SimMeR

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

UNIVERSITY OF
TORONTO

Ian Bennett

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

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 Introduction

The program is to be 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, 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 will perform the following tasks as part of it's initial setup.

1. From a pre-defined text file, load the map including wall locations and checkerboard locations.
2. From a user-defined text file, generate a simulated robot including size and sensor loadout and placement.
3. Create 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, the simulator will follow the following steps:

1. Accept TCP packet commands from a student designed algorithm, programmed in MATLAB, Python, or another language.
2. Funnel the packet to either the simulator or a bluetooth serial port (**not yet implemented**) connected to a physical robot
3. Parse the packet to extract the command type.
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 act on the command.

a. For telemetry requests, these are passed to a function that simulates the response data packet of the desired sensor. The function will 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 value and direction and/or the rotation value and direction for the configuration will be specified by the user algorithm, noise will be added based on a user-defined drive noise profile, and used to update the simulated robot's positiong. Several drive noise profiles will be pre-configured by the TAs 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 body and sensors are updated. For integration-based sensors (such as gyroscopes and odometers), their values are updated based on the movement of the robot and noise functions.

2.1 Compatibility

In order to ensure that the user-created programs work with both the simulator and a physical robot, it must take and interpret commands and respond to the environmental stimulus similarly. The input and output software interface should be the same. 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**).

3 Configuration Files

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

1. Define the maze wall locations.
2. Define the maze checkerboard pattern.
3. Define the user's robot size and shape
4. Define the sensor loadout, sensor locations, error parameters, and keybindings.
5. Define the user's robot's drive system, associated error/bias parameters, and keybindings.

3.1 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.

3.2 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 defines the x-coordinates, and the second column defines the y-coordinates, defined in a counter-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).

3.3 Sensors

a. i.e. the string 'w1-x' could correspond to moving forward (w) a number of inches (x), where x is a 16-bit float or similar. The string 'u1' would request data from ultrasonic sensor 1, whose position on the robot is defined by file #3.

3.4 Drive

4 Simulator Definition

This section details the functionality of the simulator and includes descriptions of the system-level functionality as well as descriptions of the functions that make it up.

4.1 State Diagram

The state diagram in Figure 1 shows the system-level process diagram for the robot simulator. Initialization tasks are in black, tasks related to the Bluetooth pipe are in blue, orange indicates tasks related to robot movement commands, and tasks related to querying sensors are in green (sorry for the messy diagram, I'll clean this up at some point). Red writing indicates a configuration file being queried.

4.2 Positioning and Locomotion

4.3 Sensors

This section details the available sensors in the simulator, how each is designed to interact with the simulated environment, and how the error for each is determined.

4.3.1 Sensor Positioning

Each sensor

4.3.2 Ultrasonic Rangefinder

4.3.3 Time of Flight Sensor

4.3.4 Line Following Sensor

4.3.5 Compass

4.3.6 Wheel Encoders

4.3.7 Gyroscope

The rest of this section will be filled in as the program is written.

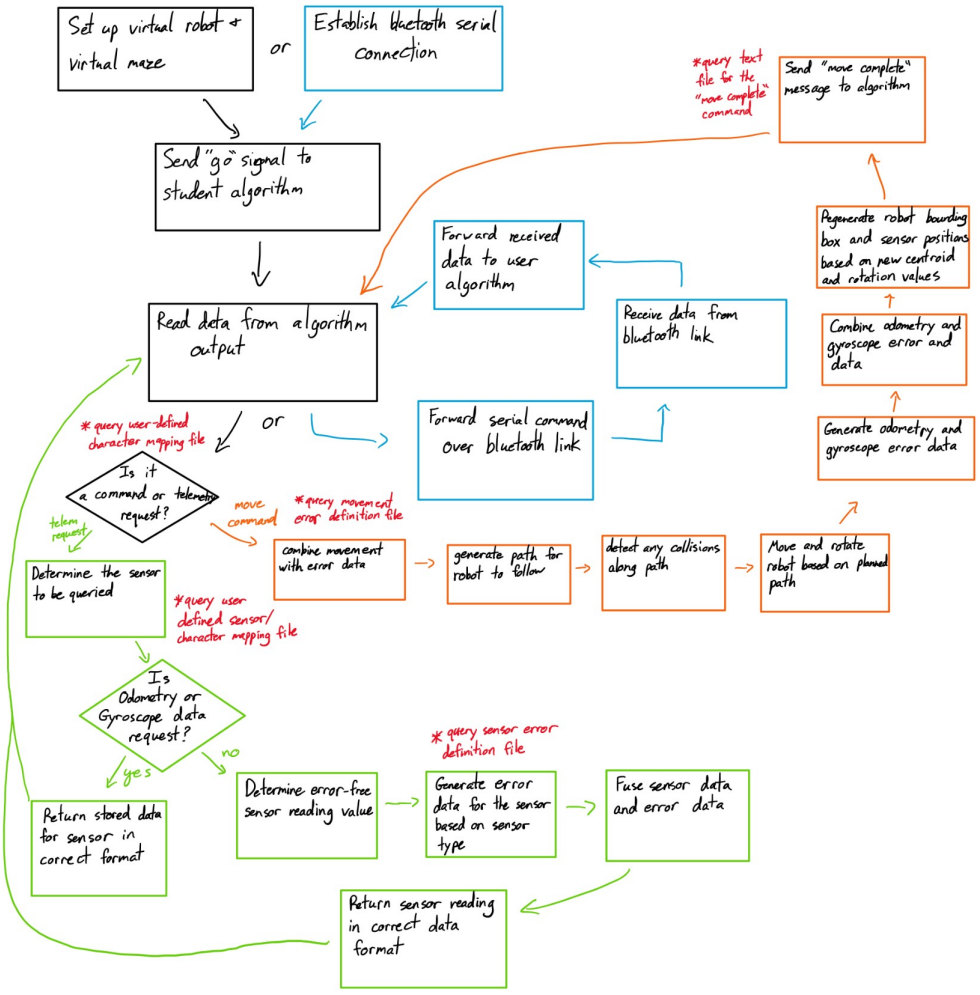

Figure 1: System level state diagram for the robot simulator.

5 Data Formats

This section details the data formats used for input to the simulator from the user's algorithm and output from the simulator to the user's algorithm. Also included is the expected input/output data format for the Bluetooth pipe program to/from the physical robot microcontroller.

This section will be filled in as the program is written.

6 Conclusion

This section will be filled in as the program is written.