MATLAB Simulator for the iRobot Create

User Manual

Author: Cameron Salzberger
Advisors: Dr. Hadas Kress-Gazit
Dr. K-Y Daisy Fan
Supported By: The MathWorks
Updated: 2/1/2011

CORNELL UNIVERSITY

Contents

Disclaimer

Glossary of Terms

Beacon – Small colored balls used for blob detection by the camera.
GUI – Graphical User Interface. A more intuitive method of interacting with the program than a text-based interface.
IR – InfraRed. IR sensors detect electro-magnetic signals in that wavelength range.
LIDAR – LIght Detection And Ranging. Measures the distance to objects within a certain range using the time delay between emission and detection of a signal.
MATLAB – MATrix LABoratory. A programming environment developed by the MathWorks, which is often used for numerical computing.
Toolbox – Set of files used in MATLAB for a specific purpose.

The iRobot Create

The Create is a robotic platform built along similar lines as the Roomba vacuum cleaner (but without the vacuum). It is made to be used for educational and research purposes, and is easily programmable and modifiable for that purpose. It contains several built-in sensors, and two input/output serial ports that allow for more to be added.

The owner's guide for the Create can be found here:
http://www.irobot.com/filelibrary/create/Create%20Manual_Final.pdf
More details on the sensors are in this guide on page 7.

The Create is controllable via the Open Interface commands through the serial port connector. Since these commands are not very intuitive, the MATLAB Toolbox for the iRobot Create was developed to send and receive the serial commands, allowing a more abstract level of control by the user. This toolbox is not necessary for the simulator to work. The files and further information can be found at:
http://www.usna.edu/Users/weapsys/esposito/roomba.matlab/

Purpose of the Simulator

The intention behind this simulator is to allow the user to test code that would normally be run on the real Create. The user writes a MATLAB function that can run using the MATLAB Toolbox for the iRobot Create to control the robot. This function can be called into the simulator, which will then display a representation of the robot exhibiting similar behavior as the normal Create would. This is much more convenient for educational institutions that have limited access to Create robots and/or testing time for code. This is also useful as a teaching tool without using a physical Create at all. The user-created control program can be used to test mapping, obstacle detection and avoidance, searching, and other algorithms. This project is open-source to allow users to customize the simulator to their needs.

Requirements

The simulator was developed and tested on MATLAB R2008b (Version 7.7) and R2010a (Version 7.10). It should work on R2008a (Version 7.6) and all newer versions to date. However, versions before R2008a used a different syntax for object oriented programming, and most likely will not run the simulator. Also, different operating systems have different settings for application windows and fonts, so the appearance may be somewhat different than pictured in this manual. The simulator has been tested on Windows Vista and Windows 7.

The simulator and this document assume a basic knowledge of MATLAB, including how to write and run functions, working with nested functions, and using different variable classes. If you are new at using MATLAB, the MathWorks tutorial page can be found at:
http://www.mathworks.com/academia/student_center/tutorials/launchpad.html

This document will assume that you have read the documentation for the MATLAB Toolbox for the iRobot Create, which can be found at:
http://www.usna.edu/Users/weapsys/esposito/roomba.matlab/Matlab_Toolbox_iRobot_create_doc.pdf
Included with the simulator is a similar document that focuses only on the available functions and their specifications. If you do not plan to program the real Create, it is recommended that at least the Function Specification document is used for reference.

This document is associated with simulator version 1.0 (iRobotCreateSimulatorToolbox1_0).

Installation

New Installation

1. Download iRobotCreateSimulatorToolbox1_0.zip from the SourceForge page.
   https://sourceforge.net/projects/createsim
2. Unzip the toolbox to extract the toolbox folder containing all the functions.
3. Place that folder in a permanent location on your computer's hard disk drive.
   e.g. C:\Program Files\MATLAB\R200Xx\toolbox
4. Start MATLAB.
5. In the menu toolbar, click File, then select Set Path.
6. Click Add Folder with Subfolders, navigate to the toolbox folder you just saved, select it, then press OK.
7. Click the Save button, then press Close.

If you move the toolbox folder, just delete the old path that was set and follow steps 5 through 8 again to set the new path. If the path fails to save because MATLAB does not have administrator access to the file `pathdef.m`, quit MATLAB and reopen it by right-clicking and choosing Run as Administrator. Then perform steps 5 through 7 again. If you are not system administrator on your computer, that person will need to perform these steps.

Updating an Old Installation

1. Navigate to the folder containing the files from the previous version of the toolbox.
2. Delete all of the files, but do not delete the folder.
3. Download iRobotCreateSimulatorToolbox1_0.zip from the SourceForge page.
   https://sourceforge.net/projects/createsim
4. Unzip the toolbox to extract the files.
5. Copy and paste these files into the folder that used to contain the previous version of the toolbox.

If done correctly, there will be no need to set the search path again. If the functions are not accessible from all directories, follow steps 5-7 in the New Installation section to reset the path.

Make sure you do not keep files from multiple versions of the toolbox on the MATLAB search path. This may cause older versions to be called by newer functions, resulting in errors. This is best avoided by completely deleting older versions of the toolbox.

Writing the Autonomous Control Program

Requirements

The control program must be a function with a single input argument. If the control program calls `RoombaInit` within it, then this argument must be the ComPort number that is the input parameter to `RoombaInit`. If you would prefer to call `RoombaInit` prior to running the control program on the real robot, then the input argument to the control program must be the SerialPort object returned by `RoombaInit`. Calling `RoombaInit` prior to running the control program is preferable when operating the real Create. Otherwise, serial port communication must be constantly opened and closed at every run of the autonomous control program. Remember that the SerialPort object must be the first argument in calls to functions in the MATLAB Toolbox for the iRobot Create.

Examples below show acceptable formats for calling the control program to run the real Create:

Command Window

```
>> comPort= 8;
>> serPort= RoombaInit(comPort);
>> controlProgram(serPort)
```

Control Program

```
function controlProgram(serPort)
% Autonomous control function

% Example function from toolbox
BeepRoomba(serPort)

...

end
```

-OR-

Command Window

```
>> comPort= 8;
>> controlProgram(comPort)
```

Control Program

```
function controlProgram(comPort)
% Autonomous control function

serPort= RoombaInit(comPort);

% Example function from toolbox
BeepRoomba(serPort)
...
end
```

To see particular function descriptions and input/output specifications, type into the command window "`help CreateRobot/functionName`". For example:

```
>> help CreateRobot/SetFwdVelAngVel
```

Recommendations

Most control programs will have the majority of their functions contained in a loop. Inside will be a number of conditionals that will allow the robot to react to particular events. For example, the program will call `BumpsWheelDropsSensorsRoomba`. If a bump sensor is activated, the robot will back up and turn, before moving forward again. This very simplified; the real control program could specify more complex behavior based on multiple inputs. For one way to structure such a program, see the file `ExampleControlProgram.m`.

Control programs should probably specify a time limit on the loop as well. The simulator will record the autonomous code execution output data regardless of whether the Save Data checkbox is marked or not, so the memory taken up by this can become quite large. The maximum size depends heavily on the operating system, version of MATLAB, and other workspace variables, but it is good in general to keep the array size down. This could easily be implemented with the following code fragment for the main loop.

```
tic
while toc < 1200
...
```

Create Sensors and Sensor Functions

Used in the Simulator

Bump

Summary: The bumper on the front of the robot is depressed when a solid object is hit. There are two bump sensors (one on the left, one on the right). When the front of the robot is hit, both left and right sensors are depressed, so the MATLAB toolbox regards this as a front sensor press and not a left or right.

Called by: `AllSensorsReadRoomba`
`BumpsWheelDropsSensorsRoomba`

Buttons

Summary: The Create can determine if the Play (**>**) or Advance (**>>|**) buttons are pressed or not. The simulator represents these buttons in the upper left corner of the simulator window. During both manual and autonomous control, these buttons may be pushed by the user. Currently only one button may be pushed at a time. Note that on the real Create, `ButtonSensorRoomba` will only return true if the buttons are currently being held down. However, the buttons in the simulator are toggle buttons. They must be pushed once to activate, and again to deactivate.

Called by: `AllSensorsReadRoomba`
`ButtonSensorRoomba`

Cliff

Summary: On the underside of the bumper are four infrared cliff sensors. Each one reads the reflectivity of the surface under the sensor. A surface with a light color has high reflectivity, dark colors have low reflectivity, and no surface (a cliff) has zero reflectivity. In the simulator, the Lines contained in the map files are considered black for the cliff sensors. Therefore the signal strength of the cliff sensor will be low, but no cliff will be detected.

Called by: `AllSensorsReadRoomba`
`CliffFrontLeftSensorRoomba`
`CliffFrontLeftSignalStrengthRoomba`
`CliffFrontRightSensorRoomba`
`CliffFrontRightSignalStrengthRoomba`
`CliffLeftSensorRoomba`
`CliffLeftSignalStrengthRoomba`
`CliffRightSensorRoomba`
`CliffRightSignalStrengthRoomba`

IR Receiver

Summary: The front of the Create has an omni-directional infrared receiver and transmitter for reading signals from various Create accessories and other Creates. The only use in the simulator is to read the signal from virtual walls. The virtual walls produce a field around the emitter, and in the direction it faces. If the Create is within this field, it should "see" the virtual wall. See the Roomba user manual page 12 for more information on virtual walls:
http://www.irobot.com/images/consumer/cs/097.06-Roomba-Gnrc-Manual.pdf

Called by: `AllSensorsReadRoomba`
`VirtualWallSensorCreate`

Odometry

Summary: The wheels on the Create have encoders that keep track of how many times they spin. Knowing the radius of the wheel and the width of the robot allows the Create to keep track of how far it has traveled and how much it has turned. However, this does not account for wheels slipping, which makes the odometry inaccurate over large distances. It is not recommended to navigate using only 'dead reckoning' from odometry.

Called by: `AllSensorsReadRoomba`
`AngleSensorRoomba`
`DistanceSensorRoomba`

Wall

Summary: The Create has an infrared proximity sensor with a very low range on the front right of the bumper. The sensor only returns Boolean values of whether a wall is there or not, it cannot return distances.

Called by: `AllSensorsReadRoomba`

User-Added Sensors

Sonar

Summary: The simulator assumes four sonar sensors for finding distance are placed on the Create, facing in the cardinal directions, positioned at approximately the edge of the robot. The sonar model used to generate the parameters is the PING #28015. The relevant parameters are `rangeSonar` and `rangeMinSonar`, and are set in `CreateRobot.m` in the constant properties section. Noise parameters are set in the user-created configuration text file. If you do not plan to use sonar sensors, then do not call the function for reading them. If you plan to use fewer or more sonar, or put them in different locations, then the generation and visualization functions will have to be changed – see the `genSonar` section in the Code Documentation on page 27 for more information.

Called by: `ReadSonar`
`ReadSonarMultiple`

LIDAR

Summary: The simulator assumes a LIDAR sensor has been installed on front of the Create. The LIDAR model used to generate the parameters is the Hokuyo URG-04LX. The relevant parameters are `rangeLidar`, `rangeMinLidar`, and `angRangeLidar`, and are set in the constant properties section of `CreateRobot.m`. Noise parameters are set in the user-create configuration text file. If you do not plan to use a LIDAR sensor, then do not call the function for reading it. If you plan to change anything other than the linear and angular range of the LIDAR, see the `genLidar` section in the Code Documentation on page 27 for more information.

Called by: `LidarSensorCreate`

Camera

Summary: The simulator assumes a camera has been installed on the front of the Create. This camera is to be used for color blob detection only, the processing of which is done before the output is returned to the MATLAB control program. The camera is used for finding beacons. The relevant parameters are `rangeCamera` and `angRangeCamera`, and are set in the constant properties section of `CreateRobot.m`. Noise parameters are set in the user-create configuration text file. If you do not plan to use a camera, then do not call the function reading it. If you plan to change anything other than the linear and angular range of the camera, see the `genCamera` section in the Code Documentation on page 27 for more information.

Called by: `CameraSensorCreate`

Overhead Localization System

Summary: The simulator assumes there are cameras above the robot that are able to map the robot to the field very accurately using a two-dimensional barcode or something similar (Vicon system). The processing of the position and orientation is done before returning the output to the MATLAB control program. There are no parameters for this system.

See the Simplifications and Known Issues section in the Code Documentation on page 8 for an explanation. Noise for this system should be minimal, so there is currently no functionality for adding noise to the output. If you do not plan to use an overhead localization system, then do not call the function reading it. See the Code Documentation on page 28 for more information regarding this system.

Called by: `OverheadLocalizationCreate`

Unused in the Simulator

Battery

Summary: The Create can determine the capacity, remaining charge, current, and voltage of the robot's battery. The simulator will always return constant values for these since they have no meaning in the simulated environment.

Called by: `AllSensorsReadRoomba`
`BatteryChargeReaderRoomba`
`BatteryVoltageRoomba`
`CurrentTesterRoomba`

Dirt

Summary: The Roomba had dirt sensors in the vacuum, which were removed on the Create. The call to the dirt sensors (the Unused Bytes of the Open Interface Commands) will always yield false in both the simulator and on the real Create.

Called by: `AllSensorsReadRoomba`

Motor Current

Summary: The Create can detect if the motors are receiving too much current. The current limit for the wheel motors are 1.0 A. The simulator will always assume the motors are not overcurrented.

Called by: `AllSensorsReadRoomba`

Wheel Drop

Summary: The Create has three sensors to detect if the front caster wheel or the drive wheels are in their retracted or released wheel position. The wheels may move to their released (dropped) position from running over a cliff, in which case it may be a good idea to stop the robot from moving. There is no functionality for cliffs in the simulator, so these values will always be returned as false.

Called by: `BumpsWheelDropsSensorsRoomba`

Creating the Map

Elements of the Map

Walls – Represented by solid line. These are solid unmoving objects, and the robot is unable to pass through them. Walls will trigger bump, wall, sonar, and LIDAR sensors. The wall representation of a line has no thickness, so any object larger than a few centimeters should probably be represented by several walls around its perimeter.

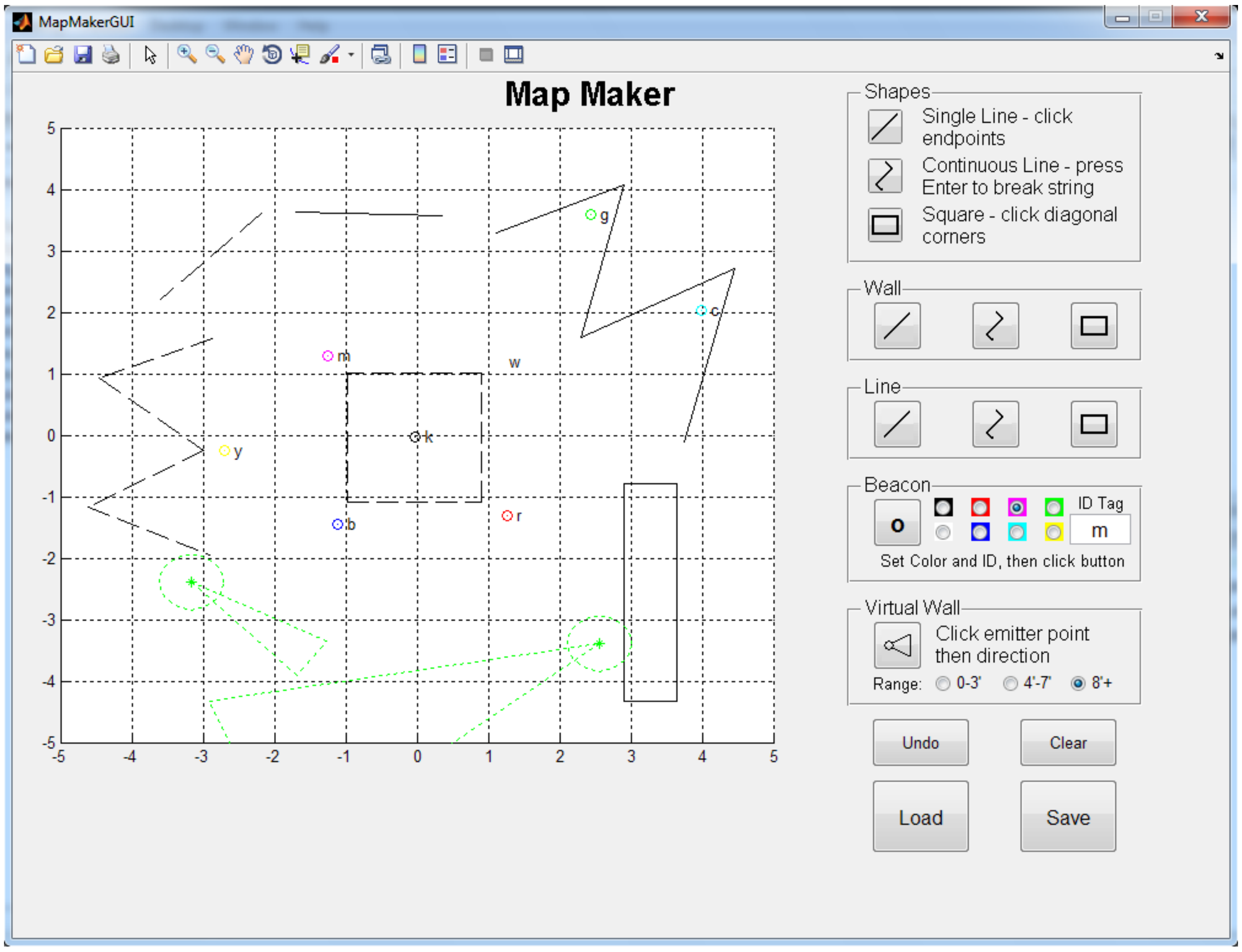
Lines – Represented by dashed lines. These are immaterial markings (such as black paint or tape) on the ground, and the robot passes over them, unaffected. Lines will change the reading of the cliff sensor signal strength, though they will not read as a cliff. Lines have no thickness, but the cliff sensors will read them over a certain width, so there is a finite time during which the sensors will be activated.

Beacons – Represented by colored circles. These are immaterial objects (such as colored paint or paper) on the ground, and the robot passes over them, unaffected. Beacons can only be detected by the camera. They each have an ID string associated with them, for labeling purposes only. These IDs default in `MapMakerGUI` to `b1`, `b2`,… for consecutive beacons.

Virtual Walls – Represented by a green asterisk and green dotted lines showing the range of the field. These are considered by the simulator to be immaterial objects, and the robot passes through them, unaffected. The real virtual wall has an emitter device that would, under normal circumstances, be moved or knocked down if hit by the robot. These occurrences are ignored to simplify simulation. The asterisk represents the location of the emitter, the dotted circle shows the range of the "halo" around the emitter, and the triangle shows a simplified shape of the main signal field. When the robot has its IR receiver within the halo or the signal field, the virtual wall sensor will be triggered. If there is a wall between the IR receiver and the emitter, neither the simulator robot nor the real Create will not pick up the virtual wall signal, even though the representation appears to go through the wall.

Using MapMakerGUI

`MapMakerGUI` is an intuitive interface used to create map files for use in the simulator. Each map contains information about the positions of walls, lines, beacons, and virtual walls. This information is saved in a particular format to a text file. To start the program, run `MapMakerGUI` as a function with no input or output arguments from the command window.


MapMakerGUI
Map Maker
Shapes
Single Line - click endpoints
Continuous Line - press Enter to break string
Square - click diagonal corners
Wall
Line
Beacon
ID Tag
m
Set Color and ID, then click button
Virtual Wall
Click emitter point then direction
Undo
Clear
Load
Save
g
c
m
w
k
y
b
r

Range: 0-3' 4'-7' [✓ 8'+]

Button Functionality:

Click twice on the plot to designate the endpoints of a single line. Note: in this instance "line" is simply referring to the geometric figure.

Click as on the plot many times as you like to create lines from one point to the next. Press any key to end line creation.

Click twice to specify opposite corners of a rectangle. The sides are always vertical and horizontal.

Choose the color and the ID for the beacon (if no ID is ever set, it defaults to `b1`, `b2`, etc). Then click the button at the left and click the point on the map to place the beacon.

Choose the range setting for the virtual wall, then click the button at the left. Click the emitter point, and then click in the direction you want the virtual wall to extend towards.

Undo — Press Undo or hit Ctrl+z on the keyboard to reverse the most recent map element placement. This will not be able to undo a load, clear, or save. There is no redo.

Clear — Press Clear, and then confirm choice on pop-up box to delete all map element information and clear the plot.

Load — Press Load and navigate to the desired file to load a previously created map.

Save — Press Save, then enter a filename to save the current map. Do not append "`.txt`" to the file name. Confirmation is required if you want to save over an existing file.

Making the Map Manually

The map can be created in the text file without using `MapMakerGUI` to more precisely control the positions of the map elements. There are a few formatting rules to be followed to allow the simulator to read the files correctly. Note that the normal usage of "lines" (as in "press enter to move the cursor to the next line") will be replaced by "rows" to avoid confusion with the map element "lines".

Comments and blank rows are allowed. Any row that does not start with "`%`", "`wall`", "`line`", "`beacon`", or "`virtwall`" will be unrecognized and a warning will display to the command window.

Walls are defined by a row that begins with "`wall`", followed by the Cartesian coordinates of the first point, followed by the Cartesian coordinates of the second point (wall x_1 y_1 x_2 y_2). An example is:

```
wall 2.30 1.59 4.46 2.71
```

Lines are defined similarly to walls, but the opening identifier is "`line`". Example:

```
line -4.612 -1 2.905 -1.95
```

Beacons are defined by a row that begins with "`beacon`", followed by the Cartesian coordinates of the beacon, then the beacon's color vector, then the beacon ID (beacon x y [r g b] id). An example is:

```
beacon 1.267 -1.310 [1.0 0.5 0.0] beac04
```

More information on color vectors can be found at:
http://www.mathworks.com/access/helpdesk/help/techdoc/ref/colorspec.html

Virtual walls are defined by a row that begins with "`virtwall`", followed by the Cartesian coordinates of the emitter, then the angle (in radians) of the direction of emission relative to the positive x-axis, then an identifier for the range setting of the wall (1 is the lowest range, 3 is the highest) (virtwall x y θ r). An example is:

```
virtwall 2.560 -3.390 -2.725 2
```

The current version of the map parser allows for any whitespace (spaces or tabs) to be the delimiter between elements in a row. Leading and trailing whitespace are allowed as well. It is recommended to use the MATLAB editor to create the text files to avoid issues with different encoding types not recognizing newline or other whitespace characters.

Creating the Configuration File

Elements of Configuration

Communication Delay – Determines the response time for the robot. Increasing this value will cause more of a delay for all function calls to the robot (for both action command functions and sensor reading functions).

Noise Data – Determines the fluctuation in the readings from the sensors. This is defined by the mean and standard deviation of the noise values. Different sensors have varying definitions for what the noise means, so they are explained below.

Wall Sensor – The noise value changes the effective range of the wall sensor. The mean of the noise will provide an offset from the defined range (specified as the property `rangeIR` in the Constant Properties section of `CreateRobot.m`). The standard deviation will give the variance in the effective range.

Cliff Sensors – With no noise, the cliff sensor signal strengths will read 1.5% when over a line, and 21.5% when not. The real values of the readings will vary based on room lighting, floor color and material, and the sensors themselves. If real sensors tend to read higher or lower, set a value in the noise mean that will change the sensor output. The standard deviation adds to the variance in the readings.

Odometry – The odometry noise parameters should be given as a percentage of the true measurement. For example, if `DistanceSensorRoomba` consistently reads 2.2 meters, when the robot has only traveled 2 m, the mean should be set to 0.1 to signify an offset of 10%. Or if the standard deviation of readings from `AngleSensorRoomba` after turning 6 radians is 0.3 radians, then the noise standard deviation should be set to 0.05 to signify 5% variation. Both angle sensors and distance sensors use the same noise parameters.

Sonar – Noise mean and standard deviation specify the difference of the readings from the real distance. All sonar sensors use the same noise parameters.

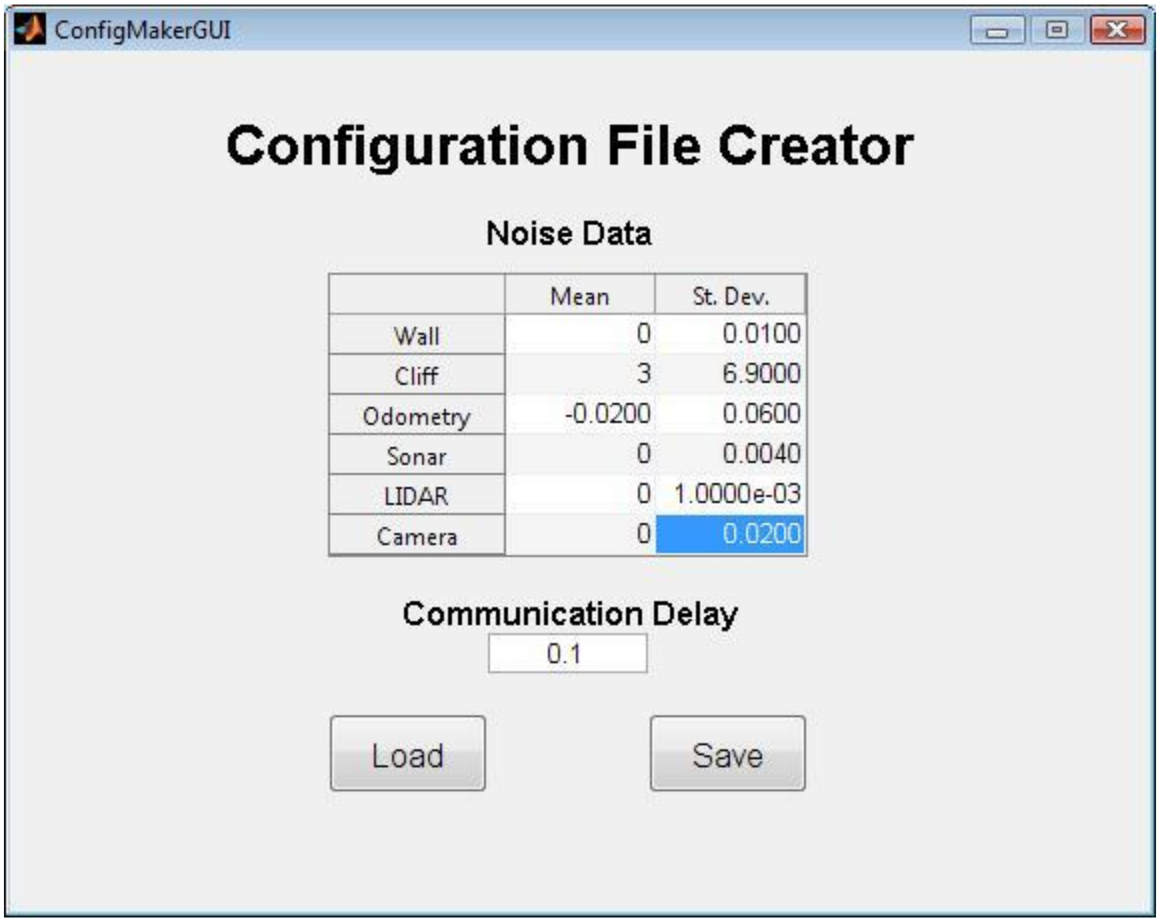
LIDAR – Noise mean and standard deviation specify the difference of the reading from each point on the LIDAR range from the real distance.

Camera – Noise mean and standard deviation specifies the difference between the readings of the distance and angle to the beacons, and the real values, in percent. This is similar to the odometry noise. There is no noise on the color reading. The distance and angle readings use the same noise parameters.

Estimated values from tests conducted on the wall, cliff, and odometry sensors are in ExampleConfigFile.txt that comes packaged with the toolbox. The values for the other sensors are based off of product specifications (Sonar and LIDAR), or pure guesswork for the rest.

Using ConfigMakerGUI

`ConfigMakerGUI` is a simple interface for editing and generating configuration files. To start the program, run `ConfigMakerGUI` as a function from the command window with no input or output arguments.



Configuration File Creator

Noise Data

	Mean	St. Dev.
Wall	0	0.0100
Cliff	3	6.9000
Odometry	-0.0200	0.0600
Sonar	0	0.0040
LIDAR	0	1.0000e-03
Camera	0	0.0200

Communication Delay: 0.1

Instructions:

When `ConfigMakerGUI` loads, the noise and communication delay values default to zero. If this is given as the configuration file to the simulator, this would simulate an ideal robot.

Push the Load button, if desired, to import data from an old configuration file.

Change the values for the noise mean and standard deviation, as well as the communication delay as desired.

Push the save button and enter a filename to export the data to a text file.

Making the Configuration File Manually

The following formatting rules must be obeyed:

Comments and blank rows are allowed. Any row that does not begin with a sensor name, "`com_delay`", or "`%`" will be unrecognized and a warning will display to the command window.

The communication delay parameter can by specified by a row that begins with "`com_delay`" followed by the communication delay (in seconds). An example is:

```
com_delay 0.1
```

All other rows (that are not comments or blank) should specify a sensor, its noise mean value, and its noise standard deviation. An example is:

```
odometry 0.092 0.05
```

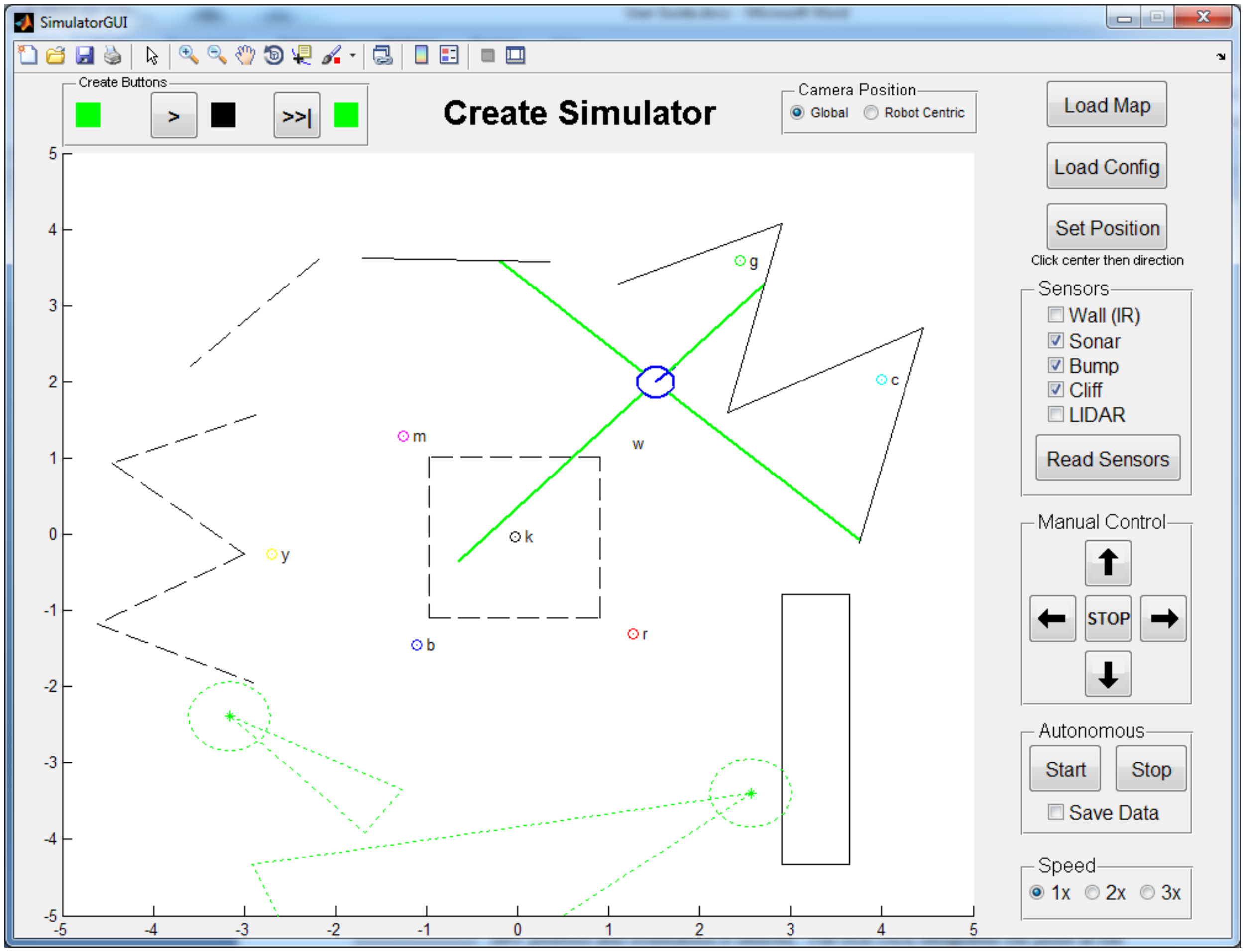
The only acceptable sensor names are: "`wall`", "`cliff`", "`odometry`", "`sonar`", "`lidar`", and "`camera`".

Similarly to the map file, leading or trailing spaces or tabs are allowed. The ordering of the rows in the file does not matter.

Using the Simulator

Setting up the Simulator

First, open MATLAB and enter “`SimulatorGUI`” into the command window. Allow a few seconds for the GUI to load and visualization to initialize. The simulator defaults to a blank map and no sensor noise.

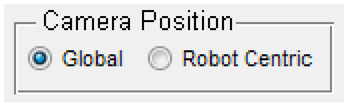


Click the Load Map button and navigate to the desired map file to set up the environment of the robot.

Click the Load Config button and navigate to the desired configuration file to set up the noise and communication delay parameters for the robot.

Click the Set Position button, then click two points on the map to set the robot’s new position and orientation if desired. The first click designates the point at the center of the robot, and the second point specifies the direction the robot will face.

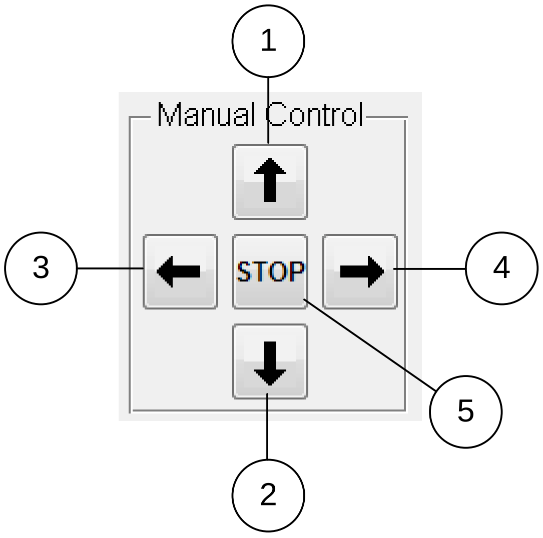
Viewpoint control

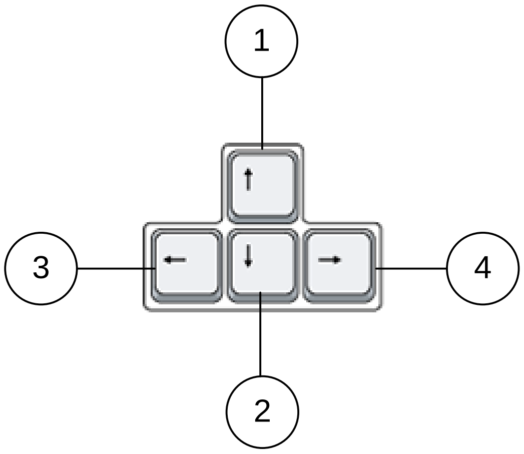


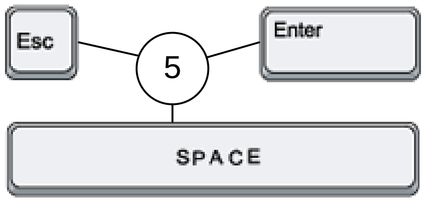
The Camera Position settings allow for adjustment from the Global viewpoint to Robot Centric. Global view will not move unless changed by the figure toolbar, while the Robot Centric view will always keep the robot in the center of the viewpoint. Switching back to the Global view from Robot Centric will merely cause the camera to stay in the same position. The figure toolbar in the upper left can be used to pan and zoom with the camera. Note that none of the buttons or keyboard controls can be used while using the figure toolbar. After clicking a button in the figure toolbar, click it again to toggle it off in order to use the GUI buttons again.

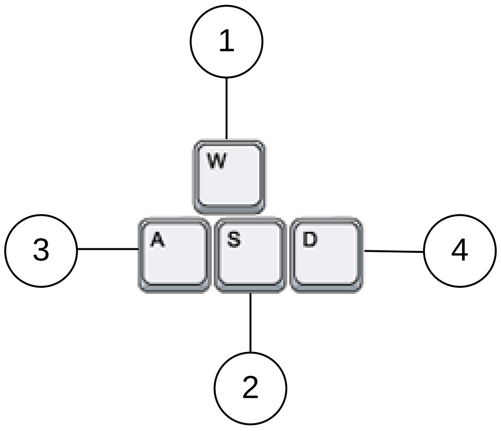
Manual Control

While the autonomous control function is not running, manual control is enabled. You can control the motion of the robot by setting forward and turning velocity using the arrow buttons, as well as some keyboard buttons. Note that angular velocity is defined as positive in the counter-clockwise direction.



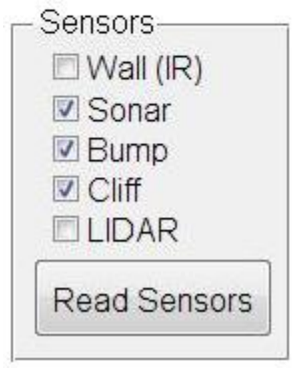






1. Increase forward velocity
2. Decrease forward velocity
3. Increase angular velocity
4. Decrease angular velocity
5. Zero all velocities

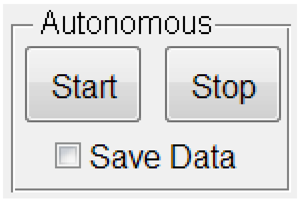
Sensors Visualization



These checkboxes control the visualization of the main sensors on the plot of the robot and map. The wall sensor shows a red line that extends out to its full range if a wall is detected. The sonar visualization shows green lines extending out of the four sonar sensors to the nearest wall, or the sonar's full range if no wall is close enough. The bump sensor visualization shows a pink line in a location depending on which sensor is activated. The cliff sensor visualization shows blue asterisks in the locations of the cliff sensors when they are active. The LIDAR visualization extends a few lines within the LIDAR range to the nearest wall, or the limit of the range.

When the Read Sensors button is pressed, the simulator will read the Bump, Cliff, Wall, Virtual Wall, LIDAR, Camera, Sonar sensors, Odometry data, and Overhead Localization information. This will be output to the command window, and the LIDAR output will be plotted in a new figure. This takes a couple seconds, so give it some time before pressing further manual control buttons. The information is also not read simultaneously, so it will reduce discrepancy if the robot is not moving when the button is pushed. Also, this will reset the odometry data, so it would be best to avoid pressing this during autonomous control mode if the control program depends on odometry for navigation.

Autonomous Control

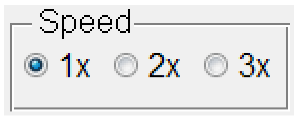


To call the user-created autonomous control program, click on the Start button in the Autonomous section. Navigate to the correct autonomous file and push Open. This will run the autonomous code and disable manual control. To exit autonomous mode and re-enable manual control, push the Stop button. Note that the 'stop' will only come into effect on the next function call from the autonomous code to one of the robot control functions (from the toolbox). Under normal circumstances, this is almost instantaneous. However, if the control program utilizes calls to `travelDist`, `turnAngle`, the built-in function `pause`, or contains large computations, then the code may not exit for some time. Closing the GUI is also an acceptable way to terminate the autonomous function.

Data Output

To assist in debugging of code, a data file can be saved to the current directory at the end of the autonomous code execution. This data file contains the time history of the position and orientation of the robot, as well as the information passed to and from the control program. To set the simulator to save the data, make sure the "Save Data" checkbox is marked by the end of the autonomous code execution. The filename defaults to `SimulatorOutputData_*.mat`, where `*` is a number that will increment so as to not save over any data currently there.

Simulation Speed

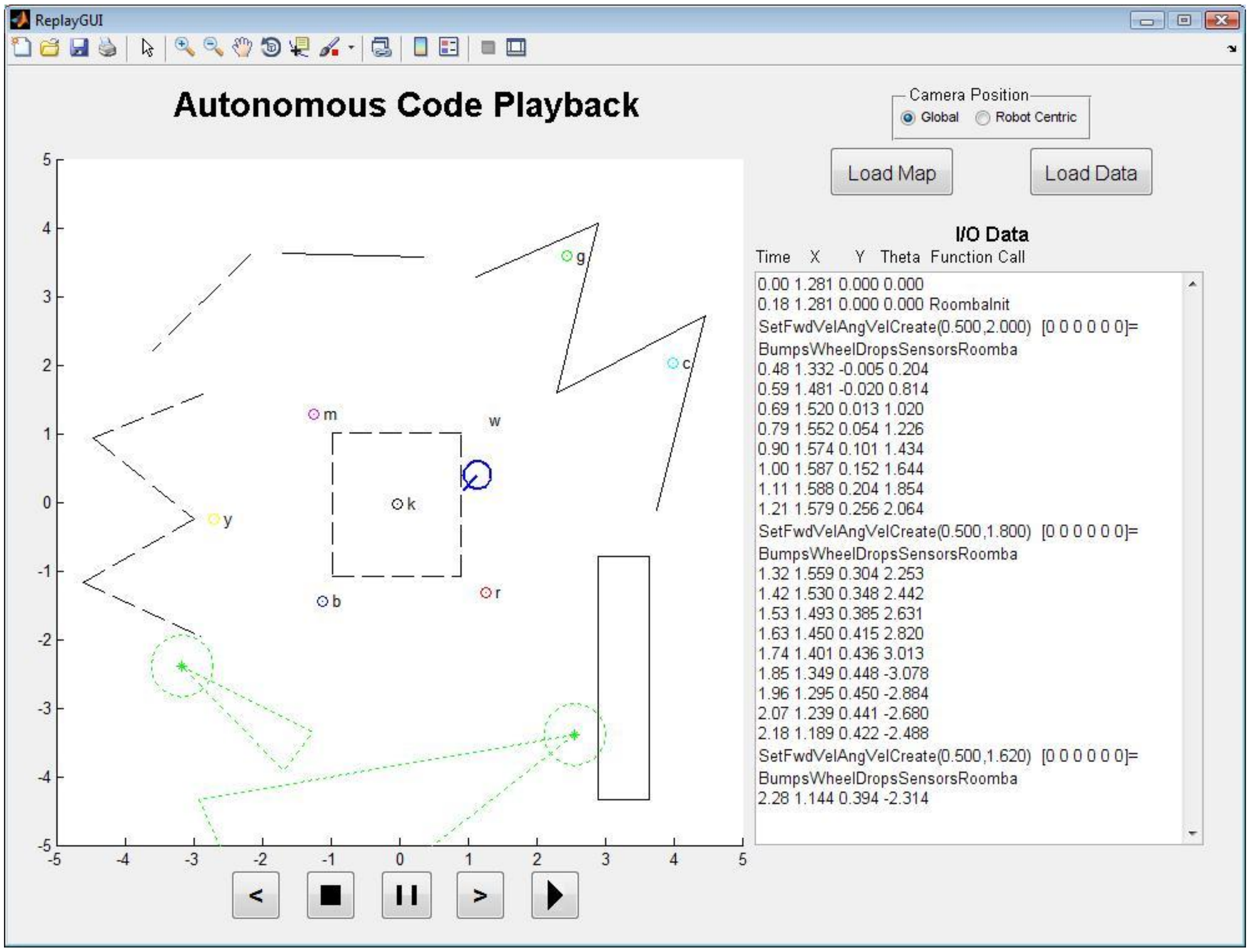


The speed at which the simulation runs can be changed using the radio buttons pictured here. This will effectively make the robot move faster across the map without changing how it perceives its own speed. However, this will not change the rate at which the simulation updates, nor how fast the control program runs. This can cause errors partially moving through walls before the physics engine notices. It also will mess with the control program if the `tic-toc` functions are used to determine turning or movement distance. Finally, if the data output checkbox is marked, the output data will remember the sped up run, instead of the real time. **To maximize the accuracy of the simulator, it is recommended that this feature not be used.**

Debugging

Using ReplayGUI

`ReplayGUI` is an intuitive interface for demonstration and debugging purposes. It will visualize the movement of the robot during autonomous execution based on the saved data from the simulator output. It will also display the time, position, orientation, and function calls of all time-steps up to and including the current point.



```
Time   X     Y   Theta  Function Call
0.00 1.281 0.000 0.000
0.18 1.281 0.000 0.000 RoombaInit
SetFwdVelAngVelCreate(0.500,2.000)  [0 0 0 0 0 0]=
BumpsWheelDropsSensorsRoomba
0.48 1.332 -0.005 0.204
0.59 1.481 -0.020 0.814
0.69 1.520 0.013 1.020
0.79 1.552 0.054 1.226
0.90 1.574 0.101 1.434
1.00 1.587 0.152 1.644
1.11 1.588 0.204 1.854
1.21 1.579 0.256 2.064
SetFwdVelAngVelCreate(0.500,1.800)  [0 0 0 0 0 0]=
BumpsWheelDropsSensorsRoomba
1.32 1.559 0.304 2.253
1.42 1.530 0.348 2.442
1.53 1.493 0.385 2.631
1.63 1.450 0.415 2.820
1.74 1.401 0.436 3.013
1.85 1.349 0.448 -3.078
1.96 1.295 0.450 -2.884
2.07 1.239 0.441 -2.680
2.18 1.189 0.422 -2.488
SetFwdVelAngVelCreate(0.500,1.620)  [0 0 0 0 0 0]=
BumpsWheelDropsSensorsRoomba
2.28 1.144 0.394 -2.314
```

Button Functionality:

Load Map	Click the Load Map button and navigate to the desired map file to visualize the environment. The map is for visualization only and does not affect the robot.
Load Data	Click the Load Data button and navigate to the desired data file to allow replaying of the autonomous execution. The replay will load the first point.
<	Rewind slow button. This will cause the replay to step backwards through the data at ¼ of normal speed.
■	Stop button. This will cause the replay to halt the playback and start over at the beginning.
❚❚	Pause button. This will cause the replay to halt playback at the current point.
>	Forward slow button. This will cause the replay to step forward through the data at ¼ of normal speed
▶	Play button. This will cause the replay to step forward through the data at normal speed.
← ↑	Up and left arrow keys. These step one frame backwards through the playback. If held down, the playback rewinds quite quickly.
↓ →	Down and right arrow keys. These step one frame forwards through the playback. If held down, the playback fast-forwards quite quickly.

Note that the viewpoint is controlled in the same way as `SimulatorGUI` (page 18).

Inspecting the Output Data

To inspect the data file, double-click on the file in the MATLAB current directory browser, which will import the variable `datahistory` into the workspace. Note that this will replace the value of `datahistory` if is already defined in the workspace. The variable can be inspected by double-clicking on it in the workspace browser. The format of the data is a cell array, in which the first column is the time since the beginning of autonomous execution, the second column is the x-position, the third is the y-position, the fourth is the angle of the robot (relative to the positive x-axis), and the fifth is a string containing autonomous code function call information.

In the function call representation, the input and output parameters will be in their correct positions in the function call, and they will display their numerical values. The serial port object as an input or output argument will be ignored completely though. If more than one function call was made at any one time step, the fifth cell in a row will contain another cell array. This will be a vertical cell array, with each cell containing a function call at that time.

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

Creaters 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 on MATLAB Central)
Zhenhai Wang (circle on MATLAB Central)

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

Index