ISR TECHNICAL REPORT 2008-18

Wall Following with Collision Avoidance and Mapping Using a Laser Range Finder

Finkelstein, Mark; Hunte, Brian,
Advisors: Krishnaprasad, P.S., Galloway, Kevin, Mischiati, Matteo, and Twu, Phillip

Institute for Systems Research
Intelligent Servosystems Laboratory

Wall Following with Collision Avoidance and Mapping Using a Laser Range Finder

Project by

Mark Finkelstein
University of Maryland

Brian Hunte
Syracuse University

Project Advisor

Professor P.S. Krishnaprasad
University of Maryland

Graduate Students

Kevin Galloway
University of Maryland

Matteo Mischiati
University of Maryland

Phillip Twu
Georgia Institute of Technology

Table of Contents

Abstract

In this project, various approaches were analyzed as possible approaches towards wall following and mapping with implementations of select approaches designed in MDLe for the Activ Media robots provided. The laser rangefinder was utilized extensively for mapping and odometry and the indoor cricket system were used for localization. The robot has come to demonstrate good wall following ability in rather unpredictable circumstances, but the quality of the mapping can be irregular due to the poor quality of the odometry.

Introduction

Robot

Our work revolved around the Activ Media Pioneer 2-AT, which is a 4-wheel robot with a computer on board utilizing a Debian distribution. Debian is a multipurpose open source computer operating system. Development of code and other modifications to the robot's behavior were executed by connecting directly to the robot's onboard computer through an SSH client. The combination of MDLe, ME, meesh, and CORBA allows for the execution of command code written in C++ on the robot. These utilities are elaborated upon in a later discussion on programming the robot.

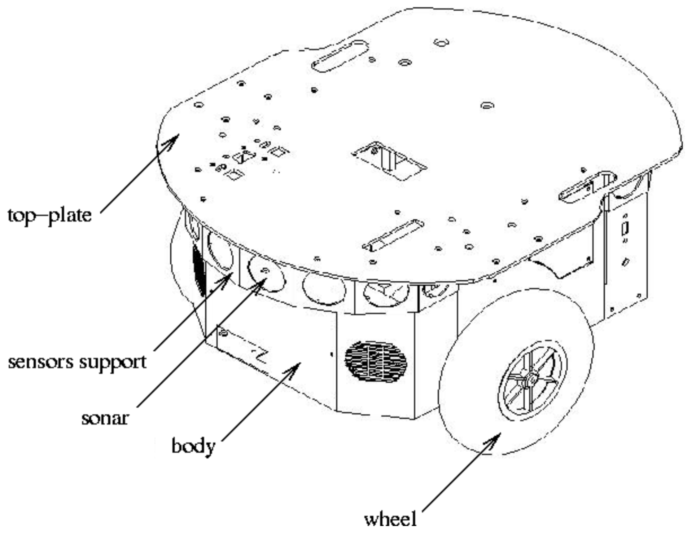

Figure 1: Pioneer2 Robot by ActivMedia (photo courtesy of http://www.cyberbotics.com)

Autonomy

Autonomy within the scope of this paper would be defined as the robot's ability to react and perform in an unpredictable environment without human intervention or guidance. Through the use of implemented programs run from the onboard computer, the robot is able to independently, at runtime, make decisions that allow it to complete its assigned task or obtain the required data.

Sensors

Laser

The laser rangefinder uses a laser beam to determine the distance between it and a reflective object. When the laser is operating, this data is then made available to programs, which use the data as sensory data to understand the surrounding environment. The laser can be used for mapping and identification of objects based on object shape.

Sonar

The onboard sonar sensors use ultrasonic sound to determine the presence of obstacles directly in front of the robot. This information could be used for the avoidance of objects in the immediate vicinity of the robot.

Bumper

The bumpers detect objects that push one of the many arrays of bumpers on the robot. Working as electromechanical devices, bumpers can be utilized for error conditions or purposely built to react to situations when the robot collides with an object.

Localization

Localization is simply the term for establishing the position of a particle, or a robot modeled as a particle, in space. There are a few different techniques that can be used to determine the robot's location with respect to the lab coordinate system. While some localization methods exploit external sensors, like the Cricket indoor position system, odometry is accomplished by calculations done onboard the vehicle itself. Both methods, however, have certain strengths and weaknesses inherent to them.

Cricket

Cricket is a MIT developed positioning system designed to mimic the Global Positioning System (GPS) in an indoor environment or in urban areas where GPS is not reliable. Certain Cricket devices are designated as beacons, which have a fixed location on the ceiling and transmit concurrent RF and ultrasonic pulses. The client is then designated as a "listener." When it hears the RF signal, it listens for the following ultrasonic pulse and is able to determine distance from the difference in arrival times of the RF and ultrasonic pulses. Once the client's distances from three beacons are known, the cricket system automatically triangulates the position.

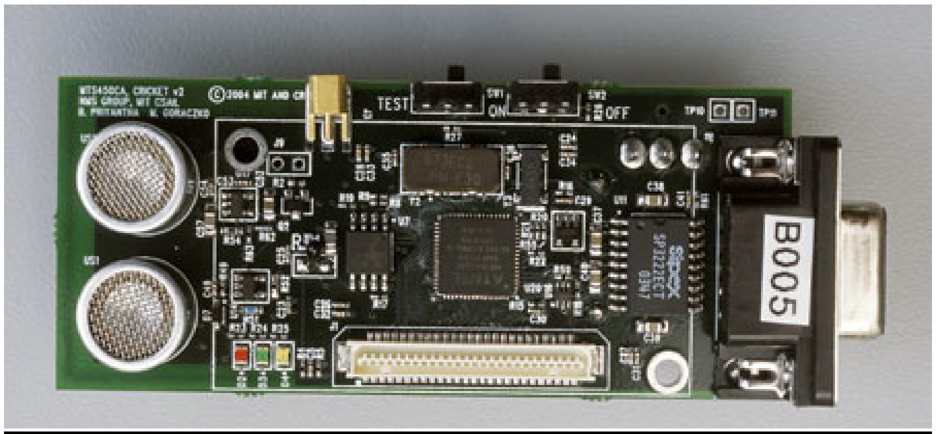

Figure 2: Cricket Sensor (courtesy of http://cricket.csail.mit.edu/)

The Cricket system used here has been modified to fit the needs of control applications in the ISL. For example, two "listeners" are used, one on the front and one on the back of the robot. Since the separation is known, this allows for us to determine the orientation of the robot as well as the absolute *x* and *y* coordinates. Also, one cricket unit is designated to be the master unit which synchronizes all other units so that they are all operating at the same time on a 2-3 second interval [5].

During extensive work with the cricket (Appendix Fig. A1-A4) we found them to be highly accurate, though we found an inherent offset by noting a sinusoidal pattern for both x and y values over time when the robot was told to rotate in place. (Appendix Fig. A2-A3). The cricket also allowed us to observe the rate of slippage in both X and Y directions, while turning in place. Figures A2 and A3 show us that the rate of slip in the X direction was approximately 2.4 centimeters per rotation. The rate of slip in the Y direction, however, was only 0.1 centimeters per rotation. These rates were fairly consistent at angular speeds of 0.1 radians per second and 0.3 radians per second.

Odometry

An odometry module is included by ActivMedia on the Pioneer2. This module uses dead reckoning as the means of determining relative position and orientation. As long as the speed, elapsed time and heading direction are known, it is possible to estimate position with respect to an initial reference point. Dead reckoning is used in inertial navigation systems as well. Since the odometry module for the Pioneer2 was not working correctly, we developed our own open-loop odometry.

Cricket and Odometry Comparison

Cricket and odometry are two vastly different methods of localization. Each method comes with certain strengths and limitations. For example, odometry is only ideal over short distances, as it is easy for small errors to accumulate over time, making long journeys very difficult to trace accurately. On the other hand, the error in Cricket estimates is predictable as a Gaussian Function with a standard deviation of just a couple centimeters. One limitation of Cricket is the fact that its readings are taken every 2-3 seconds [5]. Therefore, the speed of the unit being tracked must be low enough and the sensors provide enough samples to avoid the problem of aliasing. The most obvious limitation of Cricket however, is that it takes time for installation and programming of the sensors and there is a financial cost to obtaining them. It isn't practical to cover the entire ceiling of a building with Cricket beacons.

Theoretical Background

In order to understand how the odometry works, one must understand the use of reference frames and the parameterization of curves in space. A generic curve in space can be viewed as a time trajectory, in which the position on the curve is changing with respect to time. This tends to be the most common way that we think of the path of a moving particle. For example, when a baseball player hits the ball, the fielder quickly senses where the ball is going and makes two estimations. He must gauge how much time until the ball reaches a certain location in space and also estimate how long it will take for him to get to that position. If the time to reach the ball is insufficient the player may decide to let it fall or may even risk a diving attempt. First let us consider a curve parameterized by time.

Consider a regular curve γ on the interval of *a* to *b* in three-dimensional space. That is,

$$\vec{\gamma}(t):[a,b]\rightarrow\mathbb{R}^3 \qquad (1)$$

The curve is said to be regular if

$$\dot{\vec{\gamma}}\neq 0\ \forall t\in[a,b] \qquad (2)$$

Let us say that *a*=0 and *b*=T. Then, the distance *(s)* traveled during time *(T)* is

$$s(t)=\int_0^T V(t)dt \qquad (3)$$

Where speed, $\frac{ds(t)}{dt}=V(t)$ (4)

Now if we consider the curve parameterized by arc length, *t*=*t*(*s*) and therefore $\vec{\gamma}=\vec{\gamma}(s)$.

Using the chain rule, we can relate $\vec{\dot{\gamma}}(t)$ to $\vec{\dot{\gamma}}(s)$.

$$\frac{d\vec{\gamma}(t)}{dt} = \frac{d\vec{\gamma}}{ds}\frac{ds}{dt} \qquad (5)$$

Simplifying, yields

$$\vec{\gamma}'(s) = \frac{\vec{\dot{\gamma}}(t)}{V(t)} \qquad (6)$$

Taking the dot product of $\vec{\dot{\gamma}}(t)$ with itself from equation (6) gives the important relation

$$\vec{\gamma}' \bullet \vec{\gamma}' \equiv 1 \qquad (7)$$

Differentiating both sides with respect to *s*, we find that

$$\vec{\gamma}'' \bullet \vec{\gamma}' \equiv 0 \qquad (8)$$

Therefore, adopting the notation $\mathrm{T} = \gamma'$, we know that T is perpendicular to T' for all values of *s*. If we view T' as a vector in a plane perpendicular to the tangent vector T (see Figure 3), we can assume that any point in that plane is a linear combination of the two unit vectors that we denote by $\hat{M}_1$ and $\hat{M}_2$. Keep in mind that T is a unit vector as well. Expressed mathematically,

$$\vec{\mathrm{T}}'(s) = u(s) \cdot \hat{M}_1(s) + v(s) \cdot \hat{M}_2(s) \qquad (9)$$

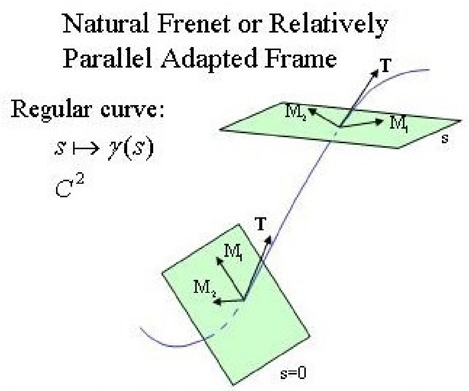

Figure 3: Moving Reference Frame (courtesy of Dr. Krishnaprasad [3])

Consider $\left[\mathrm{T}(s), M_1(s), M_2(s)\right]$ as a reference frame that is moving along the curve as *s* increases. In order for the moving frame to be chosen consistently, we add two more equations. These equations are equations of curvature that are scaled by natural curvatures *u* and *v*.

$$M_1'(s) = -u(s) \cdot \mathrm{T}(s) \qquad (10)$$

$$M_2'(s) = -v(s) \cdot \mathrm{T}(s) \qquad (11)$$

In matrix form, the system of equations including equations (9), (10), and (11) is written

$$\left[\mathrm{T}(s) \quad M_1(s) \quad M_2(s)\right]' = \left[\mathrm{T}(s) \quad M_1(s) \quad M_2(s)\right] \begin{bmatrix} 0 & -u(s) & -v(s) \\ u(s) & 0 & 0 \\ v(s) & 0 & 0 \end{bmatrix} \qquad (12)$$

Finally we can relate this system of equations back to the robot to model its motion. It is appropriate to assume that the wheeled robot's movement will be restricted to two dimensions. Therefore we know that a change in curvature in the component perpendicular to the plane cannot occur throughout the course (M_2'=0 since v=0). Therefore, equation (12) reduces to

$$\left[\mathrm{T}(s) \quad M_1(s)\right]' = \left[\mathrm{T}(s) \quad M_1(s)\right] \begin{bmatrix} 0 & -u(s) \\ u(s) & 0 \end{bmatrix} \qquad (13)$$

Therefore, we can draw the reference components as is shown in Figure 4. Now we overlay an absolute coordinate system that is Cartesian, defining theta as the angle counterclockwise from the positive x direction. Resorting to trigonometry,

$$\mathrm{T}(s) = \begin{bmatrix} \cos\theta(s) \\ \sin\theta(s) \end{bmatrix} \qquad (14)$$

$$M_1(s) = \begin{bmatrix} -\sin\theta(s) \\ \cos\theta(s) \end{bmatrix} \qquad (15)$$

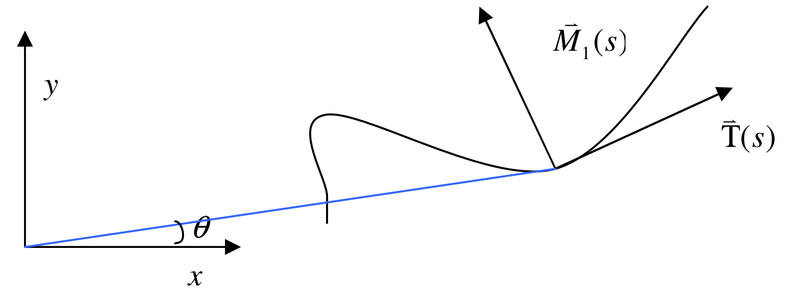

Figure 4: Two Dimensional Reference Frame Relation to Absolute Frame

Substituting equations (14) and (15) into equation (13) one discovers that $\theta'(s) = u(s)$. Therefore the governing differential equations expressed in absolute coordinates with respect to the arc length parameter *s* are

$$\begin{aligned} \gamma_x'(s) &= \cos\theta(s) \\ \gamma_y'(s) &= \sin\theta(s) \\ \theta'(s) &= u(s) \end{aligned} \qquad (16)$$

And with respect to time

$$\begin{aligned} \dot{x} &= V(t)\cos\theta(t) \\ \dot{y} &= V(t)\sin\theta(t) \\ \dot{\theta} &= V(t)u(t) = \omega(t) \end{aligned} \qquad (17)$$

Where V(t) is the robot speed, and ω is its angular velocity.

Programming Aspect

MDLe

Definition:

The Extended Motion Description Language (MDLe) was developed at the Intelligent Servosystems Lab with the goal of utilizing high level abstractions that can be extended to other devices by simply changing the definition of the motion primitives without changing the high level abstraction thereby separating the control theory from the particulars of each command [2].

Structure:

The basic building blocks of MDLe programs are quarks. Quarks have specific functions and can be divided into two categories: they either perform an action or interrupt an action. The quarks themselves are coded in C++, though a newer implementation of the MDLe framework utilizes Java. These quarks are combined into a plan, which is the highest level of abstraction of a control algorithm in MDLe.

The execution of MDLe scripts involves giving each step, which is usually made up of an action quark with accompanying interrupt quarks, a time slice within a turn. The turn is made up of these time slices and the platform’s scheduler, in this case Debian, decides the amount of time to appropriate each step.

Within the script, there is a certain format that must be adhered. The first argument is used to specify how many iterations of the plan are to be expected. A plan level interrupt quark may follow this, which will stop the current iteration if it returns a 0 instead of a 1. The atoms follow this and once one is interrupted by its interrupt quark, it is followed by the next atom in the script, if one exists. Each atom consists of an interrupt quark paired with an action quark. Interrupt quarks have binary return values with a 1 signifying the interrupt condition has not been met and a 0 signifying that the condition has been met. Once the interrupt quark returns a 0, the next atom, if it exists, will be executed. Action quarks also return binary values, with a 1 signifying the successful completion of a task and a 0 signifying the opposite. When a 0 is returned by the action quark, the action quark will continue to be executed.

Below is an example of a script in MDLe called "`cricketObAvoid`" with an accompanying explanation:

```
#!/bin/sh
meesh $MEHOST << __EOF__

m.import cricket.me
m.alloc /lib/cricket.me/Cricket /usr/cricket
cd usr/cricket
./port_num {1}
cd ../..
m.start cricket

m.alloc /lib/cricket.me/Cricket /usr/cricket2
cd usr/cricket2
./port_num {2}
cd ../..
m.start cricket2

cd usr/mdle
./load_plan { cricketObAvoid = (ExecPlan -1 (cricketEnd 0
150) (Atom (obDetect) (cricketMove 0 150)) (Atom
(noObDetect) (avoidGo))); }
./main { plans/ cricketObAvoid }
cd /
__EOF__
```

In every script, the ME shell is called first. ME, short for Modular Engine, is the link between software and system components. In ME, each system component, from the crickets to odometry, is encapsulated as modules. These modules are appropriated time to run based on the platform's distribution of time. The set of loaded modules necessarily includes the MDLe module, which itself is subdivided into steps. The "`meesh $MEHOST << __EOF__`" command simply states that all the text until "`__EOF__`" will be directed towards ME shell (`meesh`), which means that scripts are technically run in the ME shell. The "`m.import`" command makes a module active. This is followed by the command "`./port_num`", which defines a port number for the cricket unit. This is done for the first cricket and repeated for the second cricket. A plan is then loaded under the name "`cricketObAvoid`." Breaking down the plan, "`ExecPlan -1`" tells the interpreter how many times the plan should be executed, with -1 indicating that it should never stop, unless the plan level interrupt is fulfilled. This is followed by the plan level interrupt such as "`cricketEnd 0 150`", which states that the plan should stop when it reaches within a certain tolerance, defined in the program, the point (0, 150). The "`Atom`" annotation that follows indicates that the basic atom building block built of quarks is to follow. In this particular case, "`obDetect`" is the interrupt quark and "`cricketMove 0 150`" is the action quark, conforming to the set format of interrupt quark first followed by an action quark. This atom causes the robot to move towards the point (0, 150) unless an object is detected, in which case it goes to the next atom. The next atom's interrupt quark is "`noObDetect`" and the accompanying action quark is "`avoidGo`." This atom runs "`avoidGo`" until "`noObDetect`" returns 0, which means that it attempts to get into a position where it no longer detects an object in front of it, at which point it goes to the next atom, which is, incidentally, the previous atom. This goes on until the plan level interrupt "`cricketEnd 0 150`" returns 0, which indicates arrival at the point (0, 150). Running the "`./main`" command runs the plan and makes the robot reach the point (0, 150) [2].

Wall Following Algorithm

There are three main parts to the algorithm that attempt to deal with the constraints given by the robot and the additional constraint of maintaining a constant velocity unless absolutely necessary to do otherwise. The speed is a constant that is defined in the MDLe script.

The most basic and crucial part is the part responsible for wall following. This part utilizes two readings, both on the side of the wall. Note the triangle below:

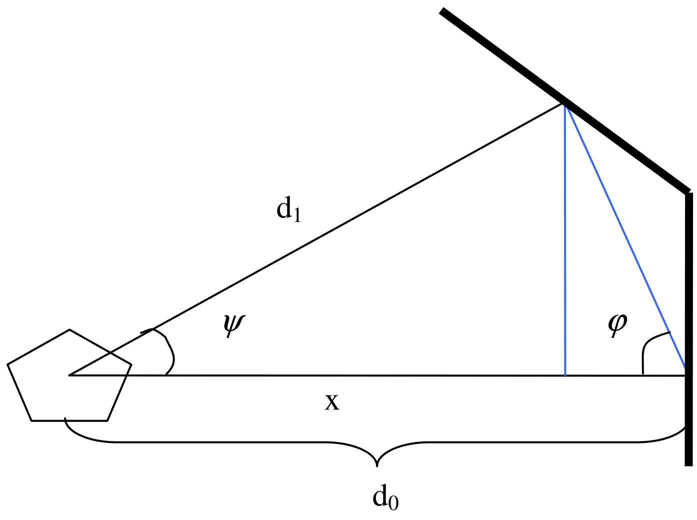

Figure 5: Model of wall following strategy

In Figure 5, let x be equal to $d_1\cos(\psi)$, where ψ is a known value that exists as the angle between readings. This would mean that when x is greater than d_0 then the wall tilts away from the robot and when x is less than d_0 then the wall tilts towards the robot. The goal can be seen as either getting phi to 90 degrees or x to d_0. The available values from the laser rangefinder readings would be ψ, d_0 and d_1. Working from these values and x, the calculation of which was previously described, the formula for ϕ can be determined to be $\tan^{-1}(d_1*\sin(\psi)/(x))$. Using a constant *r*, which is set in the MDLe script to indicate the rotational speed constant, the formula $R = r*(1-|\phi|/(\pi/2))*c_1$, where *R* is the rotational speed of the robot and c_1 is a proportionality constant used to moderate the speed. The desired distance from the wall is also a factor that is considered. Using a constant *d*, which is set in the body of the quark code, the general formula for the rotational speed, *R*, is extended to $R = r*(1-|\phi|/(\pi/2))*c_1 + (1- d_0/d)* c_2$, where c_2 is a proportionality constant used to moderate the speed.

The second major part involves determining which wall to follow. In the first iteration, the robot would choose the closest wall and follow it, but in the current iteration, the robot follows the wall closest to it initially, which means that if the wall is to the right initially, it will assume that the wall is continuously to the right of it. This is done rather simply by utilizing a flag within the code.

The third major part involves reacting to objects directly in front of the robot. This must be done while keeping in mind the special requirements for turns by the robot. This is done by using the laser rangefinder to determine if there are objects within a certain range of angles centered at centered at the heading of the robot. If an object is within a certain distance, the robot will merely turn, but, past a threshold, beyond which the robot would have problems turning, the robot stops and turns at a constant rate defined in the MDLe scripts as the previously described constant *r*.

Simulation and Experimental Results in Wall Following and Mapping

The wall following algorithm described in the previous section was implemented using a MatLab function in Simulink. The aim of the simulation was to calculate the robot's path as realistically as possible. Therefore, the laser range finder was simulated by another MatLab function, and the odometry calculations were made by a Simulink subsystem (see Appendix Fig. A8). The user is able to change the shape of the wall that must be followed by inputting one or more explicit functions. In the case of Figure 6, the boundary is composed of two semi circles with 1.5 meter radii and centered at the origin. This is essentially a closed circle with a 3 meter diameter, an idealized version of a real experimental scenario.

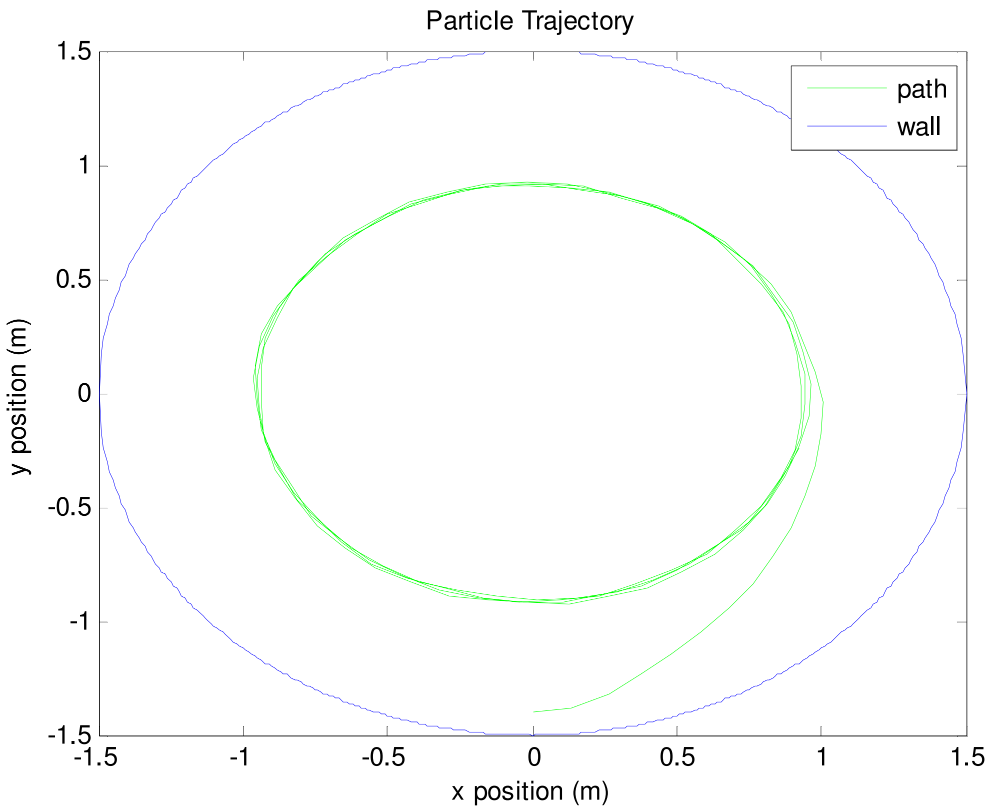

Figure 6: Simulation of Particle Trajectory

The objective of the simulation below was for the robot to follow the wall with a specified speed and distance from the circular boundary. The specified distance, a user defined parameter, was 0.6 meters for this scenario, and allowed tolerance was ± 0.2 meter. Figure 7 illustrates the change in distance from the wall during the first 50 seconds. As time progressed, the distance approaches a value of 0.56 meter, which is well within the tolerance given. Due to the fact that the boundary was being defined as two semicircles reflected across the y-axis, the spikes in the readings occur whenever the robot's heading angle is $\pi / 2$, $3\pi / 2 \ldots$ (can be verified by comparing Fig. 7 with Appendix Fig. A4). Note that these readings are flaws in the simulation technique, not the algorithm.

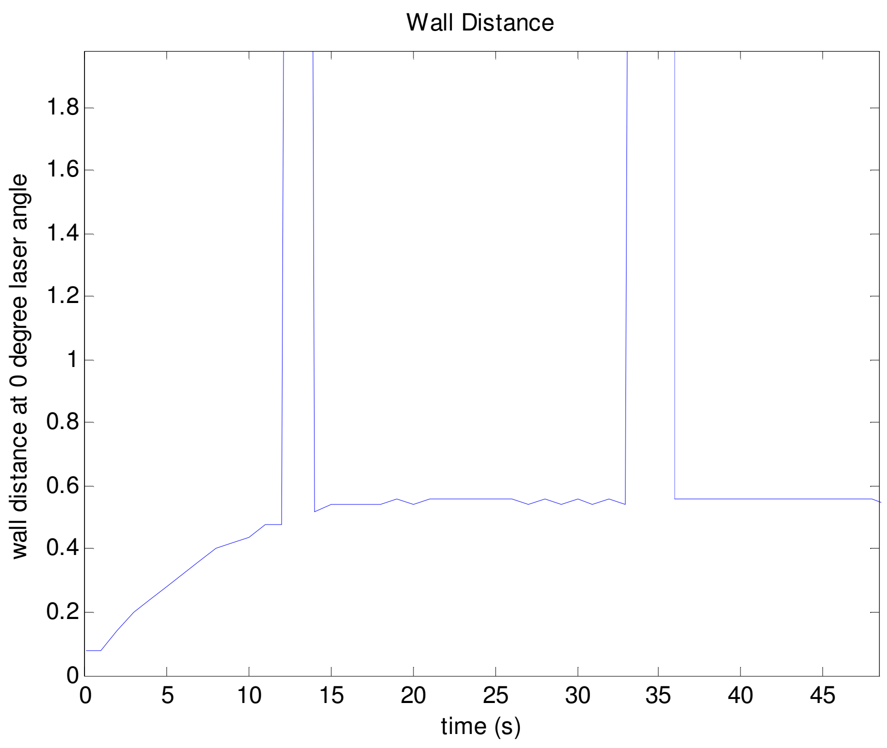

Figure 7: Wall distance as the simulation progressed from 0 to 50 seconds

The simulation shows that the algorithm works well in theory, but not necessarily in practice since it does not take factors like wheel slippage and noisy sensor data into account. This can help explain how Figure 6 shows little variation in the simulated robot's path after four full laps (Appendix Fig. A5) while there is more variation in just 3 circuits for the experiment shown by Figure 8.

This experiment, which is similar to the previous simulation, shows how the actual robot navigates a circular wall. As it circled around, it was taking range measurements from the 0 and 180 degree sensors at a rate of 1hz. One difference from the simulation is the presence of an obstacle in the middle of the "room." The obstacle, a cardboard box, was placed to show how the robot can employ the laser range finder to find the dimensions of objects within the environment it is placed in. As the robot circled around, it was constantly taking range measurements from the 0 and 180 degree sensors.

When measured by hand, the dimensions of the box were approximately 20cm x 40cm and the range finder roughly confirms these dimensions.

To make laser rangefinder readings readable and be able to create a map as shown in Figure 8, conversions must take place between the robot's frame, which is constantly moving with it, and the lab frame. This conversion is described in the Theoretical Background section and relies on reliable localization.

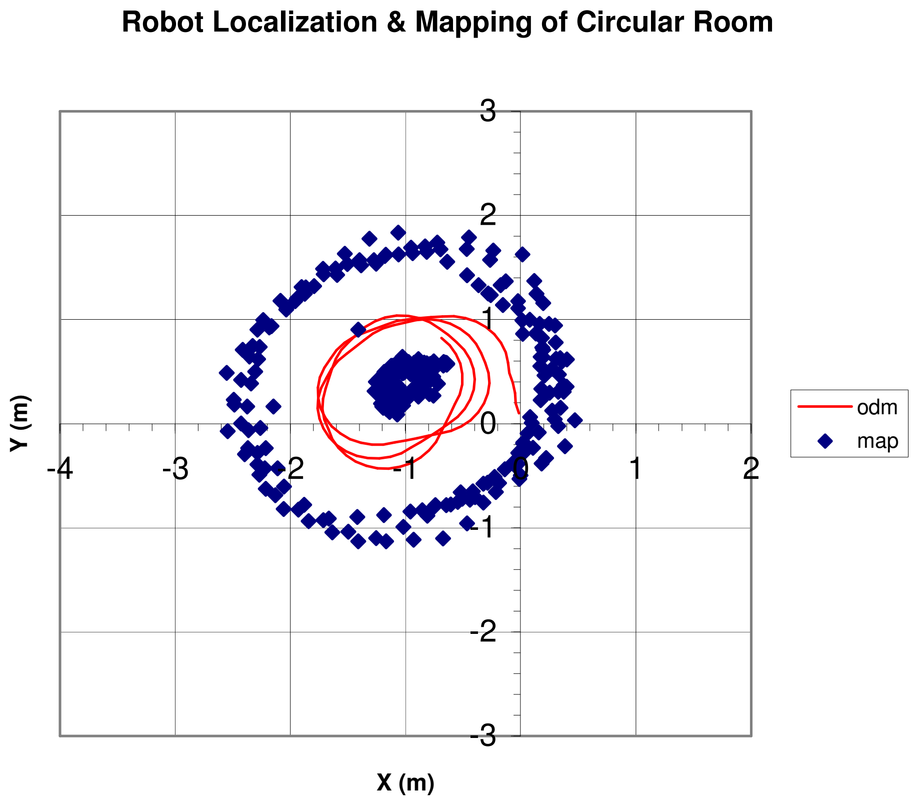

Figure 8: Experimental wall following and mapping a circular boundary w/ object

The quality of the mapping is extremely dependent on the accuracy of the odometry. This is primarily because the robot first needs to know where it is and which way it is facing before it can turn a plain range reading into coordinates that can be mapped. Therefore, as the error accumulates, the map will get distorted. This can be seen slightly in the figure below and in Figure A9, where it is more prominent.

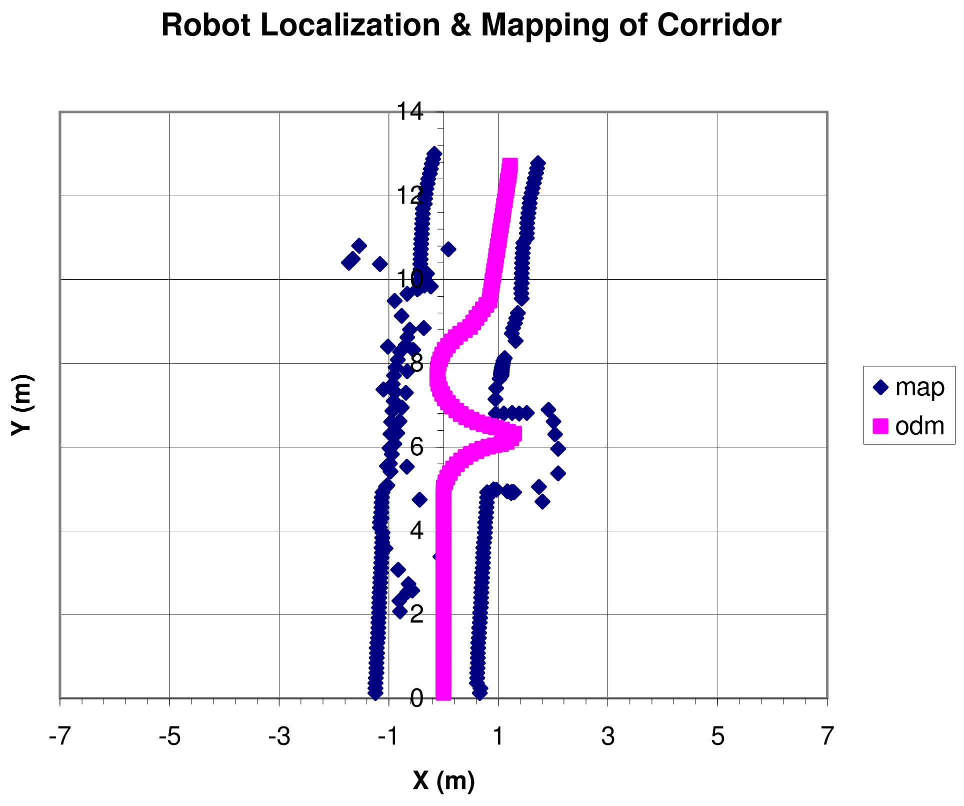

Figure 9: Experimental wall following and mapping 12m section of a corridor

The path above illustrates how the robot navigated a busy corridor in an office building. The pocket on the right is a tricky feature to maneuver, but after avoiding it, the robot settles back nicely to the desired distance, travelling parallel to the wall once again. The few scattered points come from people who walked by the robot. Also, open office doors can be seen very clearly here.

Applications and Conclusions

There are several applications of the associated technologies of this project. Robots could be used to explore unstructured environments. For example, they could be used by firefighters or military personnel when sending a human would be too dangerous. Civilian applications can also be found. One such application is a patrolling robot that uses a map to navigate.

The use of the cricket system can be especially useful and accurate when weighed against odometry, which has the problem of slippage and cumulative error. It is conceivable that the use of the cricket in indoor environments would allow for fully autonomous robots with a variety of tasks to navigate with good precision.

The general principles obtained in the study of the control theory and the general obstacles behind a real robot easily extend themselves elsewhere. The use of easily duplicable high level programming language allows for easy repetition in implementation in other systems as well as a simpler way to conceptualize portions of code.

Appendix

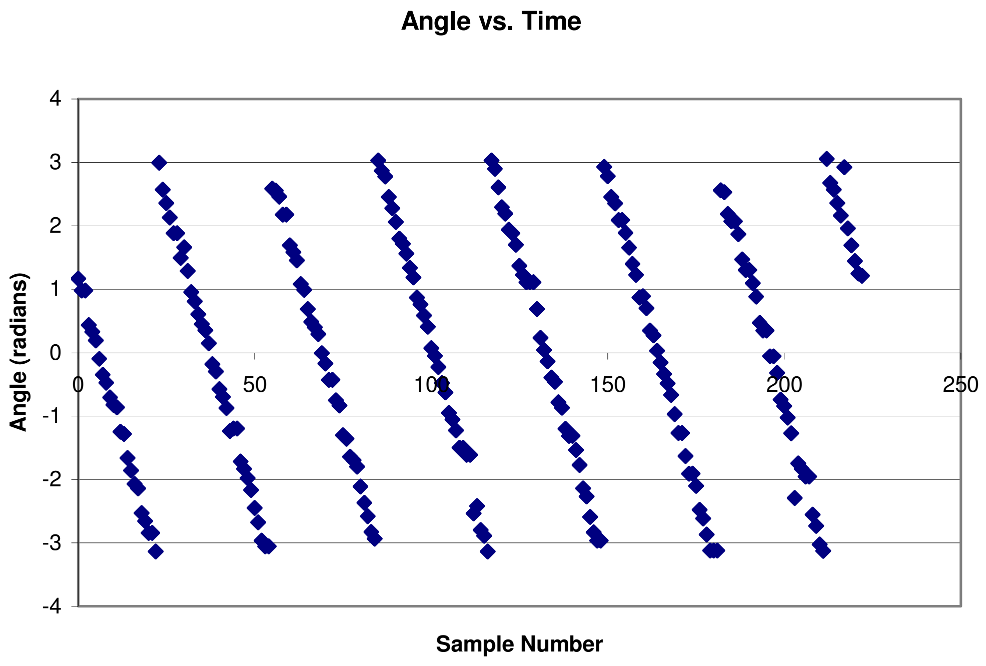

Figure A1: Cricket heading direction on a constantly spinning robot (ω = 0.1 rad/s)

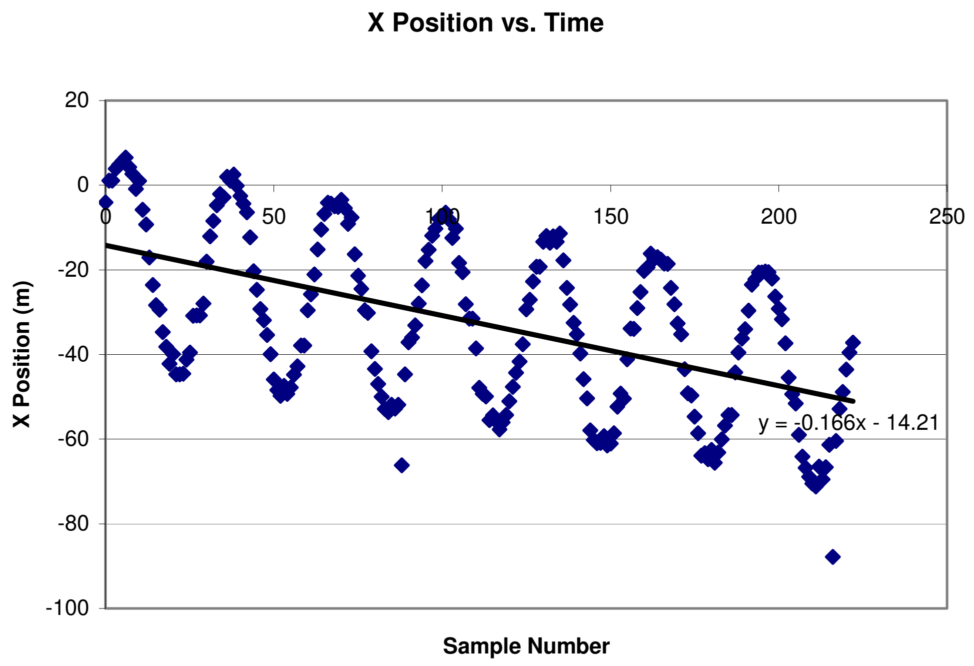

Figure A2: Cricket x pos over time on a constantly spinning robot (ω = 0.1 rad/s)

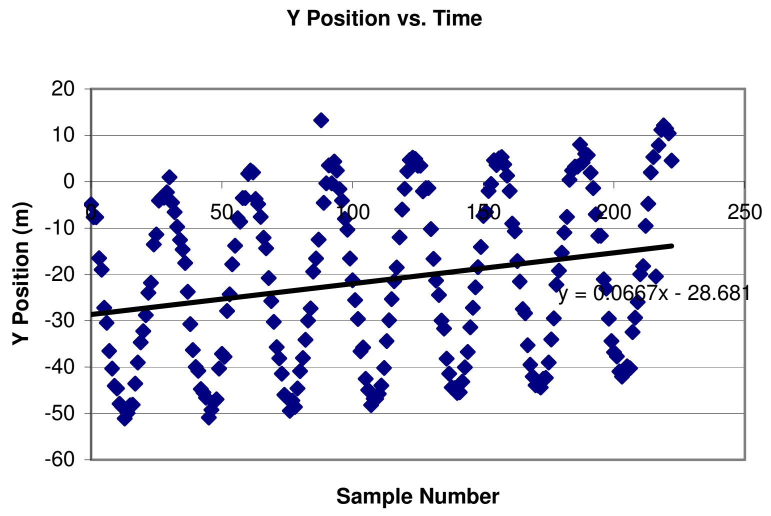

Figure A3: Cricket y pos over time on a constantly spinning robot (ω = 0.1 rad/s)

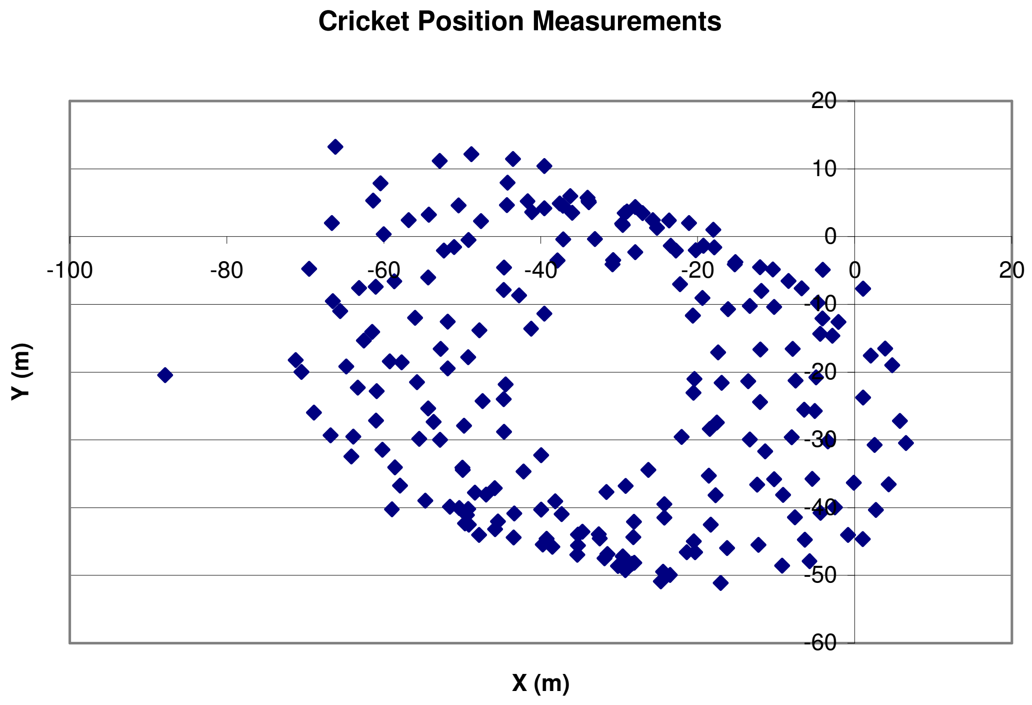

Figure A4: Cricket position measurements taken as robot spins with an angular velocity of 0.1 rad/s

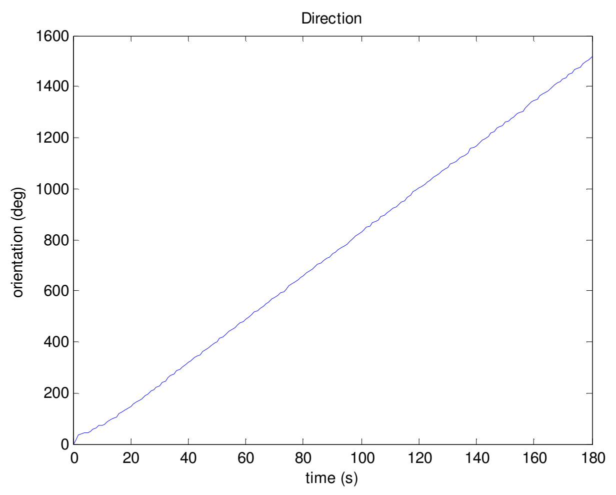

Figure A5: Simulation of heading direction over time

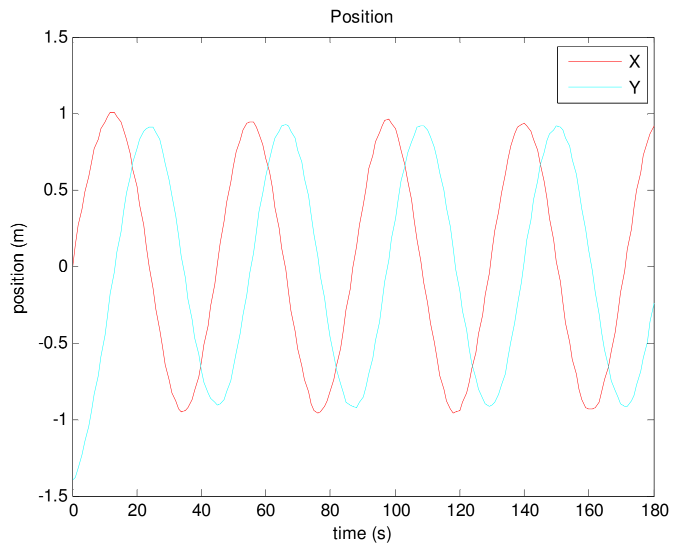

Figure A6: Simulation of X and Y positions over time

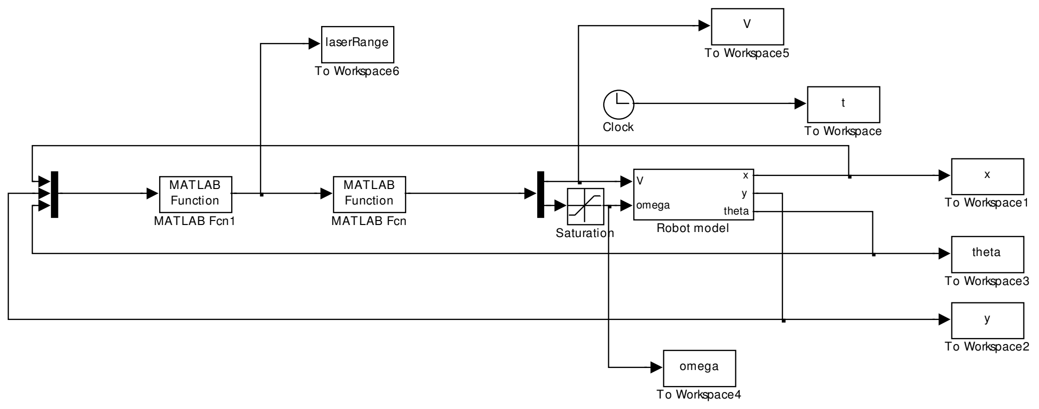

Figure A7: Simulink system model

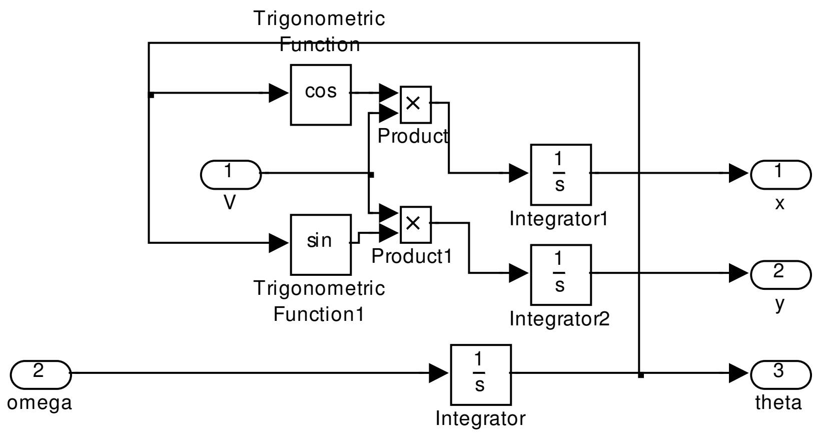

Figure A8: Simulink Robot Model subsystem

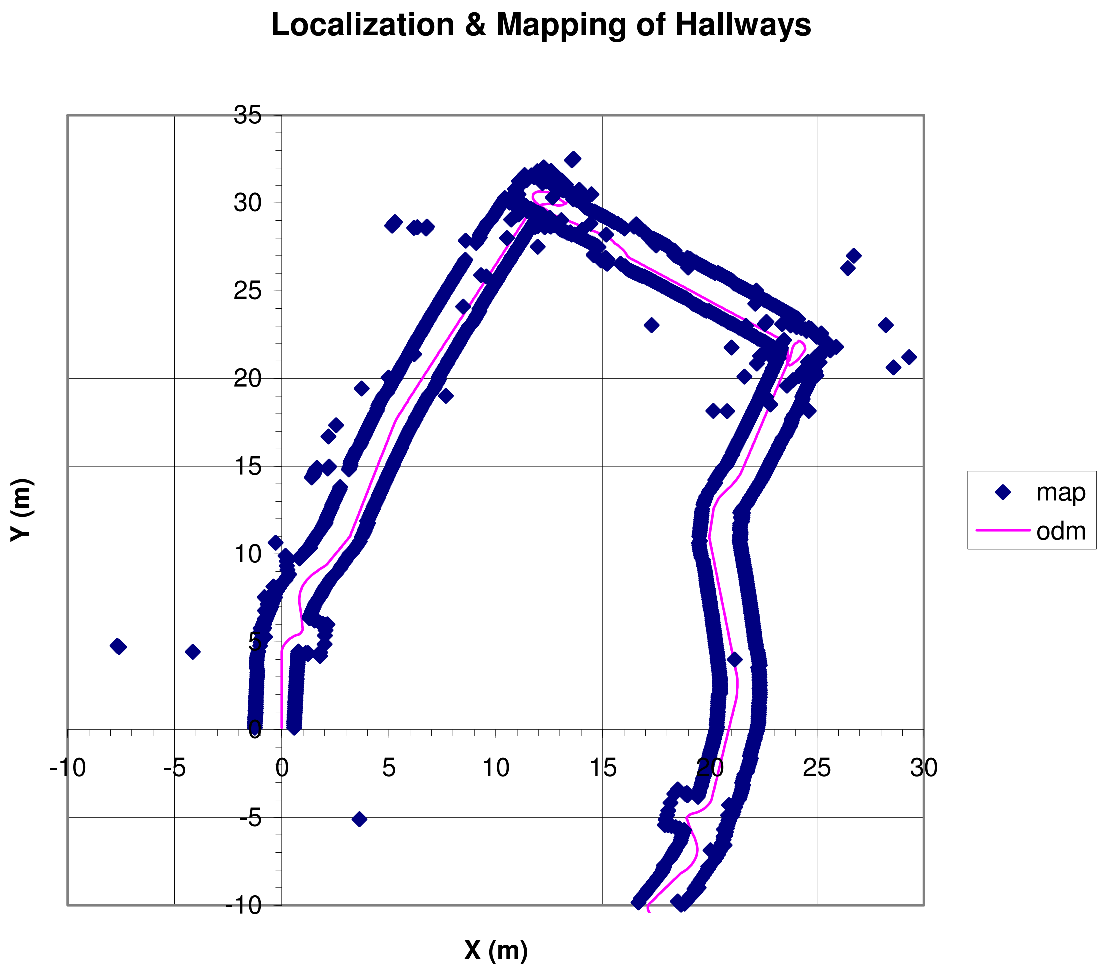

Figure A9: Example of error accumulation from odometry inaccuracies

References

[1] Andersson, Sean B., and Dimitrios Hristu. Symbolic Feedback Control for Navigation. Vol. 51, No. 6. IEEE Transactions on Automatic Control, June 2006. pp 926-937.

[2] Galloway, Kevin. Basic Operating Manual for ISL Systems.

[3] Krishnaprasad, P. S. "Pursuit and Cohesion." 46th IEEE Conference Decision and Control. New Orleans. Dec. 2007.

[4] Sheng, Jansen, and Scott Watson. Pursuit Techniques on Pioneer Robots. Rep.No. Institute for Systems Research, University of Maryland. 2007.

[5] Young, Travis. The Integration of Internal and External Positioning. Intelligent Servosystems Laboratory, Institute for Systems Research. 2007.

[6] Zhang, Fumin, Alan O'Connor, Derek Luebke, and P. S. Krishnaprasad. Experimental Study of Curvature-based Control Laws for Obstacle Avoidance. IEEE International Conference on Robotics & Automation, April 2004. pp 3849-3854.

[7] Zhang, F., E. W. Justh, P.S. Krishnaprasad. Boundary following using gyroscopic control. 43rd IEEE Conference on Decision and Control, Dec 14-17, 2004. pp 5204-5209.