of time to execute, this function will only attempt to update each sensor representation on the plot if that sensor's checkbox is marked. When a sensor's checkbox is marked, the plot for that sensor's representation is made visible. However, it is not updated until the next call to `updateSim`. This means that for a brief moment after marking a sensor checkbox, the plot will display in the wrong location. Even if all the checkboxes are marked, the simulation is not slowed down dramatically. However, to avoid clutter as well as increased computation, unneeded sensors should be unmarked.

Most of the sensors make a call to their generating function (e.g. `genSonar`) to receive the necessary data for plotting. However, since the call to `genLidar` produces readings for all of the points in the LIDAR sensor's range, it would be inefficient to call it in order to find only a couple distances. Therefore, the line orientations are calculated and calls are made to `findDist`, just the same as inside `genLidar`, but without so many iterations. Currently the LIDAR visualization only shows the five lines interspaced through the angular range.

`updateOutput:`

This function adds another row on the end of the cell array `dataHist` for output at the end of autonomous code execution. Note that it only inputs values for timestamp, position, and orientation, not for the function calls. The function call representations are added by `addFcnToOutput` at every call to a translator function.

`addFcnToOutput:`

This function inserts the representation of the translator function call into the current row of `dataHist` for output at the end of autonomous execution. All translator function calls made by the control program after one call to `updateSim` but before the next one will be retained in the last line and last cell of `dataHist`. If no translator function calls are made, that cell remains empty. If one call is made, then that cell will contain the string representation of that call. If multiple calls are made, then a that cell contains another cell array which holds all the string representations.

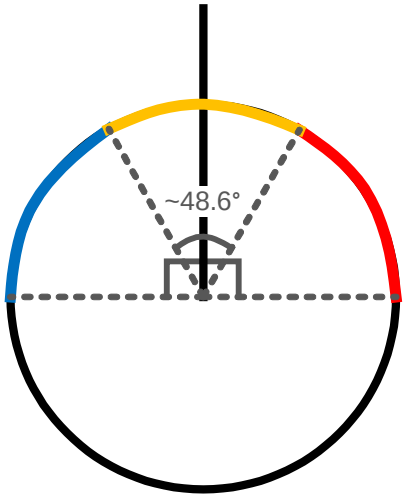
Sensor Functions

The noise is added to the sensors inside the generating functions (e.g. `genSonar`) because the sensor visualizations and command window output (from the Read Sensors button) make use of the generating functions instead of translator functions. This way, the noise can actually be visualized for demonstration purposes. Also, the Read Sensors button can be used to see what kind of data the autonomous code will receive when reading sensors in particular situations.

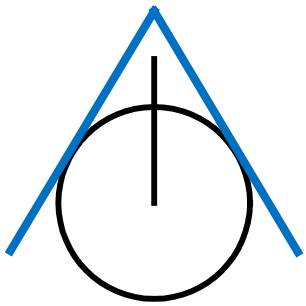
`genBump:`

To determine if the robot is intersecting a wall or not, this function makes use of `findCollisions`. The original implementation of `genBump` created a line representation of the robot circle (only the relevant angles), and checked it against each wall using `polyxpoly`. However, the algorithm used in `findCollisions` is theoretically faster, although it requires checking for collisions on all angles of the robot surface. The speed difference has not been tested.

Approximate effective angles of the bump sensors are below. Most hits within the orange region will trigger the front sensor. Hits within the blue and red regions will trigger the left or right sensors respectively. Note that the angle values were taken from experimentation, but cannot be taken as infallible. A sensor may not be activated at all on a glancing or soft hit. Whether the front versus one of the side sensors are activated also depends on the strength and angle of the hit on the wall. The simulator does not take these effects into account though. Despite these possibilities, there is no noise added into the bump sensor readings.



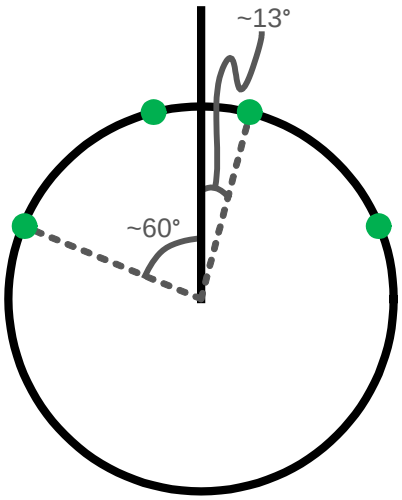
Note also that the real Create only has two physical bump sensors on the left and right side. Thus, when the front of the bumper is hit, both sensors are usually activated. Thus the toolbox for the real Create interprets this as a front bumper hit. Therefore, if the robot were to wedge itself in a corner when both the left and right bumpers are pressed by two different walls, the real Create will interpret this as a front bumper hit. The simulator has this functionality as well.



`genCliff:`

This function uses the same algorithm as `findCollisions`, but instead looks at lines instead of walls. The reasons for using this algorithm over `polyxpoly` are explained more under `genBump`. Note that since `findCollisions` only looks at walls, the entire algorithm is copied into `genCliff` with "`Lines`" substituted for "`Walls`". Coding inefficiency could be easily reduced by making `findCollisions` have an additional input deciding whether it's looking at walls or lines.

Approximate angles of the cliff sensors are below. In reality, the cliff sensors are only checking a single point on the ground, while it is the lines that have thickness. However, in the simulator, the lines have no thickness, so the point that the cliff sensors check is increased to a range. In the simulator, the cliff sensor will activate if a line intersects with the robot perimeter within 5.85° of the sensor.



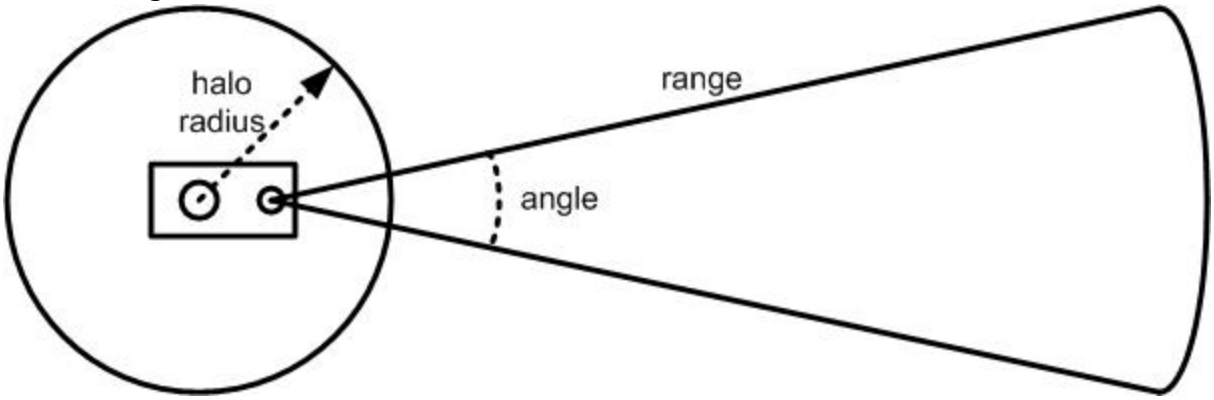
The cliff sensors on the real Create are actually reading the reflectivity of the ground. If there is no ground under the sensor, the amount of signal reflected back will be very low (usually 0%). If there is a black or dark surface under the sensor, reflectivity tends to be low (around 3%), and a lighter surface tends to reflect more (up to about 20%). The real Create toolbox function `Cliff[Direction]SensorRoomba` will only return a true value for actual cliffs, while `Cliff[Direction]SignalStrengthRoomba` will return the numerical percentage value based on reflectivity. The simulator duplicates this functionality, and returns false for all calls to `Cliff[Direction]SensorRoomba` (since there are no cliffs in the simulator). For `Cliff[Direction]SignalStrengthRoomba` it will return high values when over regular ground, and low values while over lines. The exact values depend on the inputted noise, but without noise they will be 21.5% and 1.5% respectively. The cliff sensor visualization though will show up if the value of the cliff sensor is below 5.4%, indicating it has detected a line.

`genIR:`

This function generates the reading for the infrared wall sensor. It makes use of `findDist`, despite the fact that the actual distance will not be used. This is to prevent coding inefficiency by replicating the majority of the `findDist` algorithm. There is not too much extra computation associated with using `findDist`, so there is little inefficiency.

`genVWall:`

The constants relating to the virtual walls are only used in this function. Therefore, they are only defined in this function, not as constant properties for the robot object as is customary. These constants are diagrammed below.



The virtual walls that were used in determining properties for the simulator are the iRobot Virtual Wall Schedulers. They have three settings for the power of the infrared signal, and the effective range and angle change depending on the setting, while the halo radius appears to stay the same. These values were determined by experimentation, and should be changed to more closely match the user's virtual wall system.

Halo radius: 0.45 m

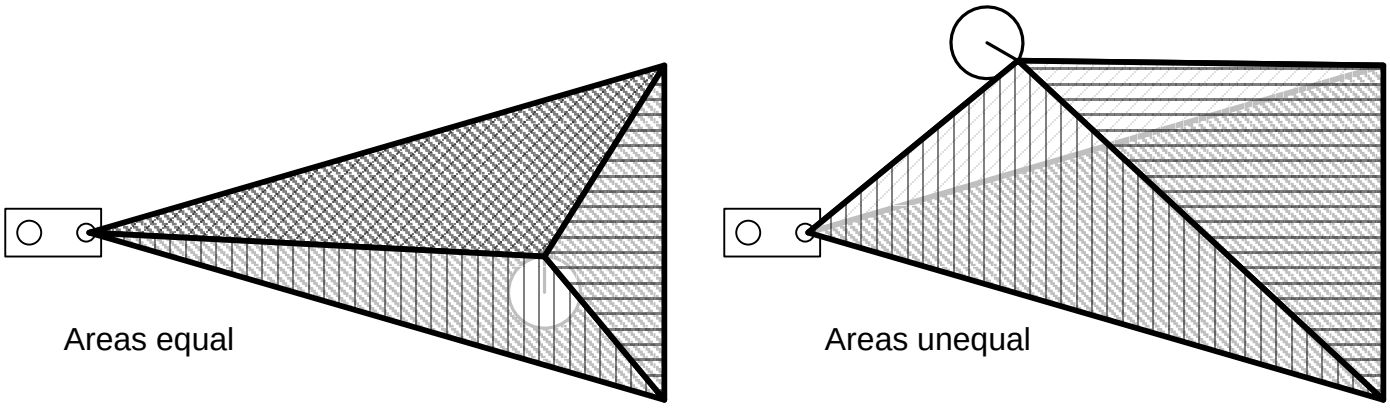
	0-3'	4'-7'	8'+
Range	2.13 m	5.56 m	8.08 m
Angle	19°	28°	35°

This function uses a slightly inefficient algorithm of checking to see if it is in the field of all of the virtual walls before completing. The efficiency could be improved by putting the check into a while loop that will terminate if it checks all the virtual walls, or if the robot is found to be in one of the fields. Efficiency could also be improved by only checking against virtual walls that have their emitter within range of the robot. Also since the algorithm to see if the robot is in the halos is quite simple, that check could be done first, before next seeing if it is within the triangular fields. These methods would need testing to see if they really would improve the processing time of this function though.

When checking the sensor against each individual wall, `genVWall` first compares the sensor position to that of the emitter to see if the robot is within the halo (a simple distance formula). If it is not in the halo, it uses an area algorithm to see if the sensor position is inside the triangular field. The area of a triangle is calculated from the vertices of the triangle using the determinant method:

$$Area = \frac{1}{2}\left|\det\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix}\right|$$

The function calculates the area of the triangular field, and then the area of the three triangles whose vertices are two vertices from the field and the sensor position. If the sensor is within the field, then the area of the field is equal to the sum of the areas of the other three triangles. If not, the area of the three triangles is greater than the area of the field.



Note that a tolerance is used to determine the equality of the areas since when MATLAB calculates the area algorithm; the areas tend to be off by a small amount due to rounding errors.

The final check if the robot is determined to be within the field of a particular virtual wall emitter is to ensure that there is no wall between the robot and the virtual wall. For this purpose, `findDist` is used. The range given is the distance from the emitter to the sensor. If that same distance is returned from `findDist`, then there are no walls in between the sensor and the emitter. Note that this assumes the walls are high enough to block the infrared signal from virtual walls (at least 10 cm high).

`genSonar:`
The simulator assumes that there are four sonar sensors, placed in the cardinal directions on the edges of the robot. This makes it easy to check each sensor since they are evenly spaced around the circumference. If the sensors on the user's robot are placed differently, the code for calculating the position and direction of the sensors will need to be changed in the `for` loop. If the sensors are not uniformly spaced, the position and direction may need to be hard-coded into this function. Note the difference between the order of output of `genSonar` ([front left right front]), and the sonar sensors that `ReadSonar` (right) and `ReadSonarMultiple` ([right front left back]) actually call.

Note that this function calls `randn` for calculating the noise in all iterations of the loop. Calculating the noise value before entering the loop will result in all sensors having the same noise. Using the same standard deviation and mean, but calling `randn` multiple times will result in different noise values for each sensor, which is closer to reality.

Note that the noise values are capped such that they cannot cause a sensor to saturate. That is, if the true distance is within sensor range, the noisy measurement will also be within sensor range. This was done to avoid sparse data for sensors with noise parameters large in comparison to their sensor range (not common in reality, but could arise in simulation), and is not necessarily a reflection of true sonar sensor behavior. This will also cause sensors with large noise parameters relative to their range to deviate from Gaussian noise.

`genLidar:`
The simulator assumes that the LIDAR sensor is located on the front of the robot. If the location is different, then the code to set the position should be changed in this function. Like `genSonar`, this function calls `findDist` to get the reading for each point in the LIDAR field of view. It also changes all readings below the minimum allowable to the minimum distance. It calculates the noise in all iterations so that the noise value for each data point is different.

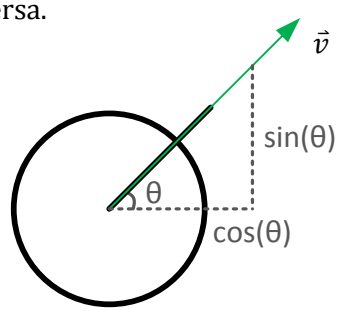
`genCamera:`
This function first uses the normal angle and distance formulae to get the heading and range to each beacon from the sensor. Then it checks the beacon to see if it is in range of the camera (within both the angular and linear range). Finally, it checks to see if there is a wall between the beacon and the camera, using the same method that `genVWall` uses. Every beacon within range that is not behind a wall is then recorded.

The same noise parameters are used for both the angle and distance readings. However, `randn` is called twice, so the two readings will not get the same noise value. This is not a very accurate representation of reality, but the simulator noise functionality was intended to just give some idea of what variance does to a program. If accuracy of noise is important for the user's simulator, the functionality should be added for separate noise values for angle and distance, or model the noise based on real readings.

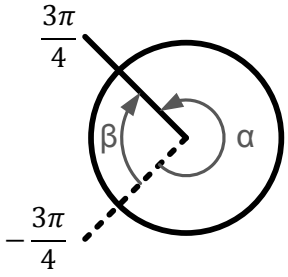
`updateOdom:`

This function is called at every execution of the timer function `updateSim`. Since `updateSim` should be executing quite frequently, the movement of the robot between two calls should be small. This can be used to simplify the algorithm in two ways. First, the odometry distance is calculated by using a linear approximation of the distance traveled between the previous location and the current one. The magnitude of the change in `odomDist` comes from a simple distance formula. The sign of the change is more complicated. If the robot is moving in the direction it is pointing, the odometry will increase, and vice-versa.

$$sign\left(\vec{v} \cdot \begin{bmatrix}\sin(\theta)\\ \cos(\theta)\end{bmatrix}\right)$$



The assumption that the angle turned is small also simplifies calculating the angular odometry. The angle turned on a given step is calculated from the start `thAbs`, and the end `thAbs`, with no knowledge of which path is taken. The path could be determined from `wAbs`, but that would be more difficult than just assuming that the robot turns the shorter amount. This is especially important when the robot turns through π or –π radians, since `thAbs` is automatically wrapped to between those values. In the diagram below, the odometry will be changed by the small angle of the two (β). It will be increased if `wAbs` is positive, and decreased if negative. This method may cause errors if the max allowable turning speed is large compared to the time between updates from the timer.



Similarly to the camera, odometry uses the same noise parameters for both the angle and distance sensor. The noise values are different though, since `randn` is called twice. This is inaccurate and could do with improving, in the same method as described under `genCamera`.

`genOverhead:`

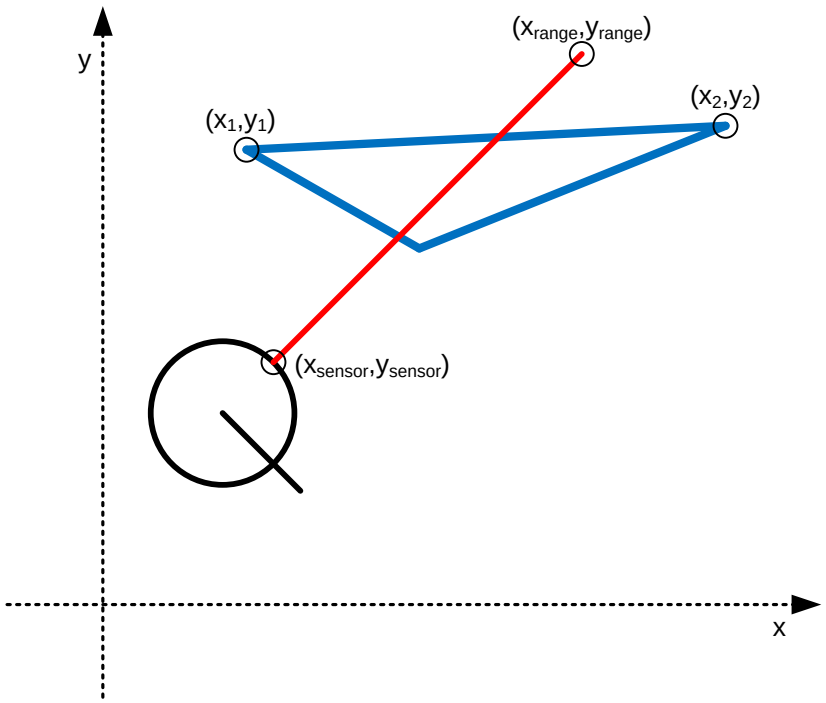
The overhead localization system is assumed to be very accurate, so it outputs the exact location and orientation of the robot with no noise. If the user's localization system is not as accurate, the functionality for noise should be added to the simulator. If some noise is desired, but accuracy is not important, one noise parameter could be used. Otherwise, separate noise parameters should be created for position and orientation. This will probably be more useful since 'drift' in the orienta-

tion reading of an object tends to be more of a problem than the position reading with these types of systems.

Computational Functions

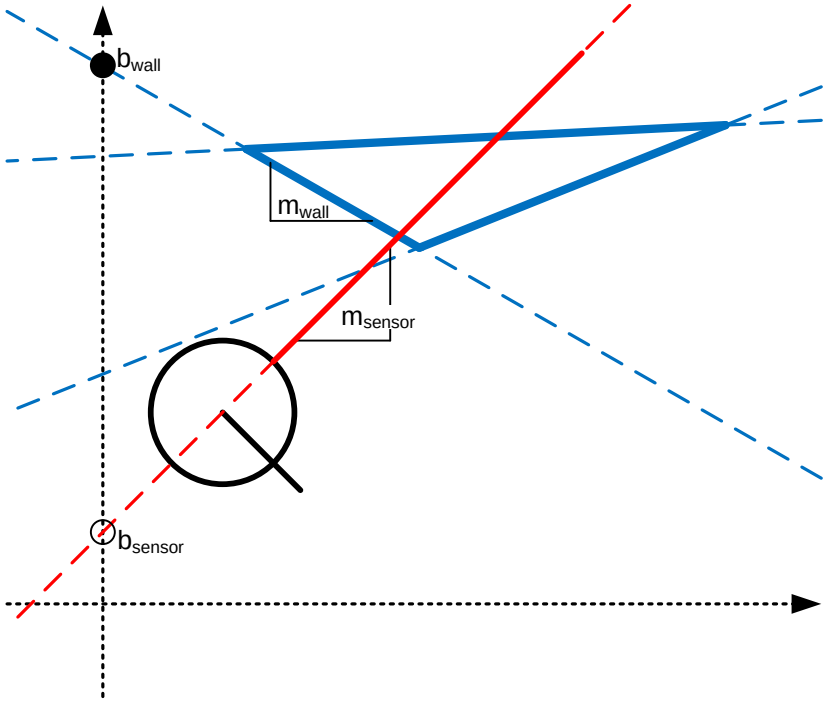
`findDist:`

This function is used to find the distance to the nearest wall as seen by a particular sensor. The known information is the location, orientation, and range of the sensor, as well as the endpoints of all the walls. From this, it is easy to get the end point of the sensor range. The example below is calculating the distance received by the left sonar sensor.



$$x_{range} = x_{sensor} + range * \cos(\theta_{sensor})$$
$$y_{range} = y_{sensor} + range * \sin(\theta_{sensor})$$

Next, the equations (of the form y = mx + b) for the sensor line and walls are calculated. It is theoretically faster to find the intersections of lines using the equations rather than using `polyxpoly`, although this has not been tested. The disadvantage is that the intersections are first found on the infinite sensor and wall lines, so they will require some pruning.



$$m_{sensor} = \frac{y_{range} - y_{sensor}}{x_{range} - x_{sensor}}$$

$$m_{wall} = \frac{y_2 - y_1}{x_2 - x_1}$$

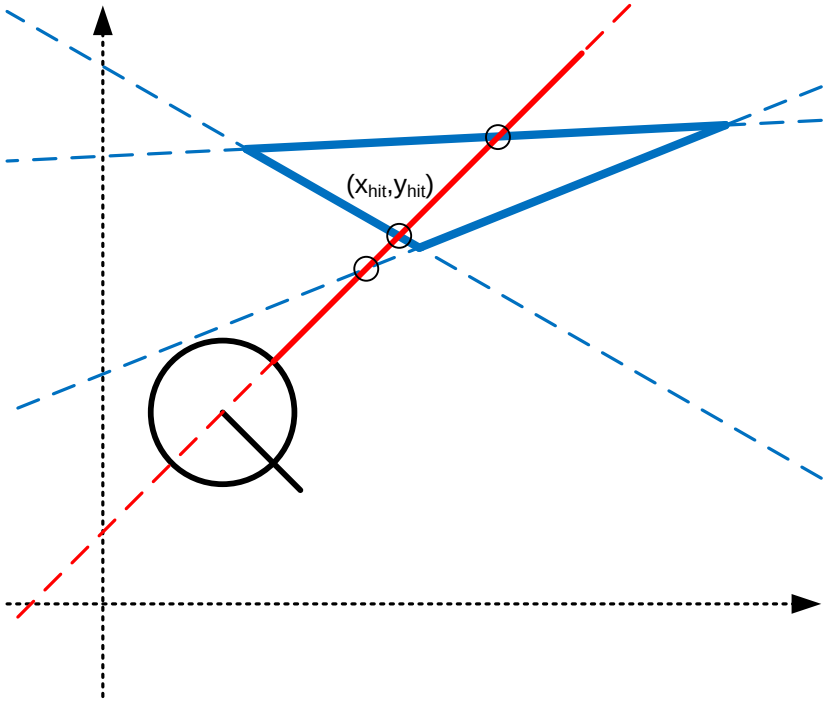
Lines that are vertical or horizontal don't always return infinite or zero slopes, so they must be accounted for.

$$m = \begin{cases} -\infty\,, & m < -10^{14} \\ 0\,, & |m| < 10^{-14} \\ \infty\,, & m > 10^{14} \\ m\,, & otherwise \end{cases}$$

The intersections between the infinitely long sensor line and the wall lines are found simply by solving the equations algebraically.

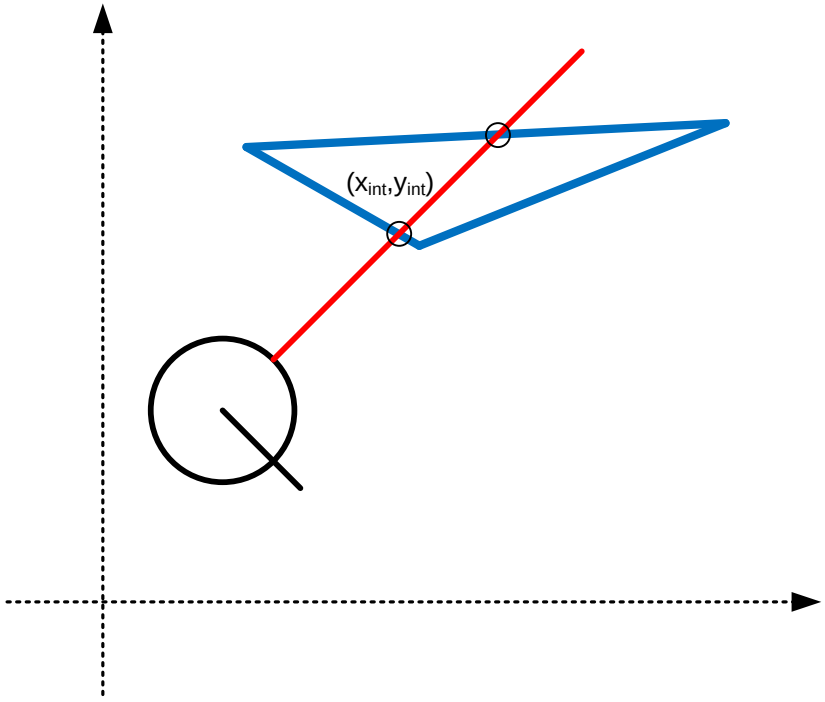
$$b_{sensor} = y_{sensor} - x_{sensor} * m_{sensor}$$

$$b_{wall} = y_1 - x_1 * m_{wall}$$

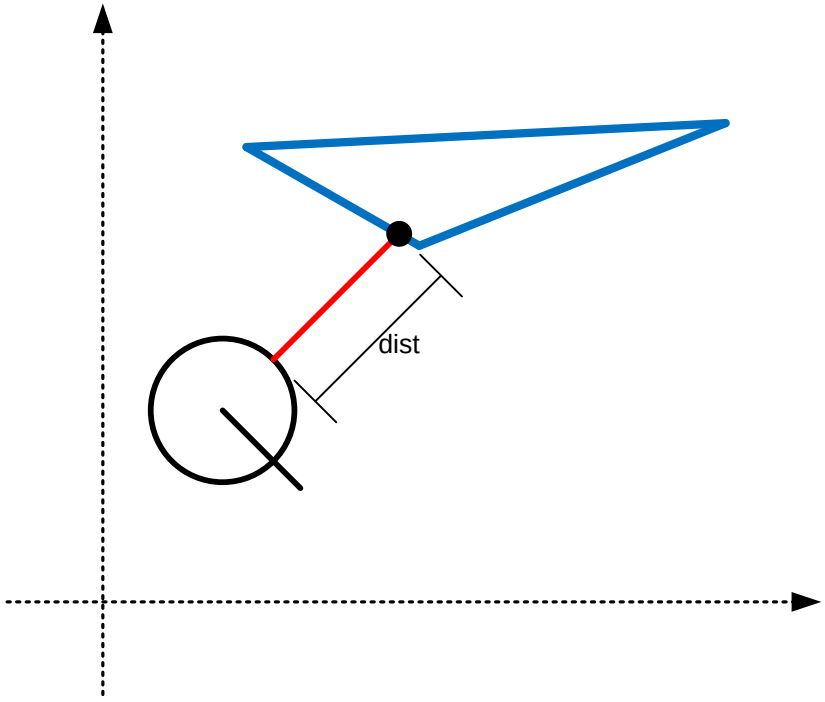


$$\begin{cases} \text{no intersection}, & m_{sensor} = m_{wall} \\ \begin{matrix} x_{hit} = x_{sensor} \\ y_{hit} = m_{wall} * x_{hit} + b_{wall} \end{matrix}, & m_{sensor} = \infty \\ \begin{matrix} x_{hit} = x_1 \\ y_{hit} = m_{sensor} * x_{hit} + b_{sensor} \end{matrix}, & m_{wall} = \infty \\ \begin{matrix} x_{hit} = \frac{b_{wall} - b_{sensor}}{m_{sensor} - m_{wall}} \\ y_{hit} = m_{sensor} * x_{hit} + b_{sensor} \end{matrix}, & otherwise \end{cases}$$

The function then determines if the intersection point is on the finite wall line and sensor line by checking that the point is within the x and y limits of the lines. Note that all limits must be checked to account for vertical and horizontal lines, and they must be checked with tolerances since MATLAB produces rounding errors that could result in the intersection being slightly off of a vertical or horizontal line.

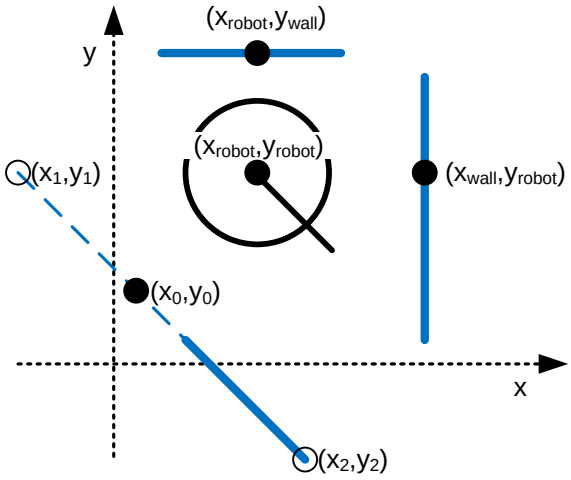


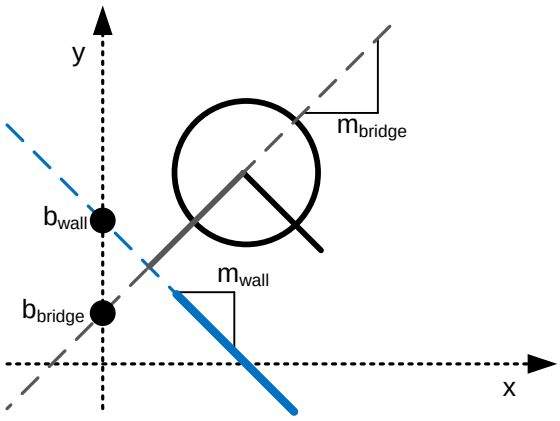
Finally, the function calculates the distance to all of the remaining intersection points and chooses the lowest distance to output.



`findCollisions:`

The method of finding the collision points is to first find the closest points on all the walls to the center of the robot. Each wall is inspected individually, first determining the line equation (in the form y = mx + b). Note that if the wall is vertical the closest point will be (x_{wall},y_{robot}). If it is horizontal, the closest point will be (x_{robot},y_{wall}). Otherwise, the closest point (x_0,y_0) must be calculated by using the line equations.





Equation for wall line:

$$m_{wall} = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b_{wall} = y_1 - m_{wall} * x_1$$

Equation for line from closest point on wall to robot center (bridge):

$$m_{bridge} = -\frac{1}{m_{wall}}$$

$$b_{bridge} = y_{robot} - m_{bridge} * x_{robot}$$

Now the intersection of the two lines can be found using the same method as in `findDist`.

$$x_0 = \frac{b_{bridge} - b_{wall}}{m_{wall} - m_{bridge}}$$
$$y_0 = m_{bridge} * x_0 + b_{bridge}$$

The intersection point is then checked to ensure that it is on the finite-length wall line by making sure the point is within both the x and the y limits on the wall. The point does not need to be checked against the bridge line, since that line is essentially infinitely long. If the point is not on the finite-length wall line, then the closest endpoint of the wall must be the actual closest point. The endpoint flag is also set, and the reason for this is explained on page 45.

If the closest point, whether it is an intersection between the lines or the corner of the wall, is within the radius of the robot to the center, then it is recorded as a collision point. It may be beneficial to reduce the required radius by a bit to ensure that the robot is truly colliding and not just brushing by the wall, but this is probably a negligible difference. Note again that the simulator makes no distinction between points that are completely inside the robot and points that are on the edge. It is assumed that the timer updates frequently enough that a wall will be caught before it gets too far inside the robot.

Finally, only the two closest collision points are kept to be passed as output. The 'closest' qualification assumes that the closest points will affect the robot the most. It may create a simulation closer to reality if another qualification is chosen, such as the two most angularly separated points, or the points closest to the heading of the robot, but those claims will need to be tested. The reason only two points are retained is the simplifications done by the physics engine (explained on page 6).

State Manipulator Functions (Physics Engine)

All physics engine functions are called by `updateSim` (which is called by the timer interrupt). One of the input arguments from `updateSim` is `tStep`, which is the time since the last update (Δt). Under the normal simulation speed option, `tStep` will be the actual time since the last update. However, when either the `2x` or `3x` radio buttons are chosen for the Sim Speed option, `tStep` will be multiplied by these options before being passed to the physics engine functions. More about the Sim Speed option is on page 46.

`driveNormal:`

This function determines the position and orientation of the robot in the new time step based on the position, orientation, and intended velocities from the previous time step. This function is called when the robot is not interacting with any walls, so the dynamics are much easier to resolve.

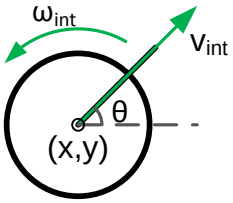
The function first checks the case of straight driving, since this is easiest to calculate. If the intended angular velocity is zero, then the robot will drive in a straight line. The change in the distance is simply:

$$\Delta dist = v_{int} * \Delta t$$

The next case checked is pure turning, where the intended forward velocity is zero. In this case, the robot will simply rotate based on the equation:

$$\Delta\theta = \omega_{int} * \Delta t$$

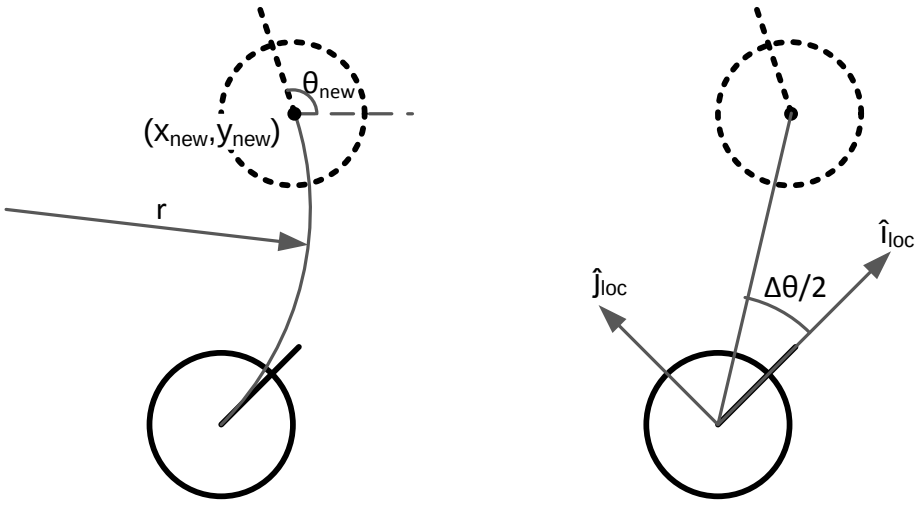
If the robot has non-zero angular and non-zero linear velocity, then the computation becomes more complicated. The information known at the start is the position, orientation, intended angular and linear velocities, and the length of the time step.



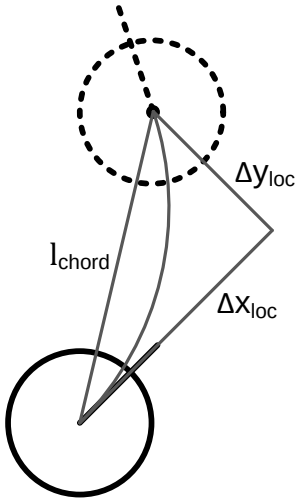
Since the angular and linear velocities are considered to be constant, the robot path will be an arc with a constant radius. The new orientation can be easily calculated in the same way in the pure turning case. The turning radius can be calculated simply by relating linear and angular velocity to arc length (and thus to radius).

$$\Delta\theta = \omega_{int} * \Delta t$$
$$r = \frac{v}{\omega}$$

To figure out the change in the position of the robot, some simple trigonometry will be used. This will be computed off of the right triangle formed by the chord (similar to secant line) from the old position to the new, and the change of position in local coordinates. The local x-axis will be in the direction the robot is pointing. The angle inside the triangle closest to the old position will be half of the angle turned.

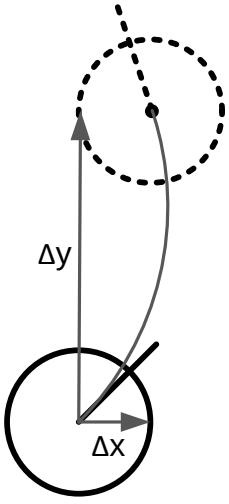


Next, the length of the chord line must be computed, and then converted to displacements in the local x and y directions. Note that in the code, the direction of the intended linear velocity is used to account for the robot backing up and turning.



$$l_{chord} = 2rsin\left(\frac{\Delta\theta}{2}\right)$$
$$\Delta x_{loc} = l_{chord}\cos\left(\frac{\Delta\theta}{2}\right)$$
$$\Delta y_{loc} = l_{chord}\sin\left(\frac{\Delta\theta}{2}\right)$$

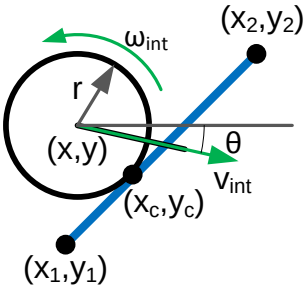
Finally, the displacements in local coordinates can be converted to absolute coordinates and added to the old position to find the new one. This is done using a simple matrix rotation (usually used to transform vectors).



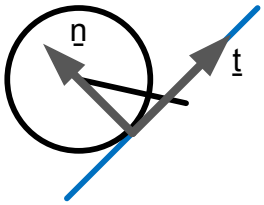
$$\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix} = \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}\begin{bmatrix}\Delta x_{loc}\\ \Delta y_{loc}\end{bmatrix}$$
$$x_{new} = x + \Delta x$$
$$y_{new} = y + \Delta y$$

`drive1Wall:`

One of the problems with the walls in the physics engine is the inability to know how exactly it relates to the robot. To resolve this, several conditional blocks are used to set values. The wall endpoints are first chosen with the convention that the left-most point will be (x_1,y_1), and the right-most will be (x_2,y_2).



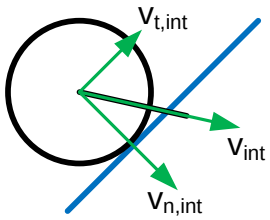
The unit vectors that determine direction of movement are calculated next. The convention for $point_1$ and $point_2$ of the wall is necessary to help determine the correct direction of the tangential vector, relative to the robot. The simulator convention is to have the tangential vector point counter-clockwise around the robot. This is to help determine the relationship between the effect on actual velocity by the linear and angular intended velocities. The normal vector is calculated from the collision point to the center of the robot. Theoretically, the tangential and normal vectors should be perpendicular, but this is not guaranteed with this type of method. An alternative might be to calculate the tangential vector from the cross product of the normal and the z-axis. This would avoid some of the messy conditionals. Note the direction of the normal vector, so if the robot moves towards the wall, it is in the direction opposite the normal.



$$\hat{t} = \begin{cases} \frac{(x_2-x_1)\hat{\imath}+(y_2-y_1)\hat{\jmath}}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}}, & \textit{contact point is below robot} \\ \frac{(x_1-x_2)\hat{\imath}+(y_1-y_2)\hat{\jmath}}{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}}, & \textit{contact point is above robot} \end{cases}$$

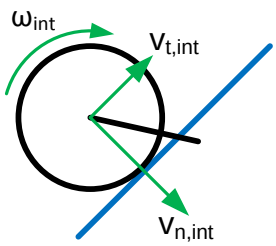
$$\hat{n} = \frac{(x-x_c)\hat{\imath}+(y-y_c)\hat{\jmath}}{\sqrt{(x-x_c)^2+(y-y_c)^2}}$$

Once the normal and tangential vectors are determined, it's simple enough to find the normal and tangential components of intended velocity. Also, the normal component of the actual velocity is simple to calculate since friction does not factor in at all. If the intended normal velocity is negative (towards the wall), obviously the actual normal velocity will be zero since the robot cannot penetrate the wall. If the intended normal velocity is positive, then it will be equal to the actual normal velocity since no wall will affect the robot.



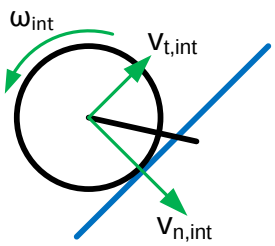
$$v_{t,int} = v_{int} \cdot \hat{t}$$
$$v_{n,int} = v_{int} \cdot \hat{n}$$
$$v_n = \begin{cases} 0, & v_{n,int} \leq 0 \\ v_{n,int}, & v_{n,int} > 0 \end{cases}$$

Now the robot can be thought of as a wheel against the ground. If the wheel is turning such that the outer edge is moving faster than the intended tangential velocity, and in the opposite direction (i.e. the robot spins clockwise while the tangential velocity is positive, and vice-versa), then the rotation will assist in moving the robot along. If the robot spins too slowly, or in the same direction, then the robot edge will drag along the wall and slow down the tangential velocity. In the first case, the tangential movement of the robot will speed up the rotation of the robot. In the second case, it will slow it down. These are all very rough approximations of dynamics, and much more accurate formulas could be implemented. Note that if the intended normal velocity is away from the wall, the friction will not impact the robot at all. Also note that the assumption about friction calculations on page 7 becomes relevant here.



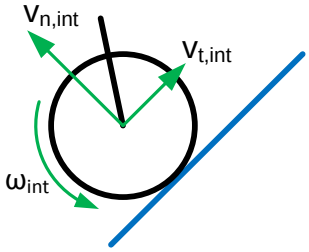
First Case

$$v_t = v_{t,int} + v_{n,int} * \mu_s$$
$$\omega = \omega_{int} + \frac{v_{n,int} * \mu_s}{r}$$



Second Case

$$v_t = v_{t,int} - v_{n,int} * \mu_s$$
$$\omega = \omega_{int} - \frac{v_{n,int} * \mu_s}{r}$$



Third Case

$$v_t = v_{t,int}$$
$$\omega = \omega_{int}$$

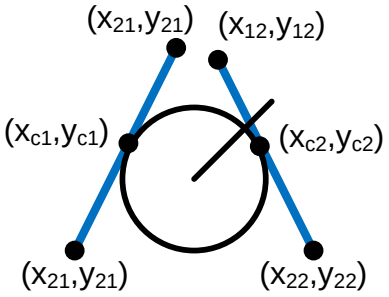
Friction will only slow down movement; it does not have the ability to reverse its direction. Therefore, the actual values for tangential and angular velocities are checked against their intended values. If the signs are opposite, the actual values are reduced to zero. This is to prevent an extreme amount of normal force from influencing the frictional force into overriding a small tangential velocity and forcing the robot in the other direction, or similar situations.

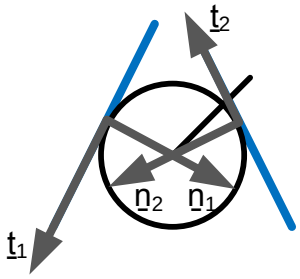
Finally, the components of the tangential and normal velocities in the x and y directions are extracted and used to compute the new position with a simple kinematic formula. The new orientation is calculated similarly from angular velocity.

$$x_{new} = x + v_x * \Delta t$$
$$y_{new} = y + v_y * \Delta t$$
$$\theta_{new} = \theta + \omega * \Delta t$$

`drive2Wall:`

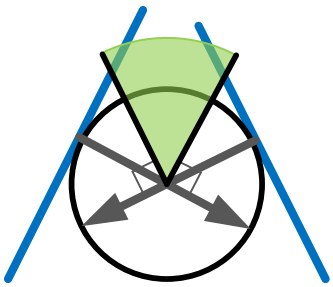
This function starts similarly to `drive1Wall`. It determines the tangential and normal vectors of both walls such that the tangential vectors point counter-clockwise around the robot, and the normal vectors go from the collision points to the middle of the robot. The normal component of the intended velocity is then found for both walls.



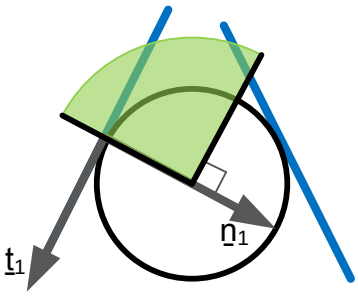


At this point, there are three possible types of movement. If the robot is touching both walls and heading into the corner, it will not move. If the robot is heading away from both walls, it can drive normally with no friction. If the robot is heading away from one wall, but towards the other, it will drive the same as if the only wall there was the one it's heading towards. The difficulty comes in determining which situation is in effect.

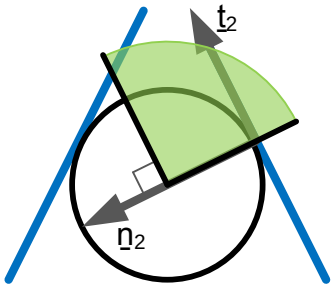
The methods for determining which situation is in effect differ depending on the angle of the walls. Acute angles between walls will be explored first. The robot will be stuck in the corner if the velocity points between the normal vectors on the wall. There are several methods of determining this, but some are simpler than others. The easiest is if the intended velocity has a component in the opposite direction of both normal vectors. However, this leaves out two areas in which the velocity could be pointing, and still be driving into the corner. This area is trickier to compute, since the tangential vector is required, but it is not known which tangential vector points towards the corner, and which points away. The following conditions will cover all cases. Note that the first case is used, even though it is covered by both of the other two cases since it is much faster computationally. If this case is found to be true, then the other two cases do not have to be checked. In any of these situations, the velocities will be zeroed and the robot will not move anywhere.



$$\vec{v}_{int} \cdot \hat{n}_1 \leq 0$$
AND
$$\vec{v}_{int} \cdot \hat{n}_2 \leq 0$$

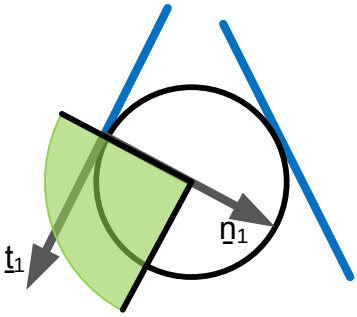


$$\vec{v}_{int} \cdot \hat{n}_1 \leq 0$$
AND
$$(\vec{v}_{int} \cdot \hat{t}_1 \leq 0$$
XOR
$$\hat{t}_1 \cdot (\hat{n}_1 + \hat{n}_2) < 0)$$

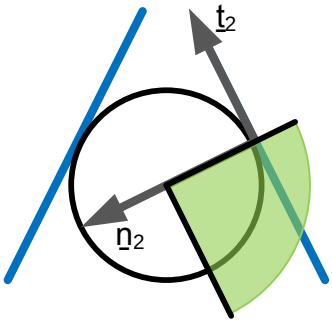


$$\vec{v}_{int} \cdot \hat{n}_2 \leq 0$$
AND
$$(\vec{v}_{int} \cdot \hat{t}_2 \leq 0$$
XOR
$$\hat{t}_2 \cdot (\hat{n}_1 + \hat{n}_2) < 0)$$

The next possibility is that the robot will drive away from the corner, but still towards one of the walls. This will occur if the intended velocity points opposite of one of the normal vectors. Note that this does also include a case where it is driving towards the corner, but that case has already been checked so it will be handled correctly. If the intended velocity is pointing negative to the first normal vector, `drive1Wall` will be called, with the only wall used now the first wall (and similarly for the second wall).

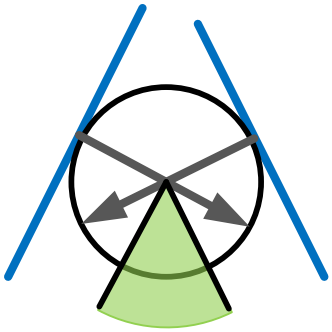


$\vec{v}_{int} \cdot \hat{n}_1 \leq 0$



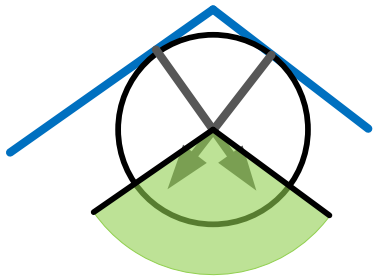
$\vec{v}_{int} \cdot \hat{n}_2 \leq 0$

The final situation occurs in the only area not yet covered, where the intended velocity has a component in the positive direction of both normal vectors. In this situation, the robot will drive without any interference from friction. The function `driveNormal` could be called, but this is probably an unnecessary amount of computation since this situation will only occur for one or two time steps (when the robot moves far enough to no longer contact the wall). Instead, the robot will move based only on linear velocity, and turn based on angular velocity. There will be no path-arc computation.



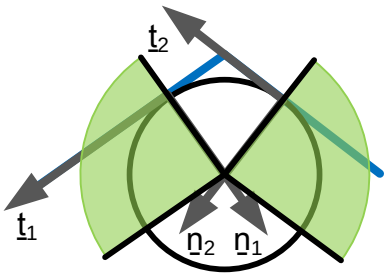
$\vec{v}_{int} \cdot \hat{n}_1 > 0 \ \ AND \ \ \vec{v}_{int} \cdot \hat{n}_2 > 0$

Next, the arrangement where the walls are at obtuse angles with each other must be examined. The first situation is where the robot can drive freely away from the corner. This where occur when the intended velocity has a component in the direction of both normal vectors.

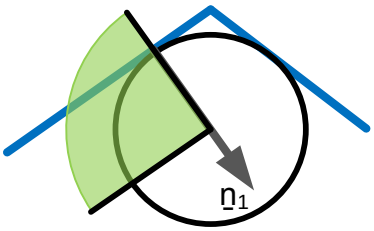


$$\vec{v}_{int} \cdot \hat{n}_1 > 0 \;\; AND \;\; \vec{v}_{int} \cdot \hat{n}_2 > 0$$

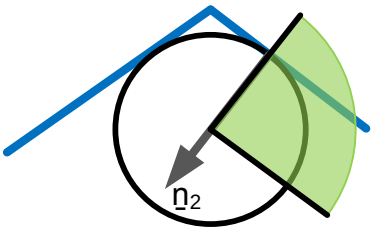
If the normal movement condition is not met, the condition for driving along one of the walls is checked. If the intended velocity has components in the direction of both tangent vectors, or in the negative direction of both tangent vectors, then the robot is driving towards a wall. Since it is not specified which wall is which (1 or 2 on the left or right), these conditions alone cannot tell us which wall should be considered. If the intended velocity has a larger component in the opposite direction of a normal vector than in the opposite direction of the other normal vector, then it is driving towards that wall.



$$sgn(\vec{v}_{int} \cdot \hat{t}_1) = sgn(\vec{v}_{int} \cdot \hat{t}_2)$$

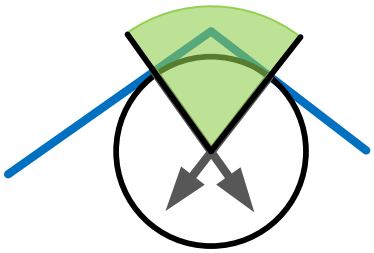


$$\vec{v}_{int} \cdot \hat{n}_1 < \vec{v}_{int} \cdot \hat{n}_2$$



$$\vec{v}_{int} \cdot \hat{n}_1 > \vec{v}_{int} \cdot \hat{n}_2$$

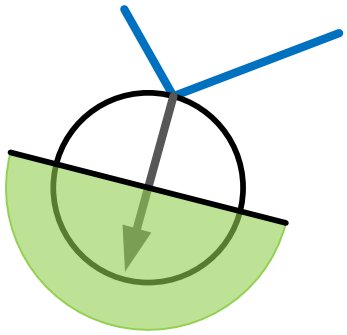
Finally, the only situation left is the robot being stuck in the corner. This will happen most of the time when the intended velocity has a component in the opposite direction of both normal vectors, but this region will also overlap slightly with the above areas for driving into the wall. Thus, this situation is left for the `else` statement.



`driveCorner:`

The remaining physics engine function operates when the robot is contacting the corner of one or two walls. There are only two situations considered here, the robot moving towards the corner, and the robot moving away. In reality, much more complex algorithms could be used to determine if the robot is stuck on the corner, sliding along it, bouncing off of it, or completely stalled. However, this simplified version of a physics engine seems sufficient to give an idea of what a real robot would do in this situation.

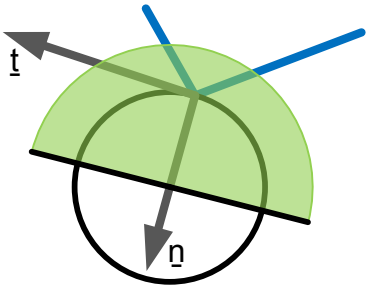
The normal vector is set to be the line from the corner to the middle of the robot. If the intended velocity of the robot has a component in the direction of the normal vector, then it is driving away from the corner, and thus will proceed unhindered. The function `driveNormal` is called in this situation.



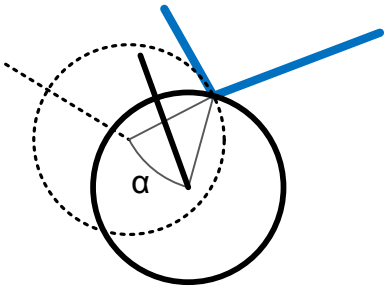
$\vec{v}_{int} \cdot \hat{n} \geq 0$

The other situation is when the intended velocity is in the other direction, towards the corner. In this case, the robot will execute a rolling motion around the corner. The center of the robot will move in a circular path about the corner, while the heading of the robot will rotate oppositely, as if glancing off the corner. The heading will also be affected by the angular velocity of the robot mitigated by friction from the corner. This combination of behavior types is probably not the most realistically accurate, and would probably be better off replaced with different types of behavior based on incident angles of intended velocity on the corner.

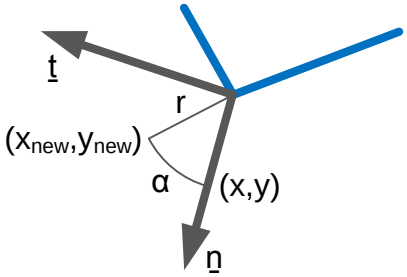
Determining which direction the robot will roll around the corner on is done by calculating the 'tangent' vector from the corner (cross product of the normal and z-axis vectors), and comparing the intended velocity to that. The tangent vector will always point counter-clockwise around the robot, so if the intended velocity is also in that direction then the robot center will rotate clockwise about the corner. The opposite situation produces the opposite results.



$$\vec{v}_{int} \cdot \hat{n} < 0$$
$$\hat{t} = \hat{n} \times \hat{k}$$



$$\alpha = -\frac{\vec{v}_{int} \cdot \hat{t}}{r} * t_{step}$$



$$x_{new} = x + \overrightarrow{\Delta n} \cdot \hat{\imath} + \overrightarrow{\Delta t} \cdot \hat{\imath}$$
$$y_{new} = y + \overrightarrow{\Delta n} \cdot \hat{\jmath} + \overrightarrow{\Delta t} \cdot \hat{\jmath}$$
$$\theta_{new} = \theta - \alpha + (\omega_{int} + (\vec{v}_{int} \cdot \hat{n}) * \mu_k) * t_{step}$$
$$\overrightarrow{\Delta n} = r * (\cos(\alpha) - 1) * \hat{n}$$
$$\overrightarrow{\Delta t} = -r * \sin(\alpha) * \hat{t}$$

Translator Functions

Each of the functions falling into this category are used as the 'middle function' between the autonomous control program and the simulator's raw algorithms and data. Each translator function follows a pattern that serves different purposes in making the simulator work or helping the user test their program.

1. Check for valid input. If the robot object is not passed properly, and any other input parameters are not within acceptable values, an error with the message ID `Simulator:invalidInput` will be thrown. Note that this will not check for input out of acceptable ranges that are checked for later (e.g. `SetFwdVelAngVelCreate`).

2. Call the `autoCheck` function. This will throw an exception (error) with the message ID `Simulator:autonomousDisabled` if autonomous mode is not in effect. This is done so that the `Stop` button in `SimulatorGUI` will work. See page 20 for more information about this process. Note that this means none of the translator functions will work from the command window without first changing the `autoEnable` property. The translator functions should only be used by loading an autonomous program into `SimulatorGUI` anyway.

3. Pause for communication delay. The length of the pause is determined by the property `comDelay`, which is set by the configuration file. The pause is in the same location (before the main code execution) on both sensor and command type functions. A more accurate representation may be to only pause for half of the delay on command functions where no values are returned, but this seems unnecessary.

4. Execute the main body of code. This is where an appropriate sensor function is called, or certain values are set depending on the function specification. All output to the command window from the equivalent function in the real Create toolbox is duplicated here.

5. Create string representation of function call and add to output. Note that arguments will be omitted if the function has none, and the robot object argument is omitted in all functions. Note also that only five of the data points in a call to `LidarSensorCreate` are added to this string, since all of them would be far too many to handle, and only one beacon information is displayed for CameraSensorCreate. This can be adjusted in the function. The string representation is made in a similar format to the following:
   ```
   [argOut1 argOut2]= FcnName(argIn1,argIn2)
   ```

6. Catch all unexpected errors. The `catch` statement at the end of each translator function is to help in debugging. The default MATLAB error messages are not always helpful in determining the problem. If the issue has not already been checked (e.g. invalid input) then this is intended to give a more informative error message. The original exception will be allowed to propagate through though to allow debugging from the original message. Note that the error used to quit autonomous mode will not cause a message to be displayed.

updateSim

The first objective of `updateSim` is to move the robot to its new position. One of the four physics engine functions will be called depending on the output from `findCollisions`. If no walls are being hit, then `driveNormal` is called. If one wall is hit, but not on its corner, or if two walls are hit, but only one is hit on the corner, then `drive1Wall` is called. In the two-wall case, it will be called with the one wall that is contacting the robot not on its corner. If two walls are contacted, both not on their corners, then `drive2Wall` is called. Finally, if only one wall is hit on the corner or two walls both on corners, then `driveCorner` is called. Only the corner closest to the robot is used in the two-wall case.

Each of the physics engine functioned mentioned above require a time step argument that is the time since the last update. This will allow them to calculate the new robot position based on the very basic rule, $x_{new} = x_{old} + v*t_{step}$. `SimulatorGUI` gives the option to the user to "speed up" the simulation to 2 or 3 times faster. This does not actually speed up any computation, update regularity; it just acts as a multiplier to `tStep` so that the robot will move 2 or 3 times farther when it is traveling at the same velocity, which causes some issues. If the user's autonomous control program utilizes `tic-toc` functions or anything that measures real-time, the program's timing relative to the robot's movement will be changed. Also, because `updateSim` is not called any more frequently than at normal speed, the robot will sometimes move significantly further into the wall before the physics functions catch it. There is also evidence of other effects, the exact details and causes of which are unknown. The Sim Speed option may help users when roughly testing a long-running control program, but for quality runs the simulation should be left at normal speed.

The `updateSim` function next creates a new row in the data output cell array and adds the position and orientation data by calling `updateOutput`. Then, the old position and the new position are passed to `updateOdom`, to update the odometry data stored in the robot object. If the camera focus is set to Robot Centric, the focus of the camera is shifted if necessary to center the robot.

The final purpose of `updateSim` is to change the plot to reflect the current positions of the robot and sensor visualizations. The robot is represented by a circle (made of 20 lines) and a line indicating the heading. The sensor visualization is updated using the aptly named `CreateRobot` method `updateSensorVisualization`.

ReplayGUI

Important UserData

`figure_simulator` – Array of three handles for the plot of the robot and path

1. Circle
2. Direction line
3. Robot path line

`text_IOData` – Double of current index of the playback
`check_path` – Double of the starting index of the plot of the robot path
`edit_display` – Cell array of the output data from `SimulatorGUI`
`push_slow_back` – Boolean of whether the playback should be rewinding slowly
`push_slow_forward` – Boolean of whether the playback should be moving slowly forwards
`push_play` – Boolean of whether the playback should be moving at normal speed forwards

Data Format

The format of the output data from `SimulatorGUI` is a cell array ? x 5. The '?' is variable, since it depends on how long the autonomous code executes for. The first four entries in a row are all doubles: timestamp, x-position, y-position, orientation. The final entry in a row will be in one of three formats: empty vector, string, cell array of strings. The empty vector will indicate that there were no translator function calls during that time-step. The string will be representative of a single function call during that time-step. The cell array will occur when multiple function calls are made during that time-step. Each cell in the array contains the string of the function call.

Visualization

The map parsing and plotting algorithm is the same as used in `MapMakerGUI`. Note that the map will not actually affect the robot movement at all; it is there merely for visualization purposes. If the wrong map is loaded, there will be no warning or error, but the movement of the robot and the data output will probably not match up.

The edit text box is used for data display rather than a static text box to allow multi-line viewing with a scrollbar. Setting the property `Enable` to `inactive` will make it so the user cannot edit the text in the box, even though it will show up in normal colors. The output data is printed to the display by appending the next line onto the end of the `String` property when moving forward through playback, and deleting the end line when stepping backwards.

There are some issues with the display, since it is difficult to fit everything on one line. The text wraps around to the next line automatically, but sometimes will do it in odd places so that the function calls are difficult to read. However, if the newline character `\n` is inserted into the display, the edit box will count that is placing in a new line. Thus, when the rewind functions are used, the function call strings will count as individual lines, so the display will not match up with the current index. Another problem is that the scrollbar will move to the top every time the string inside the edit box is updated, but the new lines are down at the bottom. This could be solved by putting all the new output data at the top of the box, but that is not intuitive to read. Another option is to get deep into the Java code behind the MATLAB uicontrols, but that gets fairly complicated.

Contact Information

Suggestions and bugs can be reported in the appropriate forum on the project's SourceForge page:
Forums: https://sourceforge.net/projects/createsim/forums/forum/1204154

Other inquiries can be directed to:
Email: CreateMatlabSim@gmail.com

For more information, visit:
Website: http://web.mae.cornell.edu/hadaskg/CreateMATLABsimulator/createsimulator.html
SourceForge: https://sourceforge.net/projects/createsim

Contributing Parties

Author:
Cameron Salzberger

Advising Professors:
Dr. Hadas Kress-Gazit
Dr. K-Y Daisy Fan

Coding and Testing Assistance:
Jason Hardy
Francis Havlak
Ankit Arora

Creators of the MATLAB Toolbox for the iRobot Create:
Joel M. Esposito
Owen Barton
http://www.usna.edu/Users/weapsys/esposito/roomba.matlab/
2008

Code Resources:
Nassim Khaled (inside_triangle.m on MATLAB Central)
Zhenhai Wang (circle.m on MATLAB Central)

Sponsor:

The MathWorks
http://www.mathworks.com/

Index