SimMeR

Simulator for **Me**chatronics **R**obots

Version 0.2

UNIVERSITY OF TORONTO

Ian Bennett

Developed for the University of Toronto Department of Mechanical and Industrial Engineering

Contents

1 Abstract

This document presents the design and key features of a robotics simulator for MIE444 Mechatronics Principles. The simulator is a software tool written in MATLAB, designed to allow students to test navigation and localization algorithms remotely during the COVID-19 pandemic. The simulator provides software inputs and outputs for several different types of sensors and is plug and play compatible with a similar Bluetooth communication program used to send commands to a microcontroller on board a physical robot. Communication with the simulator is performed using TCP protocol, and is compatible with any programming language that can send and receive data in the correct formats, including MATLAB and Python.

The program begins by generating a maze and user robot from a number of user editable text files. These files will contain the maze wall locations, robot sensor suite and mounting locations, and control character mapping for the sensors and movement. Measurement and control precision is also definable by the user and can be configured to replicate the types of error seen in real world systems based on the configuration chosen by the user. The simulated robot is displayed on a plot, and all data from the simulator can be accessed by the user if needed.

2 Change Notes

2.1 Version 0.1

September 27, 2020

1. Initial Release

2.2 Version 0.2

October 16, 2020

1. Added "Change Notes" section to this document
2. Completed "Configuration Files" section of this document
3. Updated format of data returned by the simulator to single precision float
4. Added requirement for a newline character at the end of messages to simulator

3 Introduction

This simulator was created as part of a project for the University of Toronto Department of Mechanical and Industrial Engineering, for the course MIE444 Principles of Mechatronics. The program including all scripts, functions, files, and documentation are distributed under the GNU AGPLv3 license. The author would like to acknowledge the work of Douglas M. Schwarz, whose program **intersections.m** is used in this program. The license for **intersections.m** is available in the folder "documentation" within the main directory.

SimMeR is written in MATLAB using version 2020a, the most recent version available at the time of creation. No packages other than the standard install are used aside from the instrument control toolbox (included in most standard and student licenses), which is used to communicate over TCP with the simulator. Some open-source, freely available functions are used, including **intersections.m**, but only when they provide functionality that can't be easily replicated using the standard installation.

The goal of the project component of the course MIE444 is to have students design and construct a robot to perform simple mapping and localization to autonomously collect a small object from within a maze and deposit into a predetermined zone.

To this end, students must program mapping and localization algorithms to take and fuse input data from a number of sensors and create output commands to control motors that will move the robot. The simulator provides an environment in which these algorithms can be tested without actually having to build the and operate it in the maze.

3.1 Program overview

SimMeR performs the following tasks as part of its initial setup.

1. Loads/generates the maze, including wall locations and checkerboard

locations.

2. From a user-defined text file, generates a simulated robot including size, shape, and sensor loadout and placement.

3. Creates a TCP server object to listen for commands from the control algorithm.

4. Creates a text file that data from each step the robot takes will be appended to for users to view and debug with (**not yet implemented**).

As part of its main command loop, SimMeR performs the following steps:

1. Accepts a TCP packet command from the control algorithm programmed in MATLAB, Python, or another language.

2. Funnels the packet to either the simulator or a Bluetooth serial port (**not yet implemented**) connected to a physical robot.

3. If using the simulator, parses the packet to extract the command code.

4. Search a user-defined lookup table to determine whether the packet is a telemetry request (i.e. to collect sensor data) or a control command to move the simulated robot in a direction, then acts on the command.

    a. For telemetry requests, these are passed to an appropriate function that simulates the response data packet of the desired sensor. The function takes into account the position of the simulated robot within the maze, generates the value that the sensor should read, adds noise based on a defined noise profile for that sensor, and constructs a return data packet that is identical to one produced by the physical robot's microcontroller.

    b. For control commands, these will be passed to a function that moves the simulated robot within the maze. The drive distance

and direction and/or the rotation value for the configuration is specified by the user algorithm. Error is added based on a user-defined drive error profile, and used to update the simulated robot's position. Several drive noise profiles will be preconfigured by the course teaching assistants based on common wheel/motor configurations (**not yet implemented**).

5. For a control command, the program generates a path based on the movement. If a collision is detected between the robot and the wall along the path, the program assumes that the robot stops moving at the collision point.

6. Based on the new location and/or rotation of the robot, positions of the sensors are updated. For integration-based sensors (such as gyroscopes and odometers), their values are updated based on the movement of the robot and user-defined noise profiles.

3.2 Compatibility

In order to ensure that the user-created programs work with both the simulator and a physical robot, they both must take and interpret commands and respond to the environmental stimulus similarly. The input and output software interface are designed to be as similar as possible. To this end, in addition to the simulator, a Bluetooth command piping program will be created to pass data from the user program to a Bluetooth module on the physical robot (**not yet implemented**). The physical robots' microcontrollers should be programmed to read commands and output data over Bluetooth in the same way as SimMeR does for ease-of-use. For more information on data formats, see Section 5.5.1.

4 Configuration Files

This section details the various configuration files used by the robot simulator to do the following things:

1. **maze.csv** - Define the maze wall locations.
2. **robot.csv** - Define the user's robot size and shape
3. **sensors.csv** - Define the sensor loadout, locations, error & definition parameters, and command codes.
4. **drive.csv** - Define the user's robot's drive system, associated error/bias parameters, and command codes.

Note that the checkerboard pattern is not predefined in a config file. It is generated using a fixed-seed random number generator within the function **import_checker.m**.

4.1 Maze Wall Locations

Relevant Files

1. **maze.csv** - maze definition configuration file
2. **import_maze.m** - MATLAB function to import and create the maze

The file **maze.csv** in the config folder contains the definition for the maze walls. Maze walls are defined using a 4 row by 8 column matrix, with each cell containing a number between 0 and 3. Each of these numbers defines a 1 foot by 1 foot square in the maze as either an square surrounded by a wall (0), an empty square (1), a robot starting location (2), or a block location (3). If the maze dimensions are to be changed, this can be done by editing the variables *dim1* and *dim2* in the **import_maze.m** file, and adding additional rows/columns to the **maze.csv** configuration file.

The function **import_maze.m** creates an n-by-2 matrix *maze_xy* that defines the (x,y) perimeter points of each of the wall squares, each separated by a pair of NaN values. The function begins by drawing an outer rectangle around the perimeter of the maze as defined by the *dim1* and *dim2* variables (the x and y dimensions of the maze perimeter, in feet), and stored in *maze_xy*. The imported data from **maze.csv** is iterated through, and for each square indicated as surrounded by a wall (value 0) a 1 by 1 foot square is appended to *maze_xy* via its perimeter points. Note that the corner points of the outer wall are added in a counter-clockwise order, and the corner points of each internal square is added in a clockwise order. This enables collision-checking functionality for the rover when it moves, later in the program.

Before passing the variable *maze_xy* out of the function, it is multiplied by 12 to convert the dimensions from feet to inches.

4.2 Robot Shape

Relevant files

1. **robot.csv** - robot definition configuration file
2. **import_bot.m** - MATLAB function to import and create the robot

The file **robot.csv** in the config folder is used to define the robot perimeter size and shape. Column A contains labels, and doesn't affect the robot, while Column B contains data. The first row (cell B1) is used to define whether the robot is a circle (0) or rectangle (1). The second row (cell B2) defines the robot diameter in inches if the robot is circular, or the x-dimension in inches if the robot is rectangular. The third row (cell B3) defines the robot y-dimension in inches if the robot is rectangular, and has no effect if it is circular.

The function **import_bot.m** reads **robot.csv** and converts it to a shape composed of perimeter points as defined by the file. The first column de-

fines the x-coordinates, and the second column defines the y-coordinates, defined in a clockwise direction. The first point and last point must be the same to ensure that the shape is closed. If it is desired to use a custom perimeter shape instead of the standard circle or rectangle, the output of **import_bot.m** can be replaced with a list of x-y points in the same format. The robot centroid for purposes of movement and sensor position definition is assumed to be at the origin (0,0).

4.3 Sensors

Relevant files

1. **sensors.csv** - sensor loadout definition file
2. **import_sensor.m** - MATLAB function to import the sensors

The file **sensors.csv** in the config folder is used to define the robot's sensors, include the type, location, and orientation of each. Figure 1 shows the layout of an example file, with each row representing a different sensor. Each column is used to define a particular characteristic of the sensor.

	A	B	C	D	E	F	G	H	I	J
1	ID	Poll Character	Enabled	X Position (in)	Y Position (in)	Z Position (in)	Rotation (deg)	Percent Error (0-1)	Field of View (deg)	Threshold
2	ultra1	u1	1	3	0	6	0	0.02	5	0
3	ultra2	u2	1	2.12	2.12	6	45	0.02	5	0
4	ultra3	u3	1	-3	0	6	180	0.02	5	0
5	ultra4	u4	1	2.12	-2.12	6	315	0.02	5	0
6	ultra5	u5	1	3	0	2	0	0.02	5	0
7	ultra6	u6	1	0	0	0	0	0	5	0
8	ultra7	u7	1	0	0	0	0	0	5	0
9	ultra8	u8	1	0	0	0	0	0	5	0
10	ir1	i1	1	2	0	1	0	0.05	40	0.8
11	ir2	i2	1	0	2	1	0	0.05	40	0.8
12	ir3	i3	1	-2	0	1	0	0.05	40	0.8
13	ir4	i4	1	0	-2	1	0	0.05	40	0.8
14	comp1	c1	1	0	0	6	0	0.1	0	0
15	odom1	o1	1	3	0	1	90	0.02	0	0
16	odom2	o2	1	-3	0	1	90	0.02	0	0
17	odom3	o3	1	0	3	1	0	0.02	0	0
18	odom4	o4	1	0	-3	1	0	0.02	0	0
19	gyro1	g1	1	0	0	4	0	0.01	0	0

Figure 1: Sample sensor configuration .csv file

4.3.1 Column A - ID

Column A is the sensor ID. This defines the type of sensor. Five sensor types are supported, defined as a text string unique to the sensor followed by a number. The below tags **MUST** be used in this column, they can not be replaced with custom tags. Up to ten sensors of each type can be used.

1. **ultra** - Ultrasonic sensor
2. **ir** - Downward facing infrared sensor (line-following sensor)
3. **comp** - Compass
4. **odom** - Wheel odometer
5. **gyro** - Gyroscope (rotation only)

4.3.2 Column B - Poll Character

The Poll Character is a two-character code that is used to command the simulator (or robot if communicating via Bluetooth serial). These can be customized by the user to be anything in the format "letter"-"number". The letter should be consistent among sensors, i.e. if using "u1" for one ultrasonic sensor, you should use "u2" for the second, "u3" for the third, etc. If using "s1", you should use "s2", "s3", etc.

4.3.3 Column C - Enabled

This column should be either 1 or 0, indicating whether the sensor is enabled or disabled, respectively. This column allows for sensors to be temporarily disabled without deleting them from the file.

4.3.4 Column D-F - X-Position, Y-Position, Z-Position

These three columns define the position of a sensor, in inches from the center (X & Y) and base (Z). For ultrasonic sensors, the Z-position determines if the sensor can detect a block. For IR sensors, Z-position determines the area visible to the sensor. For compasses and gyroscopes, these numbers must be entered but do not factor into calculations. For information on the local and global reference frames for the robot, see Section 5.3.

4.3.5 Column G - Rotation

This column denotes the rotation of a sensor relative to the body of the robot, in degrees from the X-positive direction. For compasses, gyroscopes, and IR sensors, these numbers must be entered but do not factor into calculations. For information on the local and global reference frames for the robot, see Section 5.3.

4.3.6 Column H - Percent Error

This column indicates the normally distributed random error added to measurements by the simulator for the sensor. This is defined on a scale from 0 to 100%, where 1% error is entered as 0.01. Equation 1 is used to calculate the error. V_i is the true value, R_n is a random number selected from a weighted normal distribution centered about 0, and P is the percent error value for the sensor.

$$V = V_i * (1 + R_n * P) \tag{1}$$

4.3.7 Column I - Field of View

This column represents the field of view for sensors where it is relevant, in degrees (full-angle). This is only relevant for ultrasonic and IR sensors. For example, a field of view of 5 degrees means that a cone is visible to the sensor

from +2.5° to -2.5° off the center axis. For all other sensors, this value must be entered but is not used in calculations.

4.3.8 Column J - Threshold

This value is only used by IR sensors, for all other sensors it must be entered but is not used in calculations. This column represents the percentage of the field of view of an IR sensor that must be black before it registers as black. This is entered as a value from 0 to 1, where 0.8 corresponds to 80%.

4.4 Drive

Relevant files

1. **drive.csv** - drive definition file
2. **import_drive.m** - MATLAB function to import the drive mechanism and commands

Similar to **sensors.csv**, the drive config file is used to import information about the drive mechanism of the robot to be simulated or tested. An example drive configuration is shown in Figure 2 for a robot with 4x Omniwheel drive, capable of moving in each axis and and rotating independently.

	A	B	C	D	E	F	G	H	I
1	ID	Poll Character	Enabled	Y-Axis Error	X-Axis Error	Rotation Error	Y-Axis Bias	X-Axis Bias	Rotation Bias
2	up	w1	1	0.05	0.02	0.02	0	0	0
3	down	s1	1	0.05	0.02	0.02	0	0	0
4	left	a1	1	0.02	0.05	0.02	0	0	0
5	right	d1	1	0.02	0.05	0.02	0	0	0
6	rot	r1	1	0.003	0.003	0.02	0	0	0

Figure 2: Sample drive configuration .csv file

4.4.1 Column A - ID

Column A is the direction ID. Five drive directions are supported, defined as a text string unique to the direction followed by a number. The below tags **MUST** be used in this column, they can not be replaced with custom tags. Unlike for the sensors, each of these can only be used once.

1. **up** - Upward movement (Y+)
2. **down** - Downward movement (Y-)
3. **left** - Left movement (X-)
4. **right** - Right movement (X+)
5. **rot** - Rotational movement

For information on the local and global reference frames for the robot, see Section 5.3. Typically, right would be defined as "forward" but it is up to you to determine and configure this for your design.

4.4.2 Column B - Poll Character

The Poll Character is a two-character code that is used to command the simulator (or robot if communicating via Bluetooth serial). These can be customized by the user to be anything in the format "letter"-"number". The letter should be unique for each direction, and **MUST NOT** overlap with any of the sensor poll characters.

4.4.3 Column C - Enabled

This column should be either 1 or 0, indicating whether the drive direction is enabled or disabled, respectively. For instance, if a robot is not using omni-wheels or a similar drive mechanism, the left & right directions can be disabled to accurately represent the drive mechanism.

4.4.4 Column D-F - Y, X-Axis Error & Rotation Error

These columns, similar to the Percent Error column in the **sensors.csv**, define the normalized random component of error in either a direction or rotation axis when the robot is commanded to drive or rotate in a direction. This is similar to the process described in Section 4.3.6. Equations 2, 3, and 4 illustrate exactly how the movement is calculated, where the terms are the same as in Equation 1 with subscripts x and y indicating X- and Y- axis values.

$$V_x = (V_{xi} + V_{yi} + V_{ri}) * R_n * P_x + V_{xi} \tag{2}$$

$$V_y = (V_{xi} + V_{yi} + V_{ri}) * R_n * P_y + V_{yi} \tag{3}$$

$$V_r = (V_{xi} + V_{yi} + V_{ri}) * R_n * P_r + V_{ri} \tag{4}$$

4.4.5 Column G-I - Y, X-Axis Bias & Rotation Bias

These columns define the bias, or persistent error, for each of the drive commands. They work similarly to the normalized error in Section 4.4.4. For the Y-axis and X-axis bias, they are defined such that for each inch commanded, the value in the bias column is also moved in the appropriate direction. For example, If a robot has an X-bias of 0.1 in the *up* direction, if it is commanded to move right 2 inches, it will move right 1 inch and drift upward 0.2 inches, plus any error calculated as in Section 4.4.4. Rotation bias is calculated similarly, except inches moved is replaced with degrees rotated. Similar to the previous example, if a value of -3 is listed in the Rotation Bias column for the *right* direction, if the robot is commanded to move 2 inches to the right, it will, at the same time, rotate -6 degrees while executing the movement. For the rotation movement command, the biases act similarly, where each degree of rotation commanded translates to either an inch of movement bias or degree of rotation bias. The scale factors must be set appropriately in order to ensure representative biases in the drive. There is a way to randomize

these biases for each simulation run within the simulator code, if a user does not wish to hard code values into the config file. If randomized biases are used, they will overwrite the values entered in these columns.

5 Simulator User Guide

This section is a user guide intended for anyone who needs to use the simulator rather than make changes to its code. Prior to reading this section, readers should read Section 4 to ensure they understand how to properly set up the simulator.

5.1 Matlab setup

Ensure that Matlab version 2020a or newer is installed. The simulator may work with older versions, but keep in mind that it has NOT been tested with them for compatibility. Download the SimMeR code from GitHub, and unpack it to a location on your computer from which it can be run.

To run the simulator, you must run the script *Main.m* from the main folder. All functions that are needed are contained in the main folder, you should not need to add any additional paths for functionality. Ensure configuration .csv files are located in a folder called **config** within the main folder. This is the default location.

The simulator uses TCP to communicate with a control algorithm. This algorithm can be written in Matlab, Python, or another language that supports sending strings and double-precision floating point numbers over TCP packets. It has only been tested with Matlab, so please be aware if you are trying to use another language that some debugging may be necessary. TCP packets are exchanged over the *localhost*, and so an internet connection is not required as long as SimMeR and the control algorithm are running on the same computer. Your firewall may block Matlab's access to *localhost* by default. If you can't get your control algorithm software to connect to SimMeR, you may have to grant Matlab access (the popup shown in Figure 3 shows Windows firewall blocking it). SimMeR uses port 9000 and 9001 to receive commands and send replies, by default. These can be changed by editing the values passed to **tcp_setup.m**.

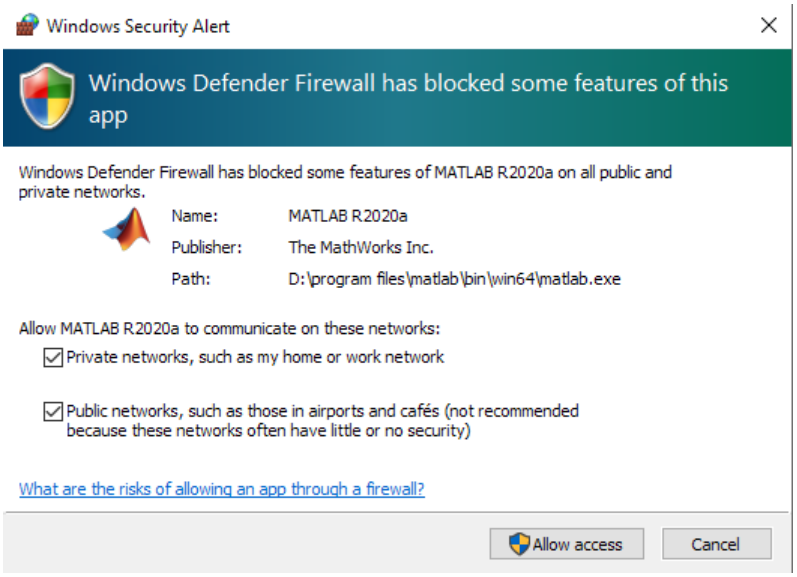

Figure 3: Windows firewall blocking Matlab TCP access.

5.2 SimMeR Display Window

SimMeR displays a visualization of the robot traversing through the maze. This section describes the plot display and the objects on it.

Figure 4 shows the display window. The x- and y-axis values represent inches from the origin, the bottom-left of the window. The black rectangle surrounding the maze is the outer wall, while the filled-in black boxes represent additional internal walls. Each of the solid walls are 1 foot by 1 foot in size. Light gray and white squares represent the checkerboard pattern of the maze. Each of these squares is 3 inches by 3 inches.

The block that the robot must pick up is denoted by the yellow translucent square on the maze. In this instance of the simulator, the robot has been defined as a circle, but it can be configured as a rectangle or another shape. In any case, the robot appears as a green translucent polygon. The +X direction on the robot is indicated by a black asterisk. Movement of the robot is displayed as a trail of red asterisks, which updates each time the robot moves.

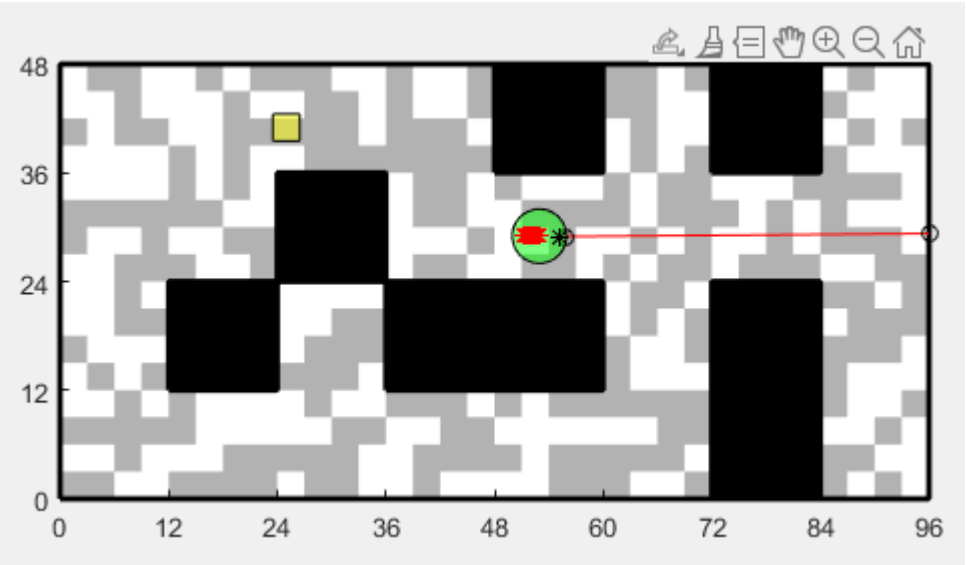

Figure 4: SimMeR main display window.

The red line extending from the front of the robot to the wall in Figure 4 is the "ray" that represents the distance an ultrasonic sensor is measuring. In Figure 5, the closeup of an IR sensor reading is displayed. The blue outer circle is the area of the ground an IR sensor can "see" based on its defined field-of-view (FOV) angle and z-position height above the surface. The red circles show each of the test points within the FOV, and black X's indicate the test points where the checkerboard pattern is black.

5.3 Reference Frames

In SimMeR, all reference frames are designated in a mathematically consistent manner. Looking at the display window shown in 4, the global coordinate system is defined as 0° (X+) to the right, +90° (Y+) in the upward direction, +180° (X-) to the left, and +270° (Y-) in the downward direction. Note that whenever a rotation must be specified, +/- degrees can both be used, as can values greater than 360 degrees. Values greater than 360 degrees or less than 0 degrees may also be output in the case of some sensors, such

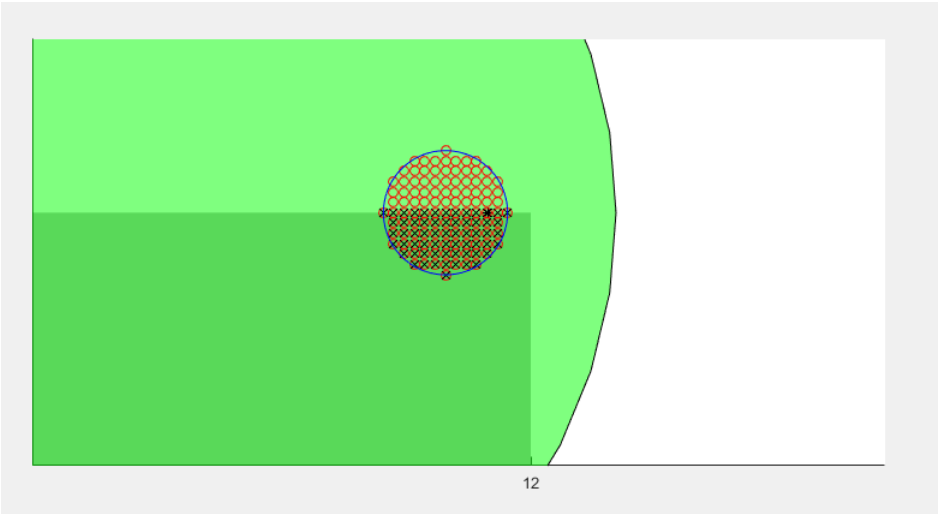

Figure 5: The visualization of an IR sensor reading (zoomed in).

as gyroscopes.

In the robot's local coordinate system, the robot displayed on the plot will always have a small black asterisk on the side defined as X+ (0°).

5.4 Setup Values and Flags

This section details a number of user-editable flags and constants that appear at the beginning of the **Main.m** script. Each of these can be set by the user, and they are the only values that should be edited in a typical configuration.

5.4.1 Setup Values

- *bot_center* - starting location of the robot centroid [x,y] in inches.
- *bot_rot* - starting rotation of the robot, in degrees.
- *block_center* - starting location of the block centroid [x,y] in inches.
- *blocksize* - side length of the block, in inches.

- *num_segments* - the number of segments SimMeR breaks each movement command down into. More segments will generally equate to more representative movement and collision detection, at the cost of computation time. Set to 10 by default.

- *strength* - if random drive bias is enabled, determines the "strength" of the bias, with higher numbers resulting in more bias. Should be defined as a two-element row vector. The first element determines bias strength of directional movements, the second determines rotation bias strength. Set to [0.05, 1] by default.

- *step_time* - Time in seconds to pause between each command executed by the simulator. Primarily used for debugging, if pauses are needed it is better to implement them in the user algorithm if possible. Set to 0 by default.

5.4.2 Flags

- *randerror* - use either true random number generation (1) or consistent random number generation (0). Set to 1 by default.

- *randbias* - use a randomized, normally distributed set of drive biases. Set to 1 by default.

- *sim* - use the simulator (1) or connect to robot via Bluetooth (0). Set to 1 by default, as Bluetooth is not yet implemented.

- *plot_robot* - Enable the main display window (1) or disable it (0). Set to 1 by default.

- *plot_sense* - Enable the plotting of sensor interactions with the maze, such as IR and ultrasonic sensors (1), or disable it (0). Set to 1 by default.

5.5 Communicating with SimMeR

Relevant Files

1. **tcp_setup.m** - creates TCP server/client objects.
2. **tcpclient_write.m** - writes string data to a tcp socket, and waits for a response. Outputs the response as an 8 byte double precision floating point number.
3. **tcpclient_write.m** - reads string data from a tcp socket.

As mentioned in Section 5.1, SimMeR uses TCP sockets to send and receive data between itself and the control algorithm. This section describes that data exchange in more detail.

If a user is programming their algorithm in MATLAB, communication is simple. In the algorithm script at the beginning before the main loop, the following lines of code need to be inserted.

```
[s_cmd, s_rply] = tcp_setup();
fopen(s_cmd);
fopen(s_rply);
```

The first line creates two TCP client sockets, s_cmd and s_rply, on port 9000 and port 9001, respectively. These can be modified by adding the input arguments *role*, *port1*, and *port2*, where *role* is 'client' and *port1* and *port2* are ports of your choosing. Keep in mind that if these values are changed, the values in the simulator file **Main.m** must also be updated accordingly.

The second and third lines open the two ports. Note that this must be done after the TCP server is set up and its ports opened by SimMeR, therefore you must run SimMeR and allow it to reach the stage where it displays *'Simulator initialized... waiting for connection from client'* before running the control algorithm.

To send data to SimMeR, use the following command in Matlab:

```
data = tcpclient_write(cmdstring, s_cmd, s_rply);
```

This function sends data over the socket *s_cmd*, which should be a string of the format [CC-V] (not including the brackets), and waits for a response on socket *s_rply*, which should be an eight byte double precision floating point number.

5.5.1 Data Formats

The first half (CC) MUST be a TWO character command code specified in one of either the **sensors.csv** or **drive.csv** files. If the command code corresponds to a drive direction or rotation, the second half (V) should correspond to the movement value, using as many characters as needed. All commands **MUST** end in a newline character (*newline* in MATLAB or '\10' in many other languages, new in version 0.2). For drive directions, this is distance in inches, for rotations, rotation in degrees. The two halves should always be separated by a hyphen character. To represent negative numbers, use a hyphen before the number, as normal. To represent a number with a decimal value, use a period, again as normal. Note that an easy way to generate strings in the correct format is using the built-in Matlab function **num2str()**.

If the command code corresponds to a sensor, the hyphen and second half (V) can be omitted. A few examples are illustrated below, note that the newline character is not shown but it **MUST** be included at the end of each of these example strings for the simulator to work correctly. Single quotes (') represent information in string format. In this example, 'u1' is the command code for an ultrasonic sensor, 'r1' is the command code for a rotation, and 'd1' is the command code for movement in the +X direction (using the robot local coordinate reference frame).

- 'u1' - polls the ultrasonic sensor. Distance is returned as an eight byte double.

- 'u1-10' - also polls the ultrasonic sensor. The trailing '-10' is ignored since it is not relevant.
- 'd1-3' - tells the robot to move 3 inches in the +X direction
- 'd1-2.4' - tells the robot to move 2.4 inches in the +X direction
- 'r1-45' - tells the robot to rotate +45 degrees (counter-clockwise)
- 'r1–45' - tells the robot to rotate -45 degrees (clockwise, note that there are two hyphens in this code)

Sensor reading commands will always return a single eight byte double as their reply. Movement and rotation commands will return Infinity once completed. If a command code can't be matched to one defined in **sensors.csv** or **drive.csv**, NaN will be returned.

5.6 Running SimMeR

To run a simulation, two programs are required. SimMeR must be run in Matlab, and the control algorithm must be run (if written in Matlab, it must be run in a second instance of Matlab). The order that programs must be run in is shown below. For this example, the sample obstacle avoidance algorithm **test.m** is used.

1. Open an instance of Matlab in which to run SimMeR (Matlab-1).
2. Open a second instance of Matlab in which the control algorithm will be run (Matlab-2).
3. In Matlab-1, navigate to the correct folder (or add it to the Matlab path) and open **Main.m**.
4. Ensure that the config files (Section 4) and Setup variables and Flags in **Main.m** (Section 5.4) are configured correctly.

5. In Matlab-2, navigate to the correct folder (or add it to the Matlab path) and open **test.m**.

6. In Matlab-1, run **Main.m**. Wait for it to display *'Simulator initialized... waiting for connection from client'* in the Command Window.

7. In Matlab-2, run **test.m**.