

AN INTERACTIVE USER INTERFACE TO THE MOBILITY OBJECT MANAGER FOR RWI ROBOTS

Innocent Okoloko and Huosheng Hu

Department of Computer Science, University of Essex
Colchester Essex C04 3SQ, United Kingdom
Email: ieokok@essex.ac.uk, hhu@essex.ac.uk

Abstract: This paper presents an extension to the Mobility Object Manager (MOM) for the RWI robots, which is currently being developed at the University of Essex. It presents a new user friendly interface extension to MOM for monitoring and interacting with the robot during the development of control algorithms. A new viewer has been incorporated to enable concurrent state reflection of both sensor and actuator data in a single window, with a module that implements interactive trajectory path planning. A motion error model has also been developed to mirror the approximate motion characteristics of the real robot in .simulation. Experiment results are presented to demonstrate the effectiveness of the approach.

Keywords: Mobile robotics, Modelling, Simulation.

1. Introduction

The development of navigation and control algorithms directly on physical robot platforms is expensive in terms of time, effort and safety. It is necessary to build simulated environments as a design tool. Many robot manufacturers deliver a simulator program with their robots, e.g. Saphira simulator from ActivMedia robotics [1] and Kephira simulator from K-Team [2]. There are also multiple robot simulator programs such as Gazebo from Player/Stage [3], [14], which can be plugged unto different types of robots. AvenuUI is a software environment developed at Columbia Robotics Lab for interacting with physical RWI robots.

Most of these platforms enable a 2D or 3D visualisation of the environment that robots operate, and the simulation may be to realise the complete system including physical structure, sensors and actuators, and the environment. Thus algorithms can be developed and tested purely in simulation, then later realised on the real robot. When there is a communication link between the simulated world and the real robot, the simulation can be regarded as hybrid.

The MOM is a software tool developed for the RWI mobile robots [4], which has rather limited functionality available for users. The focus of this paper is to present an extension that is currently being developed for MOM in order to improve the user interface, and then to be able to make profitable use of the robot simulator. In other words, the new extension will present a complete integrated environment in which design, testing and data analysis can be realised on a single software platform. Consequently, useful data can be generated in simulation and used in the real system. This also solves the problem of frequent

relocation and reconfiguration if the real robot is used instead.

Firstly the capabilities of MOM are extended by designing and incorporating new views and additional functionalities. The extensions currently include a new simulator view, which is a user friendly graphical user interface, and a global map viewer window. Secondly the new simulator can make the simulated robot more realistic for mobility users. They have been developed and incorporated to enable users observe and evaluate the motion trajectory of the robot and the position of obstacles in the environment in relation to the robot position as it traverses its environment. A trajectory generation module has also been developed and incorporated, which enables the user to interactively generate trajectory paths and control inputs for running the robot in situations where this is required. Thirdly, a methodology to model the true vehicle motion error is used to calibrate the simulated robot, so as to obtain more accurate results. Finally a new window for graph plotting and data analysis has been designed and integrated with the system graphical user interface for interacting with the robot's sensors and actuators.

The rest of the paper is organized as follows. Section 2 presents a simplified overview of the Mobility™ software architecture. Section 3 gives a brief description of the new Simulator View and added modules. Section 4 describes the vehicle error model and experimental results are presented in section 5. Finally, section 6 presents some conclusions and future work.

2. Mobility Software Architecture

Mobility™ is a distributed, object-oriented toolkit for building control software for single and multi-robot systems [4]. The CORBA 2.X IDL is used to define interfaces which are abstractions for sensors and actuators. The elements of Mobility™ software are shown in Figure 1. It consists of:

- Basic Mobility Tools which are Java User Programs.
- Basic Mobility Objects which are realised in C++.
- A C++ Class Framework (C++ shared lib. & headers)
- Interface definitions (IDL files and shared libraries)
- Support Tools and Utilities (C/C++ Programs)
- An Object Request Broker (basic shared libraries)
- An OS Abstraction Layer (basic shared libraries)

The MOM is the graphical user interface for Mobility™, which enables users to observe, tune, configure and debug the control programs while robot running. It contains a set of object viewers written in java for the visualization of the robot's actuators, sensors, algorithm outputs and debugging information. These viewers communicate with the Mobility™ components via CORBA. The scope of a Mobility system is contained within objects and is accessible from a top-level Naming Service. A simplified diagram of Mobility software system is depicted in Figure 1 and a more comprehensive description can be found in [4]. The next two sections describe extensions that have been developed and added to help in achieving that.

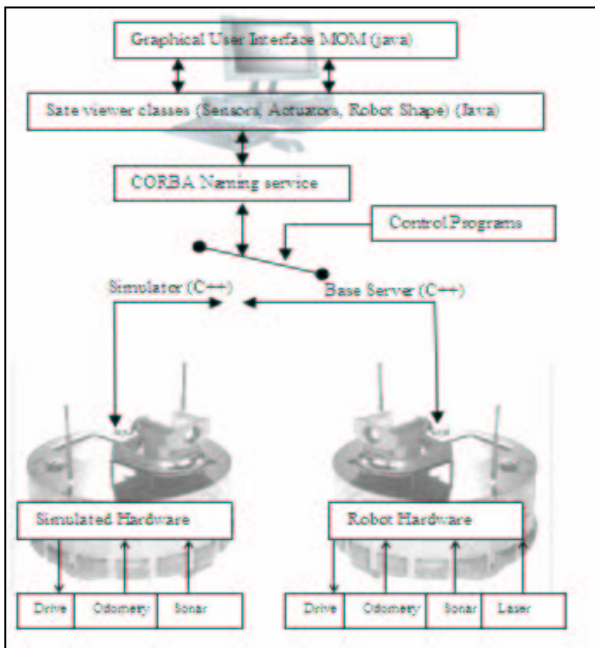


Figure 1 The Elements of Mobility™ robot integration software

3. Simulator View

In the current implementation of MOM, the views for sensor and actuator data are implemented in separate viewer classes that do not interact with each other. As a result the benefit of having a user interface that reflects the entire state of the system is limited. For example the implementation does not allow for observation of the structure of the environment in relation to actual robot poses. This section describes a new simulator view together with its functionalities.

3.1 View Description

In order to provide a more comprehensive interface, we have designed a Simulator View window which contains a model of the robot, a viewer canvas for concurrently observing the motion trajectory of the robot and sensor data, and a facility for manual trajectory generation. Given a priori world knowledge of feature positions in relation to the robot pose, the features can be depicted on the viewing canvas simply by using mouse clicks to paint them on it,

and the robot pose can be reset whenever required. A global map viewer window has also been implemented to enable online view of the complete environment as a map is incrementally built (see Fig. 2). This figure shows the real robot and laser data giving a 2D geometric impression of the immediate environment.

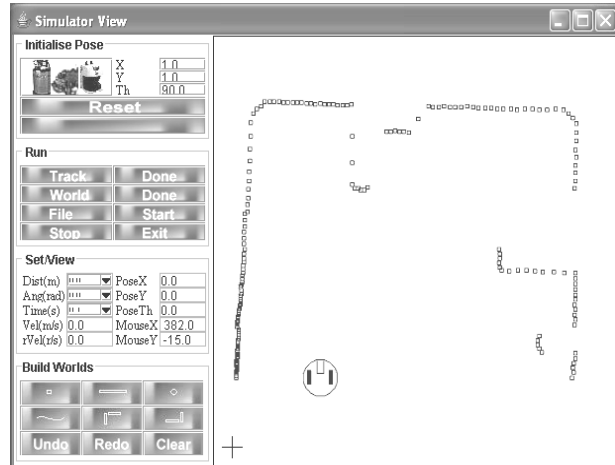


Figure 2 The Simulator View window that may be connected to either a real or a simulated robot.

The main class of the Simulator View window is actually a viewer class for accessing actuator and sensor data concurrently in one view. As the robot moves within the environment, sensor data can be seen in the viewing canvas as shown in Figure 2. The Simulator View is built on two main java classes, namely SimulatorFrame.java and DrawPointRect.java.

DrawPointRect.java: This class implements the viewer canvas and contain six methods:

- **makeBot(), makeWheel() and makeSonar():** are methods to realise a 2D graphical shape of the top view of the robot.
- **getCentre():** is a method that receives actuator data (current robot pose) as inputs. It performs calculation on the y coordinate and the orientation angle in order to convert from world coordinates to the coordinates of the drawing canvas. The resulting data is passed on to the paintComponents method.
- **getObstacle():** is a method that receives two arrays of sensor data (laser or sonar), which hold the x and y values of the current sensor readings. The method performs a negation of the y coordinates in order to convert to the coordinates of the drawing canvas. The result is passed to the paintComponent method.
- **paintComponent():** at every time step, it performs a translation and rotation based on the current pose (x, y, θ). Every item painted on the drawing canvas is affected. This is reflected when the sensor point data and robot are repainted on the viewing canvas, a geometrically meaningful view of the robot and feature positions is thus realized.

SimulatorFrame.java: This class implements the entire Simulator View window; which contains an inner class for implementing a TimerListener. The class creates an instance of the DrawPointRect.java class in order to pass current sensor and actuator parameters to it. In order to obtain sensor and actuator data (robot pose), the required object reference to the interface that implements the functionality is obtained by reading an already registered stringified object reference from a file. Three modules are:

- `getObs()`: it receives new sensor data (sonar or laser) as input and passes it on to the `getObstacle()` method of the `DrawPointRect.java` class.
- `getXY()`: it receives new actuator pose data as input and passes it on to the `getCentre()` method of the `DrawPointRect.java` class.
- `KeyPressed()`: it generates trajectories and control inputs by using keyboard commands. The output is a file containing a set of control inputs.

In a CORBA distributed environment as Mobility™, it should be able to obtain the Object References of all required Objects in order to access multiple Objects from a single program. Since it is necessary to obtain an Object Reference to the Naming Service, the Object Request Broker (ORB) variable in the main class that connects to the Naming Service is declared as public. After connection, every other client class can have an access to the IOR (Interoperable Object Reference) of the Naming Service where required.

Every initial viewer class has been modified so that when the MOM is started the viewer class writes the Object reference of its relevant Object to a file as a string by using the `object_to_string ()` CORBA function, another program can therefore obtain the Object Reference from the file as a string and convert it back to the required Object using the `string_to_object ()` function, the public ORB variable facilitates this method.

In order to ascertain the robustness of the new Simulator View, several algorithms have been run on the real robot with different environment arrangements. Figure 2 depicts the position of the robot in relation to a partial geometric view of its environment. Online map building can be implemented simply by writing the sensor data to file as the robot traverses the environment.

3.2 Trajectory Generation

Basic commands for motor control consist of a set of control inputs $u(t) = [V, \theta]$ sent at each cycle time, where V and θ are translation and rotation velocity commands. The trajectory path followed by the robot is usually made up of line and arc segments. After it is generated, a set of control inputs is required for the robot to execute it [5][6] [7][8].

In our simulator, we assume a scale factor in converting from pixels to unit measurements so that a trajectory path and set of control inputs can be automatically generated by using the keyboard to drive the robot on the viewing canvas in the track generation mode. The set of control inputs generated can then be downloaded and used with an algorithm to run the robot (simulated or real). The generator

is implemented by the `KeyPressed()` module of the `SimulatorFrame.java` class, and a scale factor of 100 pixels to 1 metre is used. Figure 3 depicts a simplified structure of the trajectory generation module.

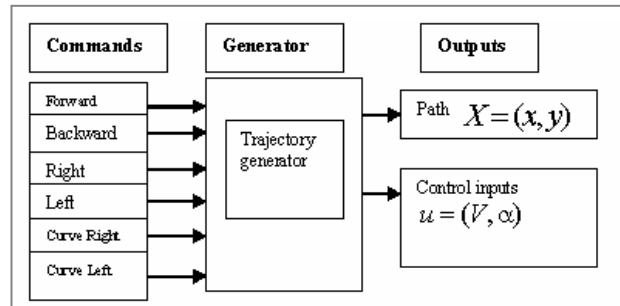


Figure 3 Trajectory generator module. Six keyboard commands generate a trajectory and a set of control inputs

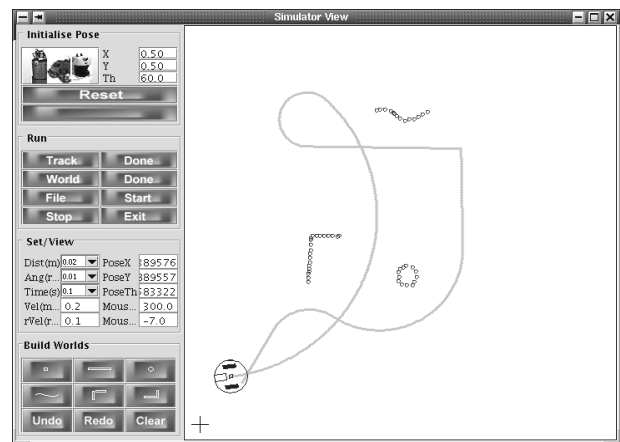


Figure 4 The user interface in the trajectory generator mode.

Figure 4 shows the user interface in the trajectory generation mode. In this mode, the robot model within the drawing canvas is controlled by using the keyboard. As it moves within the canvas it paints the trajectory path along as an indicator of the path traversed. A file containing a set of control inputs is automatically generated. The red spots on the screen are obstacle positions painted by simple mouse clicks, their position in relation to the robot coordinates can be stored in a data structure as temporary world knowledge or recorded in a file as an a priori map. This mode runs without connection to any hardware (simulated or real). After trajectory generation, the position can be reset and the trajectory automatically replayed to test the correctness of the set of control inputs generated.

4. Vehicle Kinematics and Error Model

Simulation attempts to mimic the behaviour of the real robot, we can only ascertain the integrity of the simulation by comparing the motion of the simulated robot with that of the real robot by using the same set of control inputs and algorithm. We have done this using the simulated hardware program from RWI. After executing the required trajectory it is observed that the simulated robot is always almost exactly where it is expected to be.

However, the motion of the real robot grossly differs from the expected position as odometry errors continuously grow with time. The next section describes the method we have adopted to overcome this.

4.1 Kinematics

Assume that the position and orientation of the robot is represented by a vector $X = [x, y, \theta]^T$ at a global frame, as shown in Figure 5. For a differential drive mobile robot, its kinematics model can be described in a discrete form as follows [10] [11],

$$x(k+1) = x(k) + \Delta d(k) \cos(\theta(k) + \Delta\theta(k)) \quad (1)$$

$$y(k+1) = y(k) + \Delta d(k) \sin(\theta(k) + \Delta\theta(k)) \quad (2)$$

$$\theta(k+1) = \theta(k) + \Delta\theta(k) \quad (3)$$

$$\Delta d(k) = v_0 \times t, \quad v_0 = (v_r - v_l)/2, \quad \Delta\theta = (v_r - v_l)/W \quad (4)$$

where v_l and v_r are the left and right wheel velocities, and W is the width of the wheelbase.

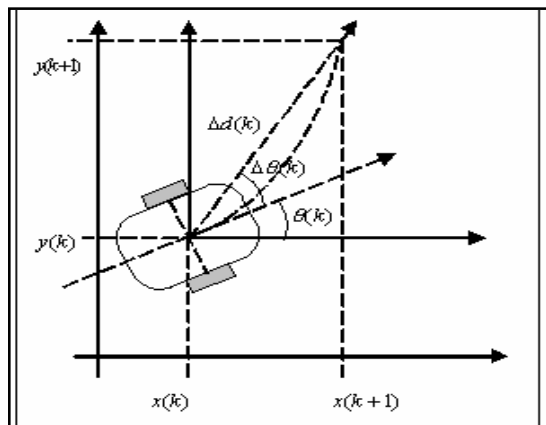


Figure 5 The mobile robot in a global frame

Given a set of control inputs required to execute a task, after execution the robot stops at a new position, the position error can be categorized as follows:

- The difference between the *actual* and *expected* position and orientation of the robot.
- The difference between the *actual* position and orientation of the robot and that *recorded* from odometry (using internal sensor data and the above equations).
- The difference between the *expected* position and orientation and that recorded from odometry.

4.2 Robot Error Model

The robot error models are dependent on individual mobile robots, and have to be empirically determined for each robot. The robot error may result from *systematic errors* (errors in mechanical design and internal components) which can be estimated and calibrated out, or *random errors* which can not be calibrated out but normally be modelled as Gaussian vehicle motion noise. Many error calibration techniques have been developed so far [9], [13]. To make the motion of the simulated robot resemble that of the real robot we have concentrated on modelling the

systematic error as it seems to have a more damaging effect on robot motion. Random error modelling can be found in [13].

To observe the motion characteristics of the real robot, experiments were performed by repeatedly commanding the robot to move a certain distance along a straight line trajectory. The difference between the desired trajectory and that of the real robot consistently resembles the plots depicted in Figure 6 and the data in Table 1. The repeated experiments imply that the odometry reading of the robot is precise for short distances even though the robot is far from expected position. This is most likely resulting from systematic error which can be determined and represented by a set of parameter values. Our task in modelling the real robot in simulation is therefore to transfer the error parameters empirically determined to the simulated robot.

Kalman filters require robot motion noise to be added to the pure state transition equations (1) - (3). In our case, we have not modelled random errors rather the systematic errors are incorporated to the simulated robot model, because it is what makes the robot's physical location to differ grossly from its expected position.

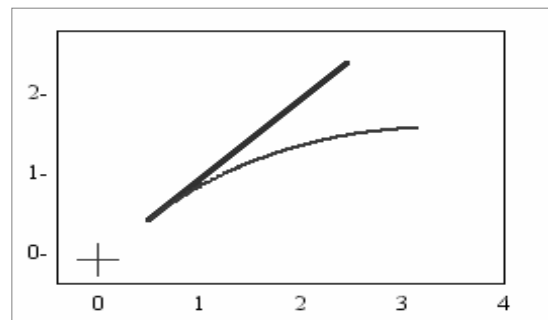


Figure 6 Position error in the real robot, the straight line indicates the desired trajectory, while the curve indicates the actual trajectory recorded from the odometry.

Table 1. The motion error of the real robot

	Desired	Actual	Error
x (m)	2.4728	3.2081	0.7353
y(m)	2.4728	1.4842	0.9886
θ (°)	45	354.57	-50.429

Equations (1) to (3) can be replaced with the following equation

$$x(k+1) = f((x(k), u(k) + v(k)) = \begin{bmatrix} x(k) + \Delta d(k) \cos(\theta(k) + \Delta\theta(k)) \\ y(k) + \Delta d(k) \sin(\theta(k) + \Delta\theta(k)) \\ \theta(k) + \Delta\theta(k) \end{bmatrix} + v(k) \quad (5)$$

where $v(k)$ is a column vector $v[x_e, y_e, \theta_e]^T$

The curved plot in Figure 6 indicates that the drift error is nonlinear and most likely resulting from small changes in orientation which is included in calculating the x and y positions for the next time step. To prove this, a simple error feedback loop was implemented on the real robot using orientation error alone in the feedback, the result

obtained from using the same set of control inputs in running the robot and the simulator without error model is depicted in Figure 7.

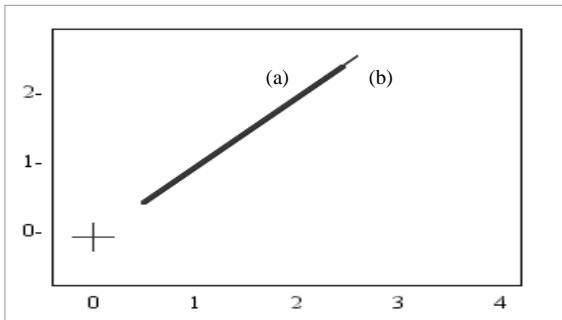


Figure 7 Using orientation error feedback on the real robot. The solid plot (a) is desired trajectory, while (b) is actual trajectory.

For straight line motion the average orientation error was empirically found to be -0.00527 radians for a distance of $0.02m$ (-0.2635rads/m), position error for a distance of x_m is a factor of 0.065 . For a curvature trajectory the orientation error is reduced to (-0.02635rads/m) , while position error for a distance of x_m is a factor of 0.065 . Clearly the error is not only nonlinear but also non Gaussian as it is more pronounced while the robot is executing a straight line trajectory rather than executing a continuous curvature path. To realize the model in simulation, rather than making the error simply additive, we express it as motion noise as a function of the current set of control inputs, thus we can rewrite Equation (5) as

$$x(k+1) = f((x(k), u(k) + v(k))) =$$

$$\begin{cases} x(k) + (\Delta d(k) + \Delta d(k) * 0.0655) * \cos(\theta(k) + \Delta\theta(k)) \\ y(k) + (\Delta d(k) + \Delta d(k) * 0.0655) * \sin(\theta(k) + \Delta\theta(k)) \\ \theta(k) + (\Delta\theta(k) + \Delta d(k) * -0.02635) \end{cases} \quad \Delta\theta < 0$$

$$\begin{cases} x(k) + (\Delta d(k) + \Delta d(k) * 0.0655) * \cos(\theta(k) + \Delta\theta(k)) \\ y(k) + (\Delta d(k) + \Delta d(k) * 0.0655) * \sin(\theta(k) + \Delta\theta(k)) \\ \theta(k) + (\Delta\theta(k) + \Delta d(k) * -0.2635) \end{cases} \quad \Delta\theta = 0$$

$$\begin{cases} x(k) + (\Delta d(k) + \Delta d(k) * 0.0655) * \cos(\theta(k) + \Delta\theta(k)) \\ y(k) + (\Delta d(k) + \Delta d(k) * 0.0655) * \sin(\theta(k) + \Delta\theta(k)) \\ \theta(k) + (\Delta\theta(k) + \Delta d(k) * 0.02635) \end{cases} \quad \Delta\theta > 0$$

5. Experimental Results

To evaluate the usefulness of the extensions that have been developed for MOM, each of the developed modules have been tested, using both the real and simulated robot where required. Results from experiments performed to test the effectiveness of using the vehicle motion error model in simulation are presented below.

5.1 Vehicle Error Model Tests

To test the vehicle error model, a trajectory path together with a set of control inputs for executing it is automatically generated using the trajectory generator. The resulting set

of control inputs is downloaded to a control program that reads it from a file and sends the sequence of control inputs to the robot's actuators. The robot stops after the sequence is completed.

First the simulated robot is run without and with error models, and results are collected. Then the real robot is made to run on this same set of control inputs. The trajectory followed by the robots in all tests are recorded for each run and compared by plotting the data sets in a graph. Several experiments have been performed in order to test the closeness of motion of the simulated robot with that of the real robot; two of the results are presented here.

Straight Line Trajectory

First the previous experiment with output depicted in Figure 7 is repeated. The initial pose of the robot (in the drawing canvas) is set at $(0.5, 0.5, 45)$, then using the keyboard, the robot is commanded to follow a straight line trajectory for an approximate distance of $2.5m$. The simulated robot (with and without models) and the real robot are then made to run with the set of control inputs at 100milliseconds time step. The results from odometry recorded for all three tests are presented in Figure 8.

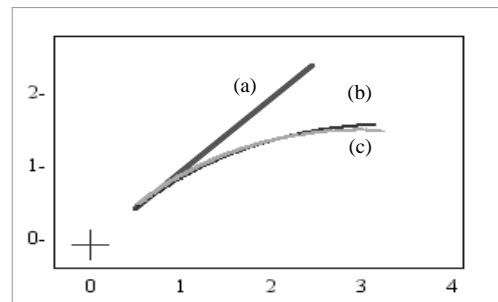


Figure 8 Comparison of trajectory for experiment 1. Plot (a) indicates the desired trajectory, followed by the simulated robot without error model, plot (c) indicates the trajectory simulated robot with error models, while plot (b) is the trajectory followed by the real robot

	Desired	Simulated	Simulated (model)	Real Robot
x (m)	2.4728	2.4728	3.0663	3.2081
y(m)	2.4728	2.4728	1.2849	1.4842
θ ($^\circ$)	45	45	349.611	354.571

Table 2 Comparison of stopping points for simulated and real robot, for moving an expected distance of $2.5m$.

Curvature Trajectory

In the second test, the robot is commanded to follow two more complex trajectories, one consisting of a connection of lines and arcs and the other a set of connected straight lines, the results obtained are presented in Figures 9 and 10 with data in Tables 3 and 4. As can be observed, the motion error is worse for straight line trajectories (Figure 10) than for curvatures (Figure 9). However the motion of the model enhanced simulated robot resembles that of the real robot, at an approximation.

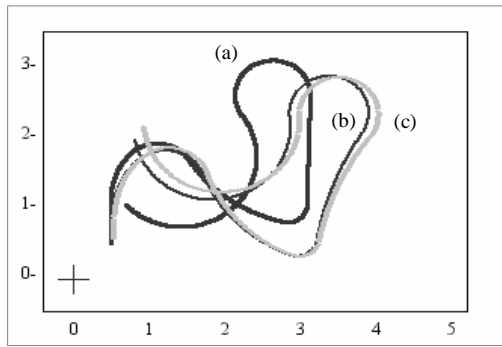


Figure 9 Plot (a) indicates the desired trajectory generated, and it is followed by the simulated robot without the error model; plot (b) is the trajectory obtained from the real robot; plot (c) is the trajectory followed by the simulated robot with error model.

	Desired	Simulated	Simulated (model)	Real Robot
x (m)	0.7115	0.711	0.910	0.832
y(m)	1.07132	1.07	2.130	1.992
θ (r)	2.3408	2.34	1.675	1.763

Table 3 Comparison of stopping points of simulated and real robot for a non linear trajectory.

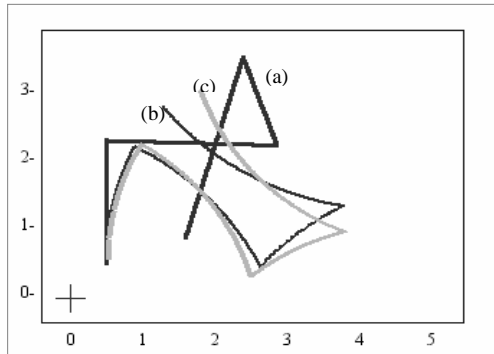


Figure 10 Plot (a) indicates the desired trajectory generated, and it is followed by the simulated robot without the error model; plot (b) is the trajectory obtained from the real robot; plot (c) is the trajectory followed by the simulated robot with error model.

	Desired	Simulated	Simulated (model)	Real Robot
x (m)	1.6066	1.61	1.79	1.35
y (m)	0.9041	0.91	3.0	2.83
θ (r)	4.4208	4.42	1.92	2.25

Table 4 Comparison of stopping points of simulated and real robot for mixed linear trajectory.

6. Conclusions and Future Work

In this paper an extension to the MOM is presented and it consists of a new Simulator View and a Global Map viewer. The new features enable concurrent state reflection of sensor and actuator data and can facilitate online map building, which has been tested and found to be very robust. A trajectory planner module is implemented, and a methodology for improving the performance of the simulated hardware by incorporating the error model of the real robot is developed and tested. Two new functionalities

that have also been added to the GUI are on-screen featuring indication (map building) by mouse clicks, and graphing windows for data analysis. With these added functionalities the simulated hardware program has become more realistic; the simulated environment will be very useful to mobility users who are interested in speeding up the development of control algorithms with less cost, time effort, more safety and accessibility.

In future a simulated laser sensor will be incorporated into the simulated program, and the viewing canvas will be replaced with a 3D structure so that environmental features can be visualised and represented using 3D maps. Because vehicle motion errors differs for each robot, a facility for interactively incorporating the error models of the real robot in the simulator will be developed and added to the user interface.

Acknowledgement: Thanks to Joshua Knox at RWI for providing the source codes for MOM and the *simulated hardware* program, the starting point for this work.

References

- [1]. Pioneer 2 Operations Manual: Copyright © 2000, ActivMEDIA ROBOTICS,
- [2]. Khepera Simulator: <http://www.k-team.com/robots/khepera/>
- [3]. Gazebo 3D Multi-robot Simulator, <http://playerstage.sourceforge.net>
- [4]. Mobility Integration Software Users Guide: <http://resources.rwii.com>
- [5]. T. J. Graettinger and B. H. Krogh, Evaluation and time-scaling of trajectories for wheeled mobile robots, *Proc. American Control Conf.*, Atlanta, GA, June 1998.
- [6]. W. L. Nelson, Continuous-curvature paths using. quintic and polar polynomials, *AT&T Bell Laboratories, internal document*, Mar. 30 1998.
- [7]. Y. Kanayama and N. Miyake, Trajectory generation for mobile robots, *Proc. 3rd Int. Symposium on Robotics Research*, O. D Faugeras and G. Giralt, eds. Gouvioux, France, pp. 333-340, 1985.
- [8]. Y. Kanayama and B. Hartman, Smooth local path planning for autonomous vehicles, *Univ. of California, Santa Barbara, Tech. Report TRCS88-15*, 1998.
- [9]. Kurt Konolige, Notes on Robot Motion- Probabilistic Model; Sampling and Gaussian Implementations; Markov Localization, September 2001.
- [10]. J. J. Leonard, *Directed Sonar Sensing for Mobile Robot Navigation*, PhD Thesis at the University of Oxford 1990.
- [11]. K. Kobayashi, F. Muneakata and K. Watanabe, Accurate Navigation via Differential GPS and Vehicle Local Sensors, *Proc. IEEE Int. Conf. on Multi-sensor Fusion & Integration for Intelligent System*, Las Vegas, Oct. 2, 1994.
- [12]. J. Borenstein and L. Feng, UMBmark-A Method for Measuring, Comparing, and Correcting Dead-reckoning Errors in Mobile Robots, *University of Michigan Technical Report UM-MEAM-94-22* December 2004.
- [13]. I. Okoloko, *Fusion of Sonar and Laser Data for Autonomous Mobile Robot Navigation*, MSc project at the Dept. of Computer Science, Essex University, Sept. 2002.
- [14]. B.P. Gerkey, Richard T. Vaughan, Andrew Howard, The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems, *Proc. of Int. Conf. on Advanced Robotics pages 317-323*, Coimbra, Portugal, June 30 - July 3, 2003.