ON-LINE OPTICAL FLOW FEEDBACK FOR MOBILE ROBOT LOCALIZATION/NAVIGATION

A Thesis

by

DAVID KRISTIN SORENSEN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2003

Major Subject: Mechanical Engineering

ON-LINE OPTICAL FLOW FEEDBACK

FOR MOBILE ROBOT LOCALIZATION/NAVIGATION

A Thesis

by

DAVID KRISTIN SORENSEN

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Sooyong Lee (Chair of Committee)	Craig Smith (Member)
Johnny Hurtado (Member)	John Weese (Head of Department)

May 2003

Major Subject: Mechanical Engineering

ABSTRACT

On-Line Optical Flow Feedback
for Mobile Robot Localization/Navigation. (May 2003)
David Kristin Sorensen, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Sooyong Lee

Open-loop position estimation methods are commonly used in mobile robot applications. Their strength lies in the speed and simplicity with which an estimated position is determined. However, these methods can lead to inaccurate or unreliable estimates. Two position estimation methods are developed in this thesis, one using a single optical sensor and a second using two optical sensors. The first method can accurately estimate position under ideal conditions and when wheel slip perpendicular to the axis of the wheel occurs. A second method can accurately estimate position even when wheel slip parallel to the axis of the wheel occurs. Location of the optical sensors is investigated in order to minimize errors caused by inaccurate sensor readings. Finally, the method is implemented and tested using a potential field based navigation scheme. Estimates of position were found to be as accurate as dead-reckoning in ideal conditions and much more accurate in cases where wheel slip occurs.

ACKNOWLEDGMENTS

Special thanks to Dr. Sooyong Lee for his help with everything from concept to implementation. He has insights into the fields of mechatronics and robotics that few others do. Without his constant support and effort, this thesis would never have come to fruition. I also consider Dr. Lee a good friend and hope that we can keep in touch in years to come.

Thanks to Dr. Craig Smith for helping to clarify and improve the writing of this thesis. His technical language skills are surpassed by few, none of whom I know.

I would like to thank Dr. Jonny Hurtado. He has been a great encouragement. I appreciate his kind spirit and sincerely thank him for it.

Thanks to Andrew Rynn for his work on hardware implementation as well as everyday problem solving. He also implemented the Matrix class in Java. Without it, the navigation implementation and experiments would have taken far longer to complete. His grammatical genius has greatly improved this thesis. He has also been a great "sounding board", so to speak, in much of my work. Andrew has been a great source of entertainment and distraction in general! Seriously, his friendship and point of view have been a good change from the norm.

Thanks to Mark Ovinis for his work on the design and simulation of an optical flow sensor. His work in Matlab has been greatly appreciated and useful in proving that the techniques using optical sensors could be implemented using CCD cameras as well. Much of his work is included in Appendix A.

Thanks to Volker Smukala for his work on multiple sensors and sensor placement. Though he has been here for only a few months, his level of effort and knowledge have been greatly helpful in understanding how errors caused by inaccurate sensor readings can be reduced.

Thanks to my parents for their encouragement, support and guidance. They have always been there when I've needed them. Their love is key to all of the successes in my life.

No set of acknowledgements could be complete if I did not mention my wife. She has been my support, strength, helper, friend, and encouragement. I could not have finished as quickly without the constant reminders of pain or dismemberment if I did not graduate on time! Her love and encouragement have been generously given. I thank her for being who she is.

I leave you with a verse: May the glory of the Lord endure forever; may the Lord rejoice in his works. He who looks at the earth, and it trembles, who touches the mountains, and they smoke. Psalm 104:31-32

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

A. Motivation

Accurate position estimation is a key component to the successful operation of most autonomous mobile robots. In general, there are three phases that comprise the movement sequence of a mobile robot: localization, path planning, and path execution. During localization, the position and orientation in the reference coordinate system is determined using external sensors. A path is then planned that passes through a goal point or a series of intermediate via points. The final phase is the execution of the planned path. The movement sequence is repeated so that the robot will remain on course towards the goal.

Localization can be further decomposed into two types, absolute and relative [1]. Absolute localization relies on landmarks, maps, beacons, or satellite signals to determine the *global* position and orientation of the robot. Relative localization (or intermediate estimation) is usually used during movement, because absolute localization methods are more time consuming.

Commonly, dead-reckoning (open-loop estimation) is used for intermediate estimation of position during path execution. Dead-reckoning is often used when wheel encoders are available for drive wheel position measurement. However, due to errors in kinematic model parameters, wheel slip, or an uneven surface, poor position estimates may occur. Poor estimates in position during path execution require more frequent localization, incurring extra overhead and possibly slowing the movement of the robot. A worse scenario is one where poor estimates would cause a collision,

---

The journal model is *IEEE Transactions on Automatic Control.*

impeding the operation of the robot. It is therefore important to minimize errors in estimated position during the path execution phase.

Dead-reckoning usually fails (causes poor estimates) in the presence of wheel slip. However, using the methods described in this thesis, accurate estimates of position can be maintained even when wheel slip occurs. Indeed, when either systematic errors (errors related to robot properties or parameters) or non-systematic errors (random errors caused by the environment) occur, dead-reckoning usually fails to accurately estimate position. Dead-reckoning only produces accurate estimates when all kinematic constraints are upheld. Two methods are developed, implemented and tested using inexpensive optical sensors. The placement of the optical sensors affects estimation errors. An optimal placement scheme (in the sense of minimizing estimation errors) is proposed. Additionally, path planning is discussed and implemented. Using the methods described, optical sensors can be used to accurately estimate the position of the robot.

B. Previous Work

In most movement schemes, dead-reckoning errors are an accepted part of the movement sequence. These errors are usually counteracted by making frequent localization "stops." This is unfortunate, because in many cases dead-reckoning proves to be inaccurate. If a more accurate method were available, fewer intermediate localizations would be necessary. This would in turn free computational resources that could be used to accomplish higher level tasks.

Other researchers have implemented similar dead-reckoning correction techniques. In [1] a towed robot (called a trailer), which has accurate wheel encoders and a rotary encoder on the connection link, is used to determine the relative movement and

direction of the trailer with respect to the robot. This information allows an accurate estimated position to be maintained. This method reduced dead-reckoning errors by an order of magnitude or more. However, the added bulk of the trailer can complicate the movement of the robot making it difficult to navigate in close quarters, especially when moving backwards.

Another method of correcting dead-reckoning errors in navigation uses optical flow. In [2], optical flow was used to aid in navigation of an omnidirectional robot. A CCD camera was positioned at a 45° downward angle to the ground in front of the robot. The optical flow obtained was combined with the results of dead-reckoning via maximum likelihood technique. The method used to calculate optical flow is quite complex, requiring a large number of computations to obtain good results.

GPS has many applications to mobile robot navigation [3]. GPS has been used to correct position estimations by adjusting kinematic parameters. In [4] GPS was used to correct for heading and step size in a pedestrian navigation system. When GPS is available the parameters are adjusted such that when GPS service is unavailable, a good estimate of position is maintained. This method is easily portable to mobile robots. However, GPS accuracy is limited, so when fine positional control is necessary it can prove ineffective. GPS is further limited by the fact that it will only work in outdoor environments where line-of-sight to at least 3 satellites is possible.

Barshan and Durant-Whyte developed inertial navigation methods for mobile robots in [5]. A series of solid state gyros, accelerometers and tilt sensors were employed in conjunction with an extended Kalman filtering method to estimate position and orientation. Using accelerometers to determine position and orientation, however, has several drawbacks such as 1-8 cm/s drift rate. Also, the minimum detectable acceleration can sometimes be too large to detect small motion. Inertial methods can be quite computationally intensive, expensive and complex.

CHAPTER II

BACKGROUND INFORMATION

A. The Movement Sequence

Mobile robots typically have three phases that comprise their movement. These phases are localization, path planning and path execution. In most movement schemes, the phases are usually repeated for two reasons. The first is that many environments are complex, requiring paths to include multiple via points. It is often convenient to stop and localize at the via points in order to maintain a better estimate of position. The second reason that most movement schemes are repeated is that most environments are dynamic. That is, there are obstacles in the environment that have the ability to change positions. For this reason, planned paths must be updated often so that a new "map" of obstacles can be tracked and collisions avoided. This sequence is shown in Figure 1.

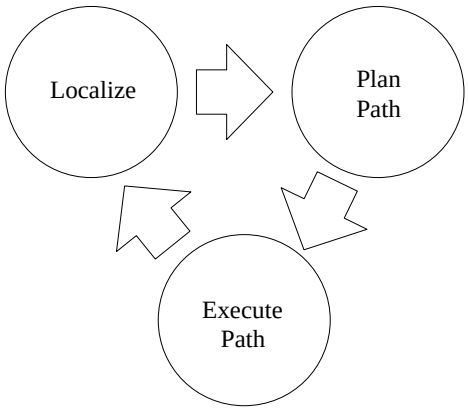

Fig. 1. Typical Mobile Robot Movement Sequence

The first phase of movement is localization, the act of finding the location of the robot. A mobile robot uses onboard sensors (such as a camera, range sensors or GPS) to determine its position and orientation based on information it has about its environment. A localization scheme gives a *global* picture of the location of the robot. This *global* position is usually referenced to some known location in the environment. Most localization schemes use sensors that are relatively slow when compared to dead-reckoning. This can greatly slow the localization process, making frequent updates impractical.

The second phase is path planning, where information about the location and orientation of the robot, as well as information about the goal and obstacles in the environment is used to determine a desired path. If an obstacle is between the robot and goal, a path around the obstacle must be planned. This is illustrated in Figure 2.

The final phase of movement is path execution. During the execution phase, the previously determined path is followed. This step requires detailed information about the mobile robot system. In actual implementation, the underlying characteristics of the system should be known *a priori* for successful path execution. Usually, the current position is updated using an open-loop estimation method. The most commonly used method is dead-reckoning. In dead-reckoning, the forward kinematics of the vehicle are used to maintain an estimate of the current position. The forward kinematics of a differential drive robot will be described in Section C of this chapter. One major problem with dead-reckoning (and open-loop estimation methods in general) is the error accumulation as path length increases.

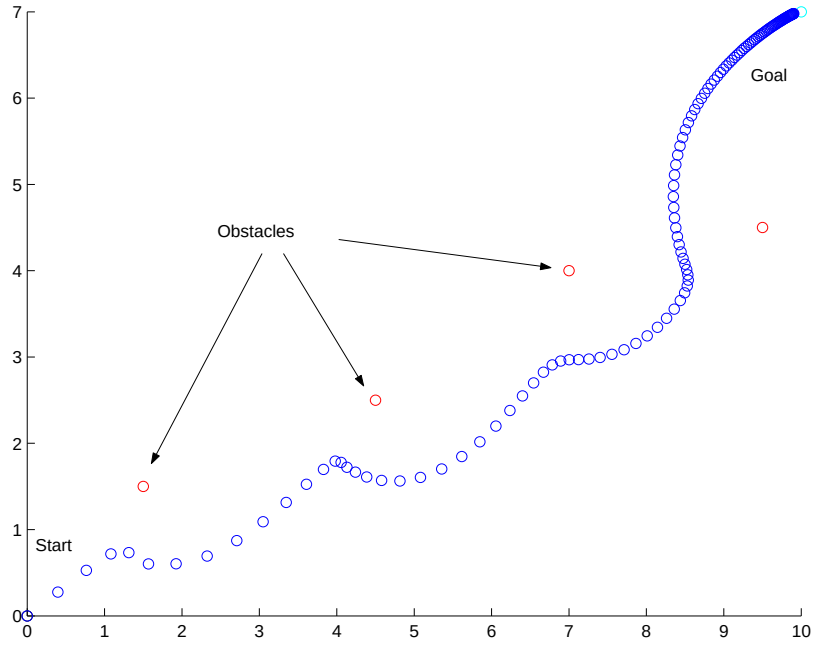

Fig. 2. Planning a Path

B. Dead-reckoning

Dead-reckoning is usually used because of the simplicity of the method and accompanying speed with which an estimate can be obtained. That is to say that the estimated position is easy to update and is not computationally intensive to obtain. Dead-reckoning, however, has some drawbacks that can make it inaccurate.

Dead-reckoning errors can be divided into two types: systematic and non-systematic errors [1]. Systematic errors are errors related to the parameters or properties of the robot. Errors caused by incorrect wheel diameter or wheel base are examples of systematic errors. Systematic errors can cause an accumulation of error in the estimated position, but these errors can be reduced via precise calibration of parameters in the robot kinematic model. However, varying load conditions can cause some robot parameters to change. Errors due to varying parameters are difficult to reduce using traditional dead-reckoning.

Non-systematic errors are those errors caused by something outside the robot system. Surface inconsistencies are the cause of most non-systematic errors. These types of errors are harder to account for because they are unpredictable and difficult to detect without using sensors other than drive wheel encoders. In most dead-reckoning schemes, the forward kinematic relationship is used to update the estimated position and orientation of the robot. When errors occur, the estimated position can become very inaccurate. This is illustrated in Figure 3. The ellipses represent a certain percent probability of the robot's actual location.

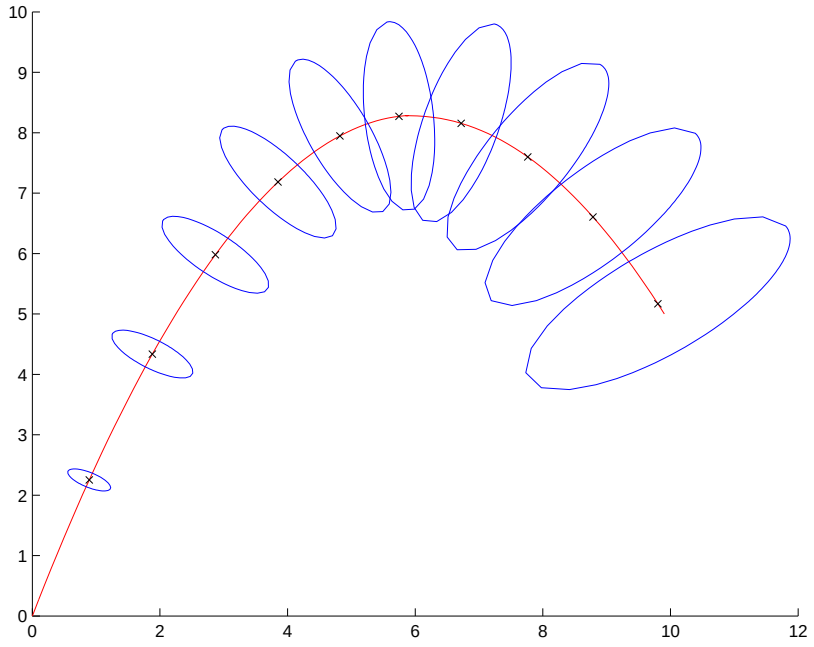

Fig. 3. Accumulation of Dead-reckoning Errors

C. Forward Kinematics

The relationship between the movement of the drive wheels and the movement of the point of interest on the robot is called forward kinematics. In this section, the

forward kinematic relationship for a differential drive robot is formulated.

Forward kinematics for a differential drive robot are derived based on the velocity equations for the left and right wheels. Figure 4 shows the robot model used to find the forward kinematic relationship.

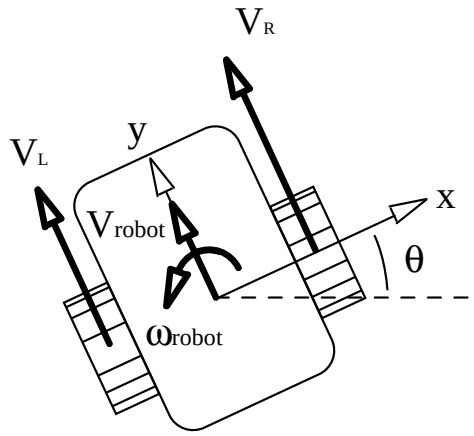

Fig. 4. Robot Coordinates for Forward Kinematic Model

Inputs to the forward kinematic relationship are the velocities of the wheels, V_L and V_R. The output from the forward kinematic relationship is the velocity of the center of the robot, V_{robot}, and the angular velocity of the robot, ω_{robot}. For round wheels,$V_{L,R}$ are defined as the angular velocity of the drive wheels, $\omega_{L,R}$, multiplied by the wheel radius, r_w.

$$V_{L,R} = \omega_{L,R} \cdot r_w \tag{2.1}$$

Assuming the robot is a rigid-body, the velocity of the center of the robot can be determined using the wheel velocities and the distance between the wheels. Since the wheel velocities are assumed to be perpendicular to the axis of rotation of the wheels, the velocity of the robot will be the average of these velocities and is always

in the robot y direction. Similarly, the angular velocity of the center is the difference in velocities of the left and right wheels divided by the distance between the wheels, D. Equations 2.2 and 2.3 below show these relationships.

$$V_{robot} = \frac{V_L + V_R}{2} \tag{2.2}$$

$$\omega_{robot} = \frac{V_R - V_L}{D} \tag{2.3}$$

Using these equations, V_{robot} and ω_{robot} are now transformed to the *global* coordinate system. This is done using the coordinate transformation shown in equation 2.4.

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V_{robot,x} \\ V_{robot,y} \\ \omega_{robot} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{2.4}$$

where $\dot{x}$, $\dot{y}$, and $\dot{\theta}$ are the changes in x position, y position, and orientation, respectively.

Since $V_{robot,x} = 0$, equations 2.2, 2.3 and 2.4 can be simplified to:

$$\dot{x} = -V_{robot,y} \sin\theta \tag{2.5}$$

$$\dot{y} = V_{robot,y} \cos\theta \tag{2.6}$$

$$\dot{\theta} = \omega_{robot} \tag{2.7}$$

Using equation 2.5, the *global* coordinates can now be updated in a stepwise manner. This method is known as dead-reckoning and is used as a benchmark for comparisons with the new optical flow rigid-body methods presented.

D. Inverse Kinematics

Inverse kinematics is the relationship between the movement of the robot and the corresponding movement of the wheels. The inverse kinematic relationship is the reverse of the forward kinematic relationship. That is, instead of computing the movement of the center of the robot based on the movement of the wheels, it is desired to find the movement of the wheels given a desired V_{robot} and ω_{robot}.

Rewriting equation 2.5 in matrix form:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -\sin(\theta) & 0 \\ \cos(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} V_{robot} \\ \omega_{robot} \end{bmatrix} \tag{2.8}$$

Rewriting equations 2.2 and 2.3 in matrix form:

$$\begin{bmatrix} V_{robot} \\ \omega_{robot} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{D} & -\frac{1}{D} \end{bmatrix} \begin{bmatrix} V_R \\ V_L \end{bmatrix} \tag{2.9}$$

Combining 2.8 and 2.9:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -\frac{\sin(\theta)}{2} & -\frac{\sin(\theta)}{2} \\ \frac{\cos(\theta)}{2} & \frac{\cos(\theta)}{2} \\ \frac{1}{D} & -\frac{1}{D} \end{bmatrix} \begin{bmatrix} V_R \\ V_L \end{bmatrix} \tag{2.10}$$

The above equation is in the form $b = Ax$. The desired quantities are the wheel velocities, V_R and V_L. In order to solve for the wheel velocities, the equation can be put in the form $x = A^{\#}b$, where $A^{\#}$ is the pseudo-inverse. This method is also known as the least squares solution. The equation returns the vector x that best satisfies the equation "$A \cdot x$ is approximately equal to b", in the least squares sense.

The solution to this least squares problem can be found as follows:

$$x = A^T \left(A^T A\right)^{-1} b \tag{2.11}$$

For a discrete path with desired changes in position, $\dot{x}_t = [\dot{x}_t, \dot{y}_t, \dot{\theta}_t]^T$, the pseudo-inverse must be recomputed every iteration. This is because θ is not constant and is present in the A matrix.

CHAPTER III

SENSOR INTERPRETATION

Optical flow estimation using a small CCD camera is described in Appendix A, however, commercial optical mouse sensors were used in the methods that follow. These sensors are inexpensive, reliable, accurate, and very fast. Specifics about the sensors can be found in Chapter VI.

In Appendix A, computed changes in position were calculated using two successive 8×8 images. Without a larger image (100×100 or greater), information about rotation is unreliable. The rotation of the robot must be estimated using other methods.

Two methods of estimating the location and orientation of the robot are investigated in this chapter. The first uses a single sensor and a constraint on the kinematics of the robot. The second method uses two sensors and no kinematic constraints.

A. Single Optical Sensor

Unless the optical sensor is placed at the point of interest on the robot, measurements directly from the optical sensor will not be useful. Additionally, because the optical sensor only provides displacements in the x and y directions information about the angular displacement of the robot is lost. As a result, another method must be used to determine the angular velocity of the robot. Figure 5 shows the coordinates used for the rigid body method.

The robot can be viewed as a rigid-body where the velocity at the sensor (Δx/Δt and Δy/Δt) is known. The kinematic constraints of a differential drive robot allow the calculation of movement, given this velocity. Specifically, the center of the robot is assumed to move only in a direction perpendicular to the wheel axis. The following

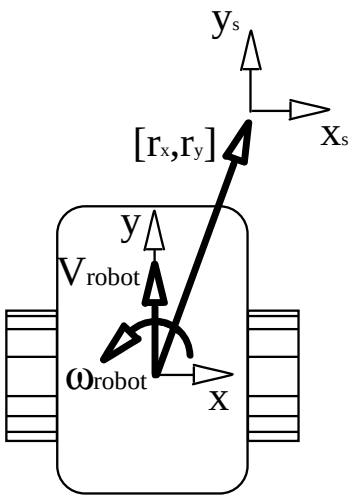

Fig. 5. Rigid-Body Model for Sensor Interpretation

equation relates the velocity of a point on the axis of rotation of a rigid body (robot center) with a point not on that axis of rotation (the sensor location):

$$V_r = V_s + \omega_r \times r_{s/r} \tag{3.1}$$

Where V_r is the velocity of the center of the robot, V_s is the sensor velocity, ω_r is the angular velocity of the robot, and $r_{s/r}$ is the vector from the location of the sensor to the robot center (Figure 5). Rewriting in matrix form:

$$\begin{bmatrix} V_{r,x} \\ V_{r,y} \end{bmatrix} = \begin{bmatrix} V_{s,x} \\ V_{s,y} \end{bmatrix} + \begin{bmatrix} -\omega r_y \\ \omega r_x \end{bmatrix} \tag{3.2}$$

It should be noted that the sensor may not be oriented along the x and y robot coordinates. Denoting the orientation of the sensor as β, the sensor coordinates as $\tilde{x}$ and $\tilde{y}$, the following transformation is made to robot coordinates:

$$\begin{bmatrix} V_{s,x} \\ V_{s,y} \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{bmatrix} \begin{bmatrix} V_{s,\tilde{x}} \\ V_{s,\tilde{y}} \end{bmatrix} \tag{3.3}$$

With the assumption that $V_{r,x} = 0$, ω_r can be found using the $\hat{i}$ equation and similarly, $V_{r,y}$ using the $\hat{j}$ equation. From the rigid-body velocity equation and the kinematic constraints, we obtain the following two equations:

$$\omega_r = -\frac{V_{s,x}}{r_y} \tag{3.4}$$

$$V_{r,y} = V_{s,y} + \frac{r_x}{r_y} V_{s,x} \tag{3.5}$$

The robot's velocity can now be easily translated to the *global* coordinate system. A similar transformation was performed in (2.4).

A similar approach to using the kinematic constraint $V_{r,x} = 0$ is to use the weighted pseudo-inverse. This method gives one possible solution to the underdetermined case, where three variables are determined from only two equations. Rewriting the rigid body equation in matrix form:

$$\begin{bmatrix} V_{s,x} \\ V_{s,y} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -r_y \\ 0 & 1 & r_x \end{bmatrix} \begin{bmatrix} V_{r,x} \\ V_{r,y} \\ \omega_r \end{bmatrix} \tag{3.6}$$

The underdetermined pseudo-inverse takes the form:

$$\begin{bmatrix} V_{r,x} \\ V_{r,y} \\ \omega_r \end{bmatrix} = W^{-1} A^T (A W^{-1} A^T)^{-1} \begin{bmatrix} V_{s,x} \\ V_{s,y} \end{bmatrix} \tag{3.7}$$

where W is a weighting matrix. If we heavily weight the velocity in the x direction to be small, nearly an identical solution to the constrained case emerges. A suitable

weighting matrix follows:

$$W = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

The figure on page 40 shows how varying the weighting matrix changes the apparent movement of the robot using experimental data.

The single sensor method will give better results when wheel slip perpendicular to the wheel axis occurs than dead-reckoning. In fact, as long as the kinematic constraint ($V_{r,x} = 0$) is not violated, the method will give accurate position estimates. If multiple sensors are used, the constraint can be removed, allowing accurate positional estimates in all kinematic conditions. This is the topic of the following section.

B. Multiple Optical Sensors

Given a second sensor, there are at least two approaches which could be used to interpret the data. First, the additional sensor could be viewed as a redundant sensor which would lead to data fusion methods. Data fusion should give better results than a single sensor given data errors. However, it would still give inaccurate results if the kinematic constraint, $V_{r,x} = 0$ were violated.

Alternatively, the velocity information from the second sensor can be used to determine the motion of the rigid-body without the constraint, $V_{r,x} = 0$, that was necessary using only a single sensor.

Looking again at the rigid-body model, two sensors at different locations give ample information to determine the motion of the rigid-body without any kinematic

constraints. In this case, the rigid-body model leads to the following four equations.

$$V_{r,x} = V_{1,x} + \omega_r \cdot r_{1,y} \tag{3.9}$$

$$V_{r,y} = V_{1,y} - \omega_r \cdot r_{1,x} \tag{3.10}$$

$$V_{r,x} = V_{2,x} + \omega_r \cdot r_{2,y} \tag{3.11}$$

$$V_{r,y} = V_{2,y} - \omega_r \cdot r_{2,x} \tag{3.12}$$

where $V_{r,x}$ and $V_{r,y}$ are the velocities of the center of the robot, ω_r is the angular velocity of the robot, $V_{i,x}$ and $V_{i,y}$ are the sensor velocities and $r_{i,x}$ and $r_{i,y}$ are the x and y distances from the i^{th} sensor position to the robot center.

To solve this system of equations for the angular and linear velocities of the robot, the system is written in the form $A \cdot \mathbf{X} = \mathbf{b}$:

$$\begin{bmatrix} 1 & 0 & -r_{1,y} \\ 0 & 1 & r_{1,x} \\ 1 & 0 & -r_{2,y} \\ 0 & 1 & r_{2,x} \end{bmatrix} \cdot \begin{bmatrix} V_{r,x} \\ V_{r,y} \\ \omega_r \end{bmatrix} = \begin{bmatrix} V_{1,x} \\ V_{1,y} \\ V_{2,x} \\ V_{2,y} \end{bmatrix} \tag{3.13}$$

This over-determined system can be solved using the pseudo-inverse [6], $A^{\#}$, which was mentioned earlier in Chapter II.

The least squares overdetermined solution can also be weighted similarly to the underdetermined case. However, the weighings would be of the sensed velocities $(V_{1,x}, V_{1,y}, V_{2,x}, V_{2,y})$. It is not readily apparent which of these velocities should receive a higher importance than another. This avenue of research was not investigated.

Although multiple sensors do not increase accuracy when compared to a single sensor, the kinematic constraint can now be removed. This yields a powerful method to determine intermediate estimates of the robot position and orientation.

CHAPTER IV

SENSOR PLACEMENT

A. Optimal Sensor Location

Another important issue to consider is minimizing error. Using the method described in the preceding chapter, several possible sources of error are present. These include errors in the position and orientation of the sensors and errors in the sensed velocities. Minimizing these errors will lead to better estimates of robot position.

In order to determine the best position for a single sensor, the error in the measurement of the velocity must be minimized. The maximum absolute deviation of a function F(x_0,x_1,$\cdots$,x_n) is defined as:

$$dF = \left|\frac{\partial F}{\partial x_0}\right| \cdot dx_0 + \cdots + \left|\frac{\partial F}{\partial x_n}\right| \cdot dx_n \tag{4.1}$$

Using this definition and the previously given single sensor rigid-body model, the maximum absolute deviations are as follow.

$$d\omega_{robot} = \left|-\frac{1}{r_y}\right| \cdot dV_x + \left|-\frac{V_x}{r_y^2}\right| \cdot dr_y \tag{4.2}$$

$$dV_{robot} = \left|\frac{r_x}{r_y}\right| \cdot dV_x + |1| \cdot dV_y + \left|\frac{V_x}{r_y}\right| \cdot dr_x + \left|\frac{V_x \cdot r_x}{r_y^2}\right| \cdot dr_y \tag{4.3}$$

Errors in sensor orientation are ignored because calibration of the sensors can easily be performed using a straight line path. Sensor orientation can be tuned such that the sensed path closely matches the actual path. Additionally, the position of the sensor is known within a few millimeters and is therefore assumed to be exactly known. (In experiments, the sensors were placed at 20cm from the center of the robot, and measurement errors are on the order of 0.5cm.) This leads to a further

simplified equation:

$$d\omega_{robot} = |-\frac{1}{r_y}| \cdot dV_x \tag{4.4}$$

$$dV_{robot} = |\frac{r_x}{r_y}| \cdot dV_x + |1| \cdot dV_y \tag{4.5}$$

A plot of first order deviations of these equations are shown in Figures 6 and 7.

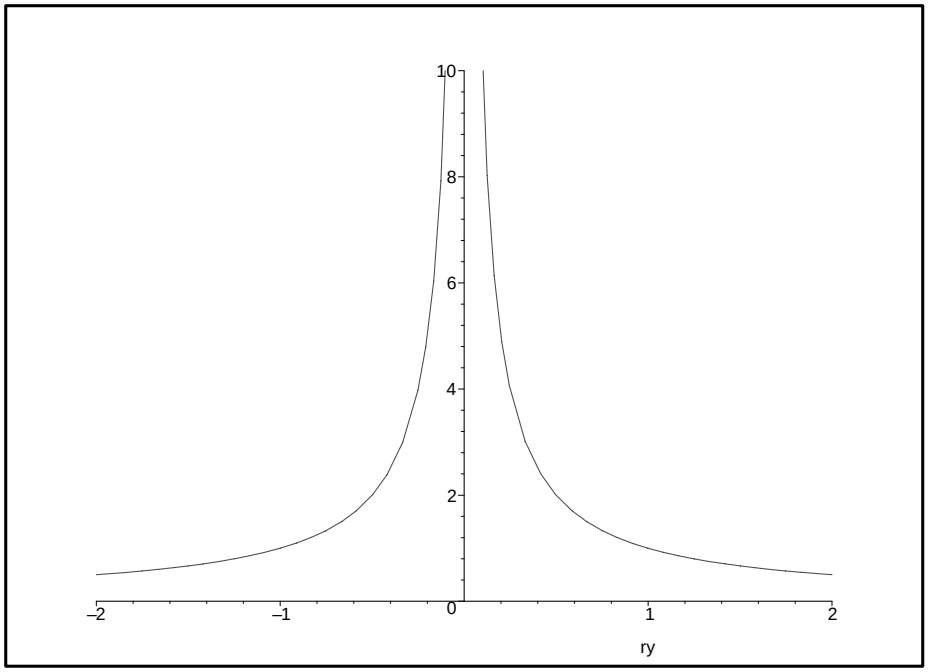

Fig. 6. Error of ω_{robot} versus r_y

As one can see, the optical sensor should be located as far away as possible from the center of the robot along the y-axis, because the global minima is located at $r_x = 0$. Ideally, the error in the determination of the robot's position will be zero for r_x=0 and $r_y = \infty$. However, this is both impractical physically and mathematically. Looking at (3.4) if $r_y = \infty$ then ω always equals zero, which is unacceptable.

To find the optimal position using two sensors, again the maximum absolute

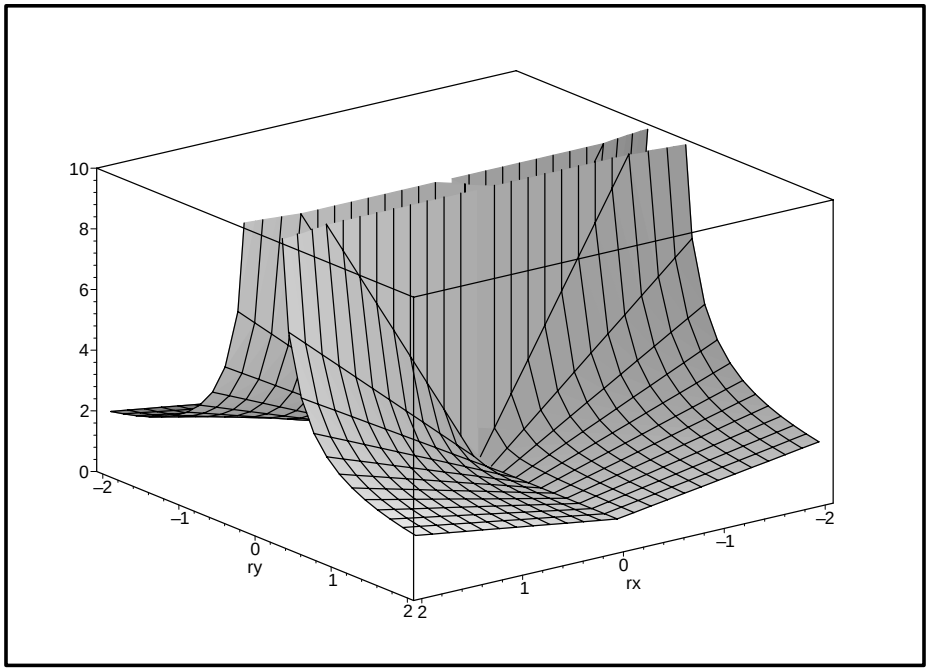

Fig. 7. Error of v_{robot} versus r_x, r_y

deviations of the functions are used. Because of the large number of parameters in each equation, visualizing the deviations is difficult. Assuming again that the positions of the sensors relative to the robot's center can be measured exactly, the only error present is in the sensor velocities.

The first observation is that there are a range of positions where the errors become minimal. To minimize the error in the measurement of the angular velocity (ω_r) the sensors should be located as far from the center of the robot as possible. The first order deviation of v_2 where v_1 is placed at $v_{1,x} = 10$ and $v_{1,y} = 0$ is shown in Figures 8, 9 and 10. Looking at Figure 9, if the error in y-direction velocity ($v_{r,y}$) is minimal, if the x-positions of the sensors have the same value with different signs. The same is observed for the error of the velocity in x-direction ($v_{r,x}$).

The location of the sensors is important because the sensors provide only the linear velocities dx and dy, but not the angular velocity, $d\theta$. The change in orientation

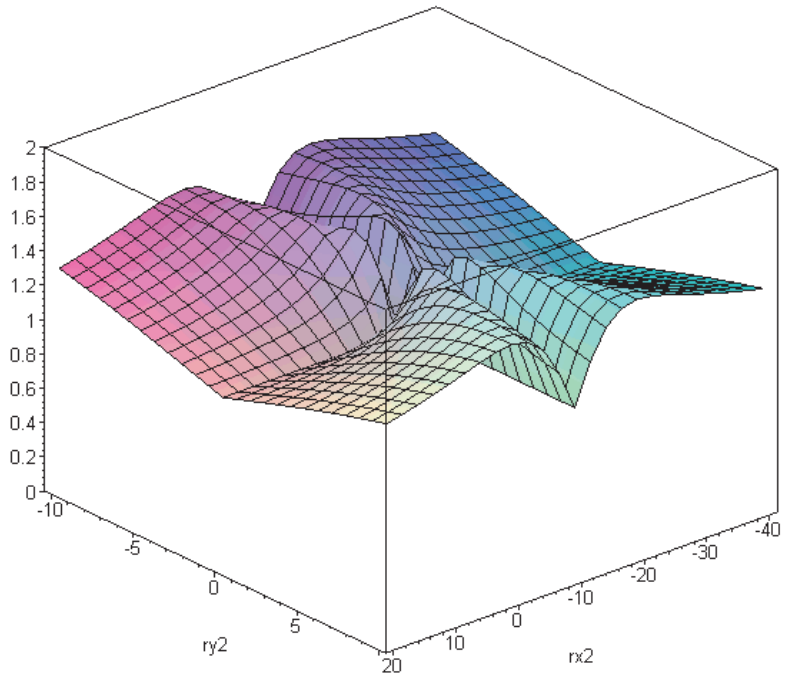

Fig. 8. Error of $v_{r,x}$ versus Sensor 2 Position

is clearly important when updating the position. Therefore, any errors in dx or dy will greatly affect the accuracy of either rigid-body method.

B. Ground Clearance

Another important aspect regarding the placement of the optical sensors is focal length. Commercial optical sensors have a narrow band of operation corresponding to the focal length. Figures 11 and 12 show the definition of the focal length and the corresponding graph showing the number of counts per inch the sensor reads.

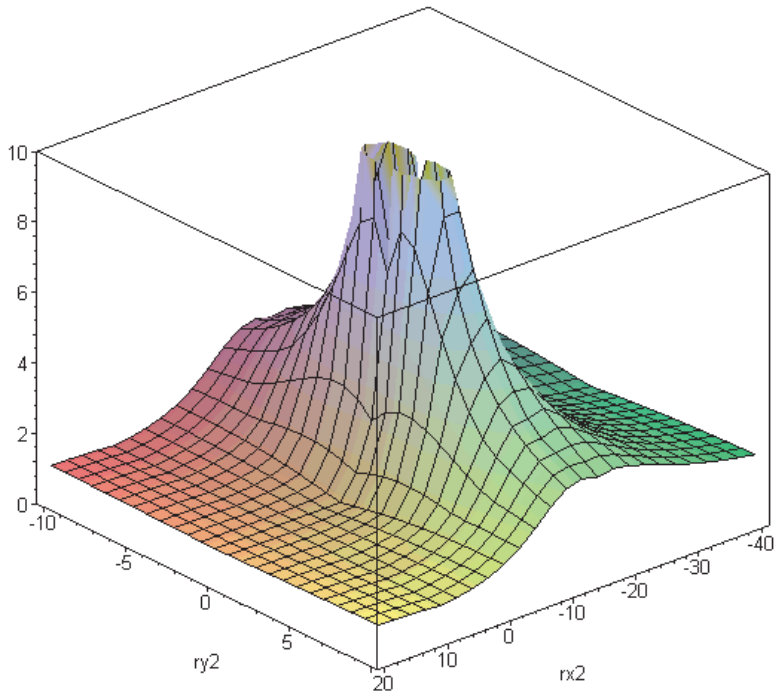

Fig. 9. Error of $v_{r,y}$ versus Sensor 2 Position

This is because in order to properly determine the movement of the sensor, the correspondence between the number of pixels in the image and a real measurable length must be known. Changing the focal length will vary this relationship, causing errors. If this constant focal length is not maintained, errors in distance will accumulate quickly.

The sensors used in the experimental section of this thesis have a nominal focal length of 2.4mm. Accurate data is obtained only if the focal length is properly maintained. Looking at Figures 11 and 12, if z is increased even a few tenths of a millimeter, inaccurate dx and dy data results. This was overcome by forcing the sensors to the ground, which tended to increase friction significantly. However, an

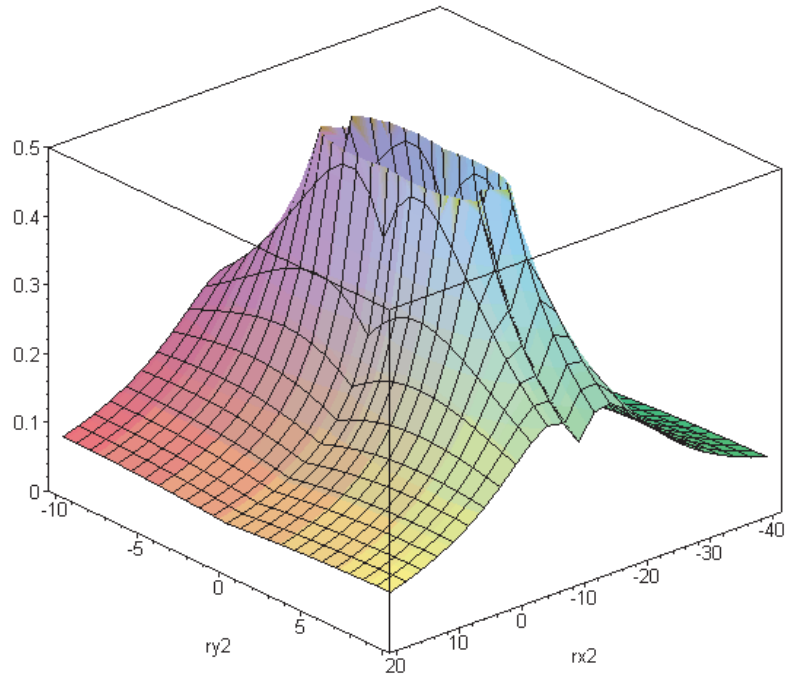

Fig. 10. Error of ω_r versus Sensor 2 Position

increase in friction can cause the wheels to slip, making it difficult to move. It is desirable to reduce friction and at the same time maintain the 2.4mm focal length recommended by the chip manufacturer.

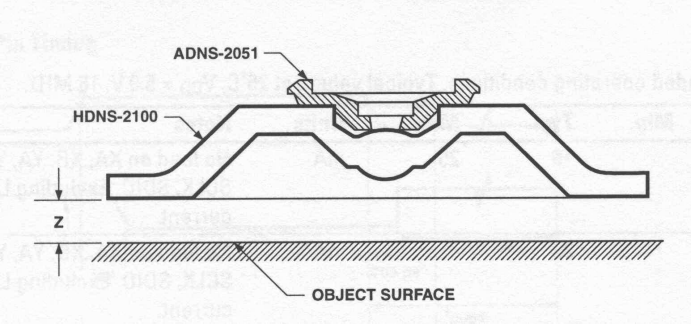

Fig. 11. Definition of z from Agilent Data Sheet

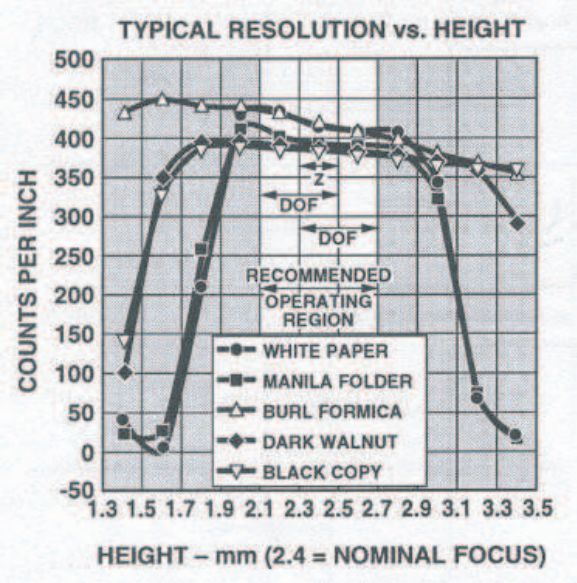

Fig. 12. Chart from Agilent Data Sheet

CHAPTER V

NAVIGATION

Navigation is defined as the art or science of maneuvering safely and efficiently from one place to another. For a mobile robot, navigation is the planning and execution of a path. A mobile robot must avoid obstacles while moving toward a goal.

With this definition in mind, there are hundreds of different methods to plan a path for a mobile robot. Two are overviewed, both of which are applicable to navigation of a differential drive mobile robot. Experimentally, potential field navigation has been implemented and is covered in Chapter VI.

A. Potential Field Navigation

In potential field navigation, the idea is that of attractive and repulsive forces being exerted between particles. The robot is attracted toward the goal point and repulsed by any obstacles. The important variables in potential field navigation are the position of the robot, position of the goal point and positions of any obstacles.

The total force exerted on the robot is equal to the vector sum of the attractive and repulsive forces. The attractive force is proportional to the distance from the goal point and the repulsive force is inversely proportional to the distance from the obstacle. Summing the attractive and repulsive forces:

$$V(q) = V_a(q) + V_r(q) \tag{5.1}$$

where $V_a(q)$ is the attractive force, $V_r(q)$ is the repulsive force and q is the configuration of the robot, namely $q_R = [x_R, y_R]^T$. Additionally, the configuration of the goal point ($q_G = [x_G, y_G]^T$) and obstacles ($q_O = [x_O, y_O]^T$) is also known.

Using a parabolic form for the attractive potential and using the distance formula,

$d(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$, (5.2) results.

$$V_a(q) = \frac{\gamma_a}{2} d^2(q_R, q_G) \tag{5.2}$$

γ_a is the attractive constant and can be freely chosen depending upon the desired attractive force. (5.2) shows that the attractive force is proportional to the distance between the robot and the goal.

The repulsive force can be derived similarly to the attractive force. As noted earlier, the repulsive force should be inversely proportional to the distance between the obstacle and the robot.

$$V_r(q) = \left(\frac{\gamma_r}{2} \cdot \frac{1}{d(q_R, q_O)} \right)^2 \tag{5.3}$$

where γ_r is the repulsive constant.

However, with this definition of the repulsive force many problems can arise. In some situations, there is no guarantee that the goal point will ever be reached. Each obstacle is usually given an area of influence or a radius in which the repulsive force acts. Figure 13 shows an example of a path and corresponding area of influence.

This leads to a second definition of the repulsive force. If the radius of influence is described as ρ_o (5.4) results.

$$V_r(q) = \begin{cases} \frac{\gamma_r}{2} \left[\frac{1}{d(q_R, q_O)} - \frac{1}{\rho_o} \right]^2 & if(d(q_R, q_O) \leq \rho_o) \\ 0 & otherwise \end{cases} \tag{5.4}$$

Looking again at (5.1), the total force exerted on the robot will be the vector sum of the corresponding repulsive and attractive forces. When multiple obstacles are present, the analysis gets somewhat more complex. Using (5.4) if multiple obstacles exert a repulsive force on the robot, all must be inside their respective areas of influ-

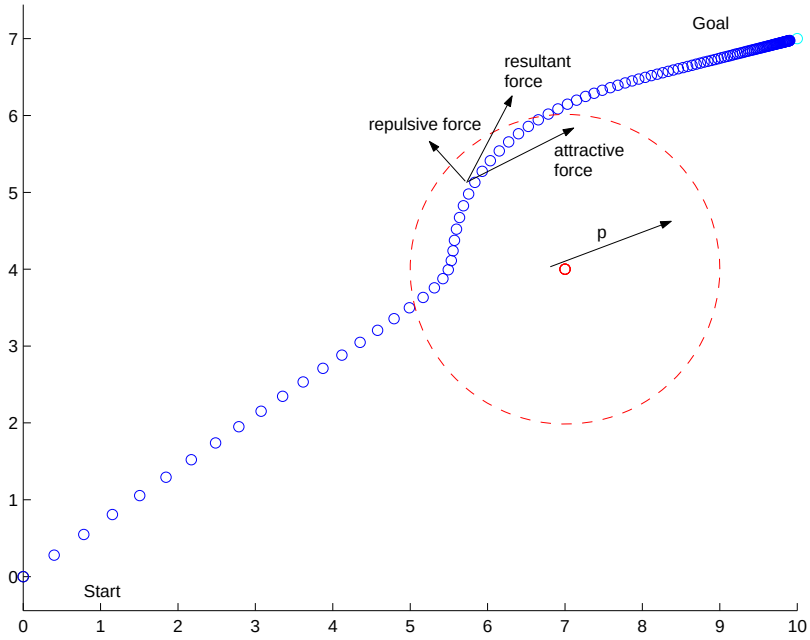

Fig. 13. Potential Field Path Showing Areas of Influence

ence, ρ_i. The corresponding vector equation will be the vector sum of the repulsive forces plus the attractive forces.

$$V(q) = V_a(q) + \sum V_{r,i}(q) \tag{5.5}$$

The potential field approach is a simple way to plan a path in the presence of obstacles. It does have many problems with potential field traps, places where the forces sum to zero. Another problem with implementation of a potential field method is that most robots are not omnidirectional. When looking at the differential drive robot mentioned, it can move in a direction parallel the the robot y direction. Therefore, depending upon the orientation of the robot, the desired velocity can be generated in only one direction. An additional term must be added to traditional potential field methods when used with a differential drive robot.

To do this, the desired heading must be found. Comparing the goal position and

the current position of the robot, the desired heading is found.

$$\theta_{desired} = \tan^{-1}(\frac{\Delta y}{\Delta x}) \tag{5.6}$$

Desired linear and angular velocity commands for the differential drive robot are generated by using the velocity obtained from the potential field method and multiplying it by the cosine of the difference between the desired heading, $\theta_{desired}$, and the current heading, θ_R. The desired ω_{robot} was obtained by using the difference between the heading and desired heading.

$$V_{desired} = C_v \cdot V_{potential} \cos(\theta_{desired} - \theta_R) \tag{5.7}$$

$$\omega_{desired} = C_\omega \cdot (\theta_{desired} - \theta_R) \tag{5.8}$$

where C_v and C_ω are proportional constants.

These velocity commands can then be used in the inverse kinematic solution found in Chapter II, which will generate desired left and right wheel velocity commands.

B. Lyapunov Based Navigation

The Lyapunov method presented is based on [7].

A second, more powerful approach to the mobile robot navigation problem is a Lyapunov based navigation function. A properly conditioned Lyapunov function has the benefit of asymptotic stability. This is especially desirable because the distance between the robot and goal as well as the angle between the robot's heading and desired heading should always decrease.

Lyapunov's Direct Method states that the equilibrium $\vec{x} = 0$ is stable if there

exists a scalar function $V(\vec{x})$, which is continuously differentiable such that:

i) $V(\vec{x})$ is positive definite

ii) $\frac{d}{dt}V(\vec{x})$ is negative semi-definite

In addition, if:

$\frac{d}{dt}V(\vec{x})$ is negative definite, then it is asymptotically stable.

This is very useful for guaranteeing that the navigation scheme will always approach the goal. Referring back to the forward kinematic solution for the differential drive robot found in Chapter II, we again have the following equation:

$$\dot{x} = -V_{robot,y} \sin\theta \tag{5.9}$$

$$\dot{y} = V_{robot,y} \cos\theta \tag{5.10}$$

$$\dot{\theta} = \omega_{robot} \tag{5.11}$$

where V_{robot}, ω_{robot} are the forward and angular velocities respectively and θ is the orientation of the robot. (Refer again to Figure 4.)

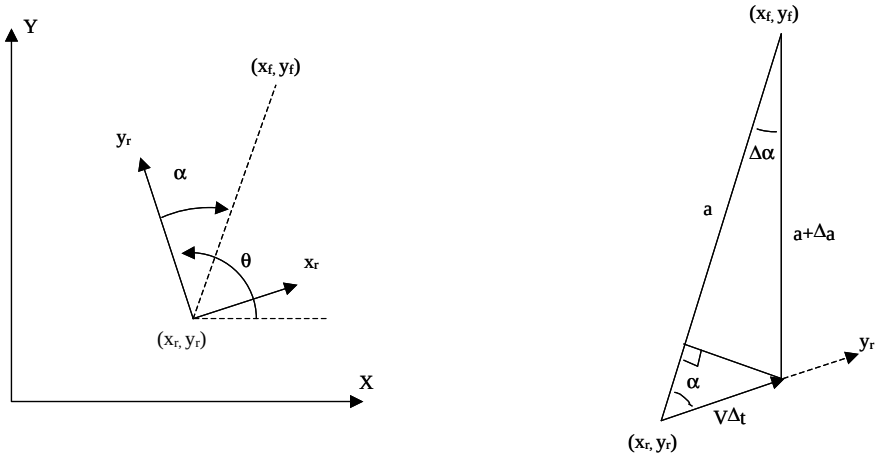

Fig. 14. Coordinates Used in Lyapunov Derivation

The configuration vector can be defined as $z = [a, \alpha]^T$, where a is the distance between the robot and the goal position and α is the difference of the angle between the horizontal and the vector to the goal point minus the heading of the robot, θ_R. This is shown in Figure 14. A non-linear stable controller is designed based on a Lyapunov function. The function should yield a solution which forces a and α to zero.

Solving for the update laws, $\dot{a}$ and $\dot{\alpha}$, the following equations are developed. Taking the limit of $\frac{\Delta a}{\Delta t}$ as $\Delta t \to 0$:

$$\dot{a} = V \cdot \Delta t \cos(\alpha) = a - (a + \Delta a)\cos(\Delta\alpha) \tag{5.12}$$

as $\Delta t \to 0$, $\Delta\alpha \to 0$.

Solving for $\dot{a}$:

$$V \cdot \Delta t \cos(\alpha) = -\Delta a \tag{5.13}$$

$$so : \frac{\Delta a}{\Delta t} = -V\cos(\alpha) = \dot{a} \tag{5.14}$$

Similarly, $\Delta\alpha$ can be derived. Taking the limit of $\frac{\Delta\alpha}{\Delta t}$ as $\Delta t \to 0$:

$$\dot{\alpha} = V \cdot \Delta t \sin(\alpha) = (a + \Delta a)\sin(\Delta\alpha) \tag{5.15}$$

If $\Delta\alpha \simeq 0$, then $\sin\Delta\alpha \simeq \Delta\alpha$. Substituting this into the previous equation:

$$V \cdot \Delta t \sin(\alpha) = (a + \Delta a) \cdot \Delta\alpha \tag{5.16}$$

Using the assumption that two very small quantities multiplied add little to the solution, $\Delta\alpha\Delta a \simeq 0$, this further simplifies the equation to the following:

$$a \cdot \Delta\alpha + \Delta a \cdot \Delta\alpha = a \cdot \Delta\alpha \tag{5.17}$$

Finally, solving for $\dot{\alpha}$:

$$\frac{\Delta\alpha}{\Delta t} = \frac{V\sin(\alpha)}{a} = \dot{\alpha} \tag{5.18}$$

The dynamics of z can now be written in (5.19). Since the previous equations only took the linear displacement into account, ω, must be subtracted in order to incorporate its contribution.

$$\dot{z} = \begin{bmatrix} \dot{a} \\ \dot{\alpha} \end{bmatrix} = \begin{bmatrix} -V\cos(\alpha) \\ \frac{V\sin(\alpha)}{a} - \omega \end{bmatrix} \tag{5.19}$$

A Lyapunov candidate is then given by:

$$L = \frac{1}{2}a^2 + \frac{1}{2}\alpha^2 \geq 0 \tag{5.20}$$

Solving for $\dot{L}$:

$$\dot{L} = a\dot{a} + \alpha\dot{\alpha} \tag{5.21}$$

For asymptotic stability, L, should be positive definite and its time derivative, $\dot{L}$, should be negative definite. Using this knowledge, the following relationships can be written:

$$a\dot{a} = -V\cos(\alpha)a \leq 0 \tag{5.22}$$

$$\alpha\dot{\alpha} = \frac{V\sin(\alpha)}{a}\alpha - \omega\alpha \leq 0 \tag{5.23}$$

This aids in the discovery of an appropriate control law for V and ω. Using (5.22), V should have an additional $\cos(\alpha)$ term so that when multiplied, $\cos^2(\alpha)$ emerges, make it greater than zero. This leads to the following definition of V:

$$V = k_1 a\cos(\alpha) \tag{5.24}$$

Where k_1 is a positive proportional constant. Plugging this definition of V into (5.23):

$$\alpha\dot{\alpha} = \frac{k_1 a\cos(\alpha)\sin(\alpha)}{a}\alpha - \omega\alpha = k_1\cos(\alpha)\sin(\alpha)\alpha - \omega\alpha \tag{5.25}$$

It would be convenient to cancel out the $\sin(\alpha)\cos(\alpha)$ term and it is also necessary to multiply by α, since α can be less than zero. This yields the following definition for ω.

$$\omega = k_2\alpha + k_1\cos(\alpha)\sin(\alpha) \tag{5.26}$$

Where k_2 is a second proportional constant. Plugging the definitions of the controllers into (5.19), the following emerges:

$$\dot{a} = -(k_1\cos^2(\alpha))a \tag{5.27}$$

$$\dot{\alpha} = k_1 a\cos^2(\alpha)\frac{\sin(\alpha)}{a} - k_2\alpha + k_1\sin(\alpha)\cos(\alpha) = -k_2\alpha \tag{5.28}$$

Finally, plugging these controller definitions into (5.21):

$$\dot{L} = -(k_1\cos^2(\alpha))a^2 - k_2\alpha^2 \leq 0 \forall z \tag{5.29}$$

Using these definitions for coordinates, $z = [a, \alpha]^T$, and control laws for V and ω, the Lyapunov candidate, L, will always be positive definite, and $\dot{L}$ will always be negative definite, satisfying the conditions of the Lyapunov Direct Method.

Investigating Lyapunov based navigation, simulations were performed in Matlab. Figures 15, 16 and 17 show a simulation for Lyapunov navigation of a differential drive mobile robot. The robot begins at $[0, 0, -225°]^T$ and the goal point is located at $[0, 0, undefined]^T$.

In this case, the robot moves backward first, decreasing both a and α and then

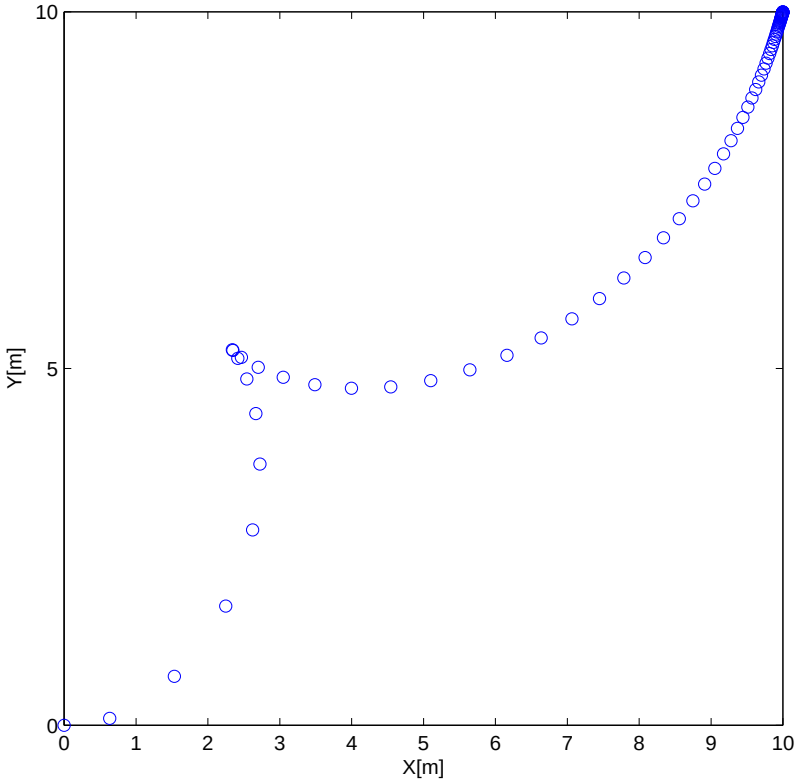

Fig. 15. Lyapunov Based Navigation - Movement

turns to further decrease α and heads toward the goal moving forward.

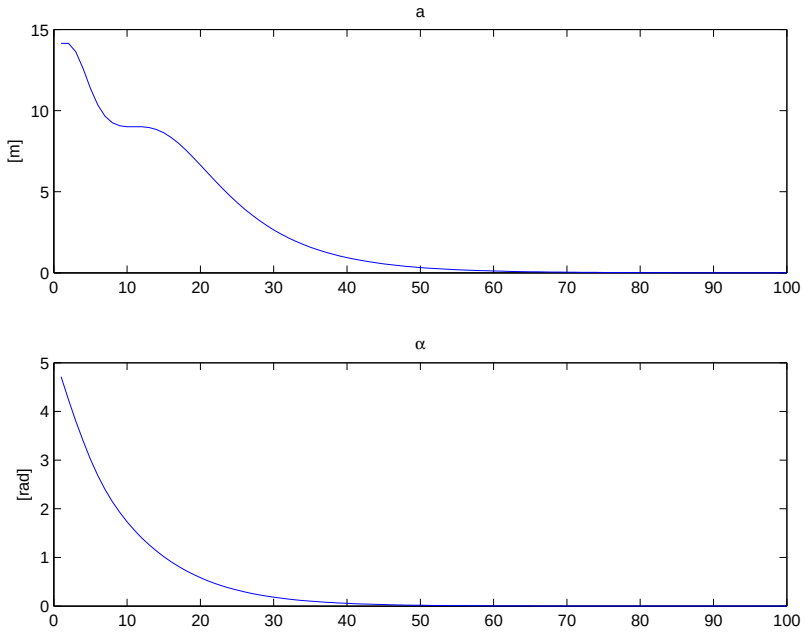

Fig. 16. Lyapunov Based Navigation - α and a

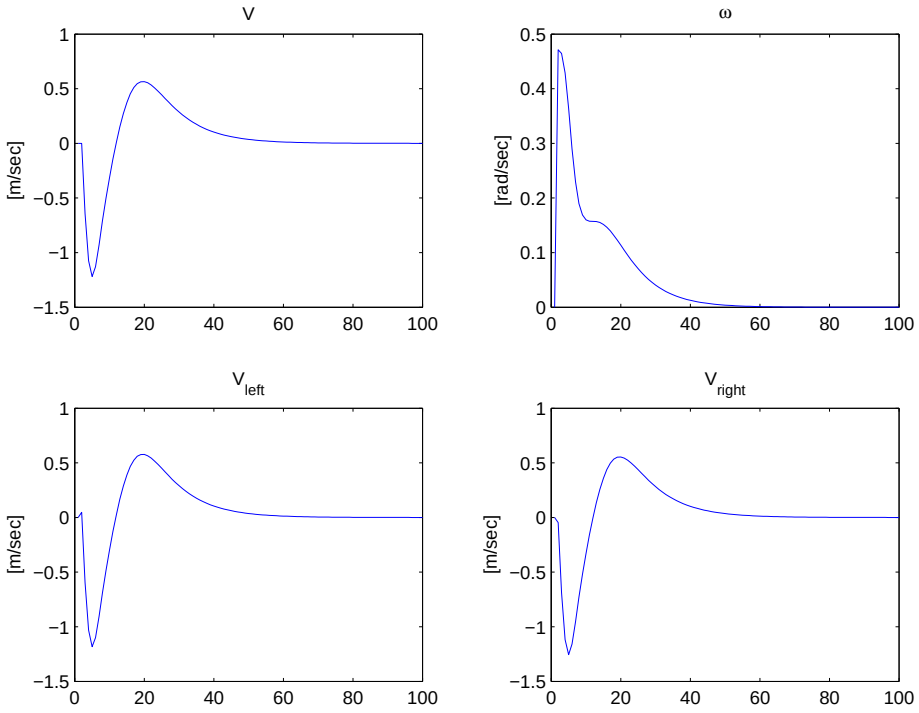

Fig. 17. Lyapunov Based Navigation - Velocities

CHAPTER VI

EXPERIMENTAL VALIDATION

A. Sensor Specifications

The optical sensors used were both Agilent ADNS-2051 Optical Mouse Sensors. This CMOS chip allows fast, accurate, optical sensing of microscopic images. The sensor is capable of 800 counts per inch while moving at up to 14 inches per second. Successive images are used to calculate the Δx and Δy values at up to 2300 frames per second.

A microcontroller is used which communicates directly with the ADNS-2051. This chip communicates with the ADNS-2051, setting modes and communicating with the PC or other device using the PS-2 standard.

The optical sensor uses successive images to interpret the movement between images. Images are taken of a point on the ground directly underneath the sensor. The images are 16×16 pixels and represent a microscopic area.

B. Robot Specifications

In order to test the algorithms presented in this paper, a robot system was designed and built. A modular design approach was implemented. Each module is responsible for one part of the overall operation of the robot. This type of design allows testing of various configurations as well as allowing easy replacement of any single system.

1. Hardware

The REX 99 from Zagros Robotics was chosen as a research test-bed. The mobile robot includes a 12 inch, round plastic base, two 1 amp 12 Volt DC geared motors, two castor wheels, two 6 inch diameter rubber wheels, and two HEDS-5500 encoders

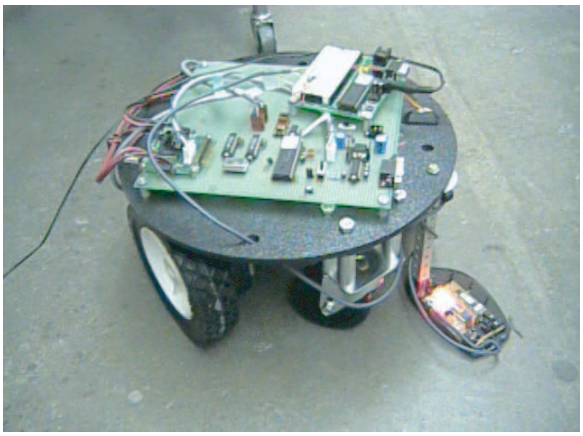

Fig. 18. Mobile Robot with Optical Sensors

attached to the drive wheel axles.

The differential driven wheels allow turns to be made in place. This symmetric design offers high maneuverability. The maximum straight line speed is approximately 78 feet per minute (50 revolutions per minute of the drive wheels). No mechanical modifications were made to the REX 99 kit except those required for mounting the sensors. The robot with an optical sensor installed is shown in Figure 18.

The robotic base consists of a microcontroller circuit with various communication and electronic chips, two motors, two 1 amp motor controllers, two HEDS-5500 wheel encoders and two Agilent HCTL-2020 encoder-decoder chips.

The sensor system consists of a microcontroller kit and two optical sensors. The microcontroller kit includes a microcontroller, serial communication chip, 5 volt power supply, and other small electronic parts. The optical sensors were inexpensive optical mice. The microcontroller interfaces with the two optical sensors and transmits the data to the PC using serial communication.

2. Software

The software consists of three independent systems which must operate congruously in order to function. These systems are the robot system, the sensor system and a PC based controller. For the robot and sensor systems, C was used exclusively. Both are based on PIC 16F877 microcontrollers running at 10 MHz. The PC controller is implemented using Java. The operation of the system is outlined in Figure 19. Appendix B has a listing of Java source code.

Software design requirements for the robotic base were to control the speed of two motors to desired velocities and output the position (or velocity) of both the left and right wheels. A real-time system, one that operates as things happen, was desired. This required an interrupt scheme to be used. The robotic base receives wheel commands from the PC and controls the difference in wheel positions between successive interrupts.

C. Intermediate Localization

Experiments were performed comparing intermediate localization (positional estimates) using dead-reckoning, the single sensor and the multiple sensor rigid-body method. Experiments were performed where no kinematic violations occurred to test the accuracy of localization. Experiments were also performed where kinematic violations were forced.

In experiments where kinematic constraints were upheld ($V_{robot,x} = 0$ and no wheel slip) as in Figure 20, little performance difference can be seen between the methods. When the kinematic constraints were intentionally violated with an outside disturbance as in Figure 21, however, only the multiple sensor method gave good results.

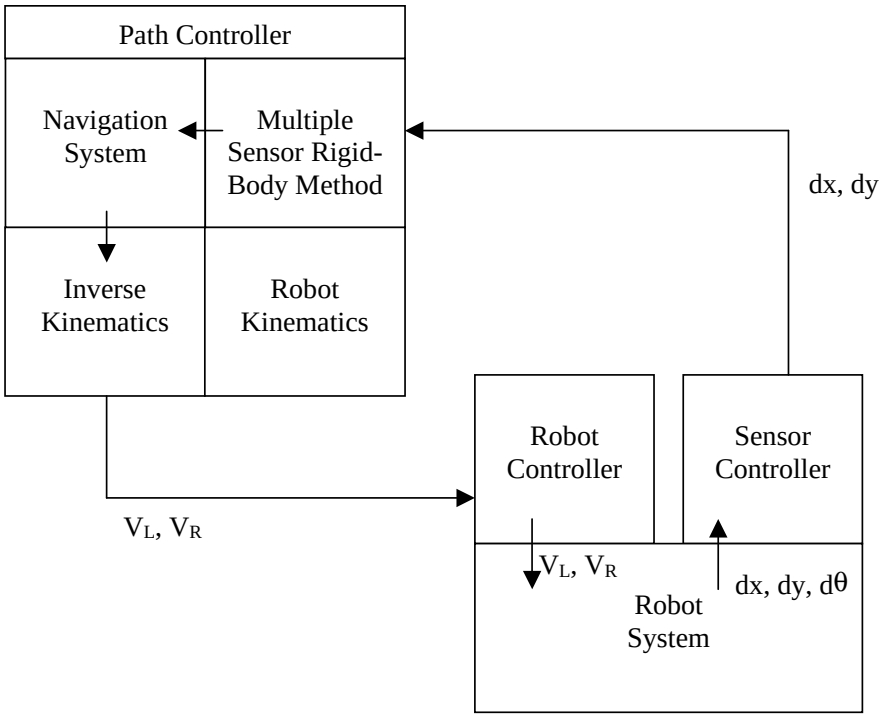

Fig. 19. Interaction Between Objects in Java

The results of the single sensor method are interesting. It would seem that when kinematic disturbances are encountered, the method gives unreliable results. However, this should be expected because the kinematic constraint that is required for a solution to exist in the first place.

Looking at a specific case, if the sensor were located at some distance along the positive y robot axis, what would happen if a displacement along the x axis occurred? The answer is that the sideways motion of the sensor would be interpreted as rotational motion of the center of the robot.

Additionally, if the underdetermined pseudo-inverse solution is used (see Chapter III), there is little accuracy gained. This solution does not use the constraint, but simply weights the solution so that velocity in the x direction is reduced. This method was investigated to determine if an accurate estimate would result if the constraint

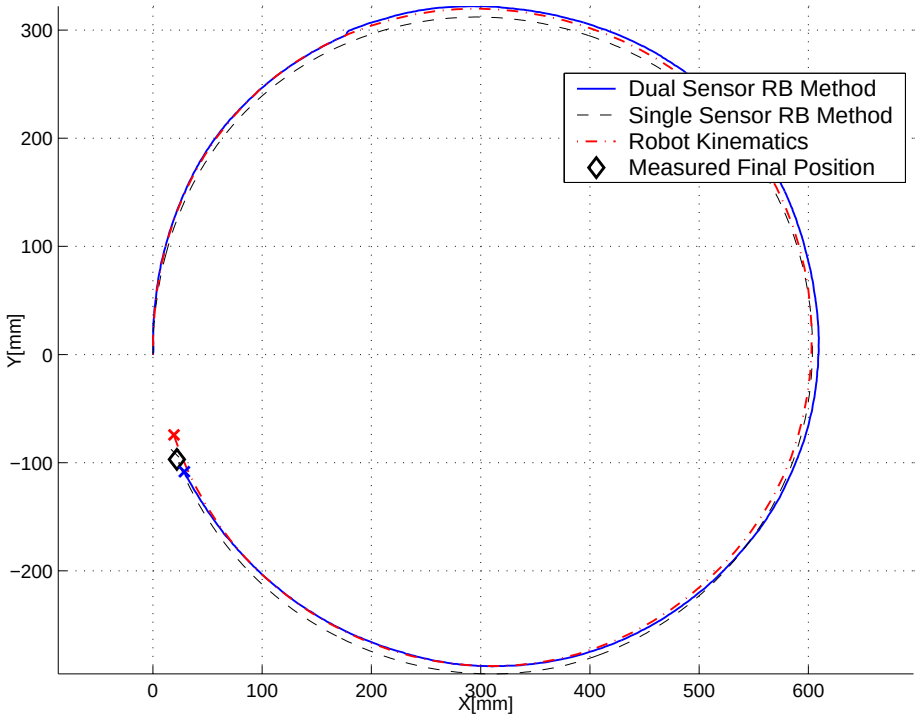

Fig. 20. Localization Methods Compared with Kinematics Upheld

$V_{r,x} = 0$ is removed. Figure 22 shows how changing the weighting of $V_{r,x}$ effects the estimated path compared to the dual-sensor method.

D. Potential Field Based Navigation

In order to test the effectiveness of the intermediate estimation method described, navigation tests were performed. The single and multiple sensor rigid-body models and dead-reckoning were compared under various conditions.

A potential field based approach was used to generate velocity commands for the mobile robot to follow. An online method was used where the desired V_{robot} and ω_{robot} were generated depending upon the current robot location/orientation and the location of the goal point. This was described in Chapter V.

A slight modification to the method was used. This is because of the non-

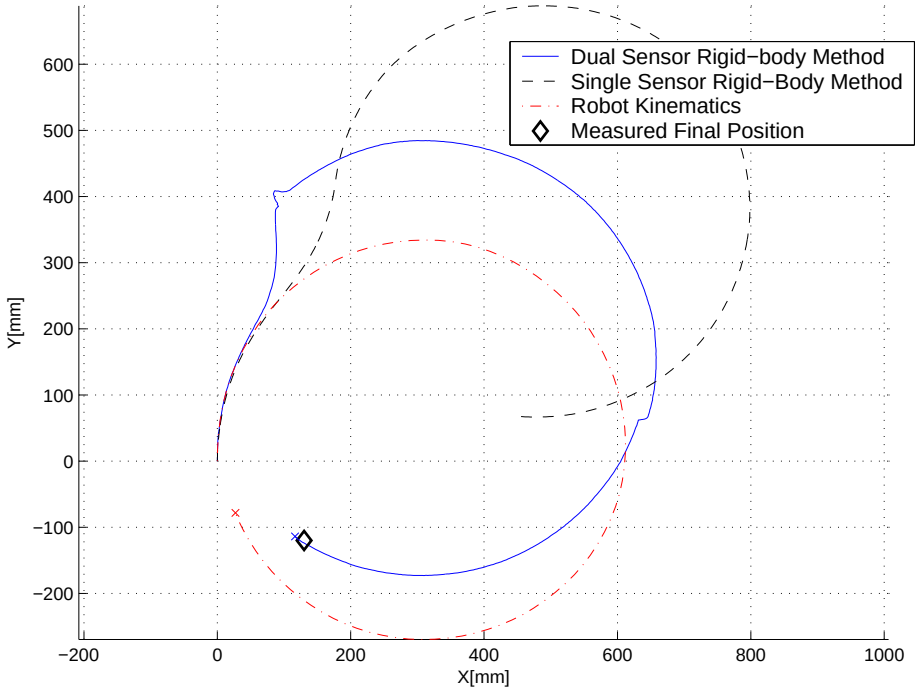

Fig. 21. Localization Methods Compared with Kinematic Constraints

holomonic (the robot can not move in all directions) constraints that are present with a differential drive mobile robot. The method was to make the robot turn towards the goal point before approaching it. The following relationships were used:

$$V_{desired} = k_v V_{potential} \cos(\alpha) \tag{6.1}$$

$$V_{desired} = k_\omega V_{potential} \alpha \tag{6.2}$$

Where k_v and k_ω are proportional constants and $V_{potential}$ is the potential velocity generated by the potential field method. Using this approach, results were satisfactory. Also, obstacles were assumed not to be present, but could easily be added.

In tests where no wheel slip or kinematic violations occurred all three methods were similarly accurate. The largest performance difference of the multiple-sensor method can be see in the presence of kinematic violations, such as wheel slip. Figure

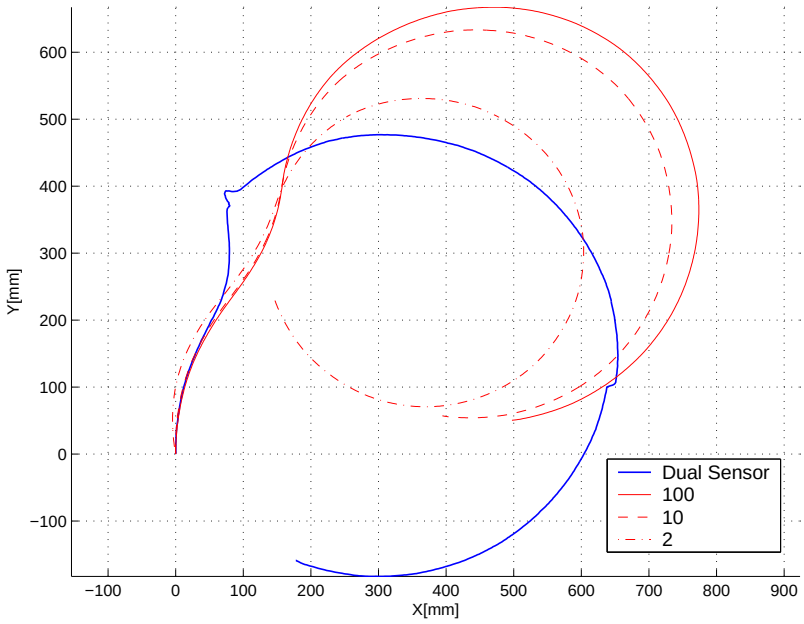

Fig. 22. Weighted Pseudo-inverse for the Underdetermined Single Sensor Case

23 shows an actual test run where wheel slip was forced. As one can see, the robot has a sudden change in orientation at the beginning of movement. The multiple-sensor method is able to detect this movement, which allows the potential field based navigation method to adjust its course accordingly. Even when kinematic constraints are blatantly violated, as in Figure 24 where the robot was pushed off course, the multiple sensor method still detects the proper change in position and orientation so that the robot successfully reaches the goal.

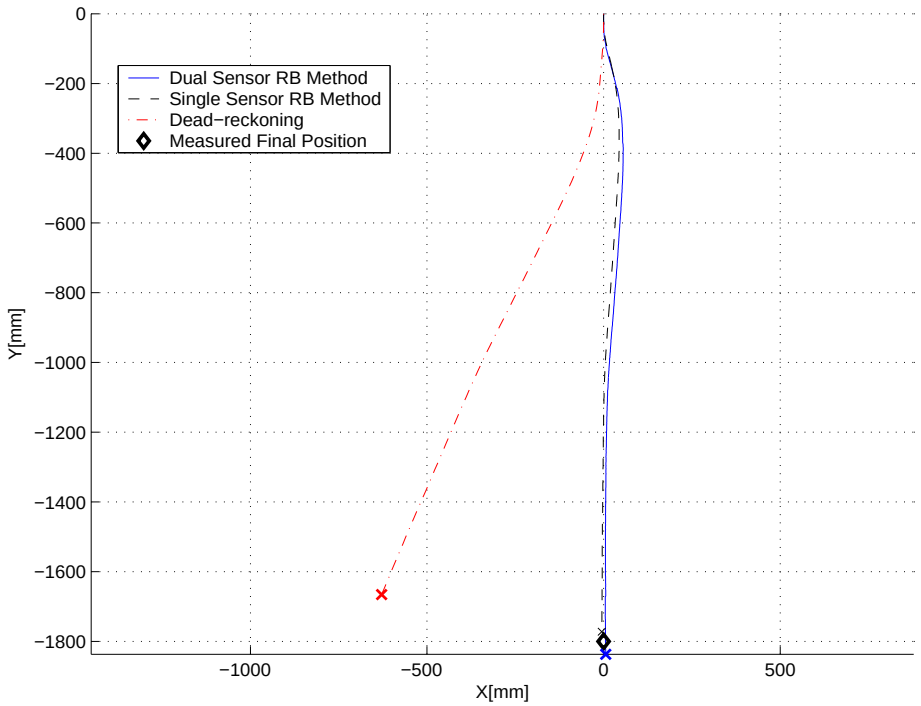

Fig. 23. Navigation with Wheel Slip

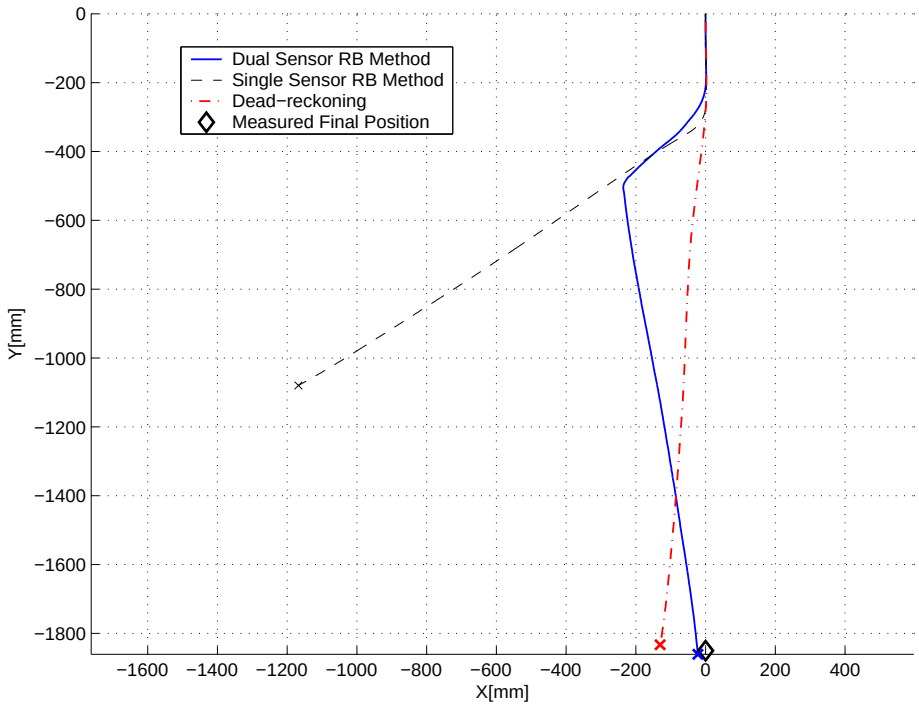

Fig. 24. Navigation with Kinematic Violation

CHAPTER VII

CONCLUSION

Using optical flow techniques, a method has been derived which can accurately give intermediate position estimates. The location of the sensors has been investigated in order to minimize potential errors introduced by incorrect sensor readings. Experimentally, the multiple sensor method has proven to be more accurate and more robust than dead-reckoning and the single-sensor method. Both systematic and non-systematic errors can be detected and a good estimate of location and orientation can be maintained.

REFERENCES

[1] J. Borenstein, "Internal Correction of Dead-reckoning Errors With the Smart Encoder Trailer," *Proc. of the IEEE/RSJ/GI Int. Conf. on Intelligent Robots and Systems '94*, vol. 1, pp. 127–134, Sept. 1994.

[2] K. Nagatani, S. Tachibana, M. Sofne, and Y. Tanaka, "Improvement of Odometry for Omnidirectional Vehicle Using Optical Flow Information," *Proc. 2000 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, vol. 1, pp. 468–473, Oct. 2000.

[3] B. W. Parkinson and J. J. Spilker, *Global Positioning System: Theory and Application,* vol. 2, Washington, D.C.: American Institute of Aeronautics and Astronautics, 1996.

[4] R. Jirawimut, P. Ptasinski, V. Garaj, F. Cecelja, and W. Balachandran, "A Method for Dead Reckoning Parameter Correction in Pedestrian Navigation System," *Proc. of the 18th IEEE Instrumentation and Measurement Technology Conference*, vol. 3, pp. 1554–1558, May 2001.

[5] B. Barshan and H. F. Durant-Whyte, "Inertial Navigation Systems for Mobile Robots," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 3, pp. 328–342, June 1995.

[6] W. L. Brogan, *Modern Control Theory*, Upper Saddle River, New Jersey: Prentice Hall, Inc., 1991.

[7] G. Ramirez and S. Zeghoul, "A New Local Path Planner for Nonholonomic Mobile Robot Navigation in Cluttered Environments," *IEEE International Conference on Robotics and Automation,* vol. 1, pp. 2058-2063, April 2000.

[8] B. Horn, *Robot Vision*, Cambridge, Massachusetts: MIT Press, 1986.

[9] L. G. Shapiro, and G. C. Stockman, *Computer Vision*, Upper Saddle River, New Jersey: Prentice-Hall Inc., 2001.

[10] N. Molton, S. Se, J. M. Brady, D. Lee, and P. Probert "A Stereo Vision-Based Aid for the Visually Impaired," *Image and Vision Computing*, vol. 16, no. 4, pp. 251-63, April 1998.

APPENDIX A

OPTICAL FLOW

For more information regarding optical flow, see [8], [9], or [10].

Optical flow is the estimation of motion given two or more images. Optical flow is usually computed by comparing image intensities of two successive images. In order for any optical flow method to work, the two images must have much of the same visual material present.

Using a CCD camera, optical flow for a motion sequence was computed. It was tested on images 8×8 pixels in size. The two images in Figure 25 represent a simple motion sequence. The right image, $I(x, y, t)$, represents the image intensity at time, t and spatial coordinates, x and y. The left image, $I(x + dx, y + dy, t + dt)$, represents the synthetic motion of dx and dy, and dt change in time. Displacing the original image in the horizontal and vertical direction pixel-wise produced a synthetic motion. In the figure below, the left image was displaced up one pixel and to the right one pixel to produce the right image.

Fig. 25. Two Images

Using a standard image processing technique, each image was smoothed by convolving the image with a 2D filter, first in x and then y. A gaussian filter of standard deviation equal to 3 was used and the image contrast was enhanced via histogram equalization. The resulting filtered images are seen in Figure 26.

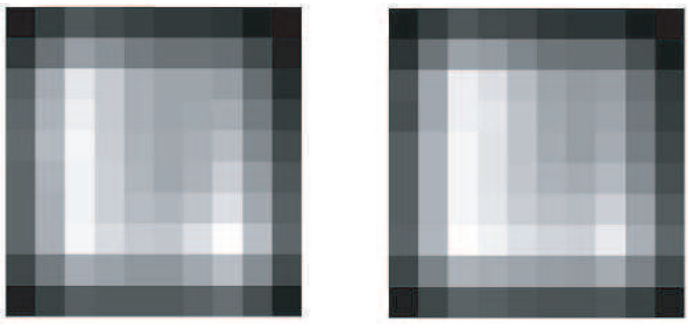

Fig. 26. Images Filtered by Convolution with a Gaussian Mask

Gradients are then estimated in x, y and t for each pixel in the current image. Each image is convolved in x, y and t with a 2×2 differencing mask to compute the spatial gradients I_x, I_y, and the temporal gradient, I_t, for each pixel.

Spatial gradients in x and y can be computed within the current frame. However, the temporal gradient, I_t, is computed using the current image, and the previous image. The gradients are then smoothed. Spatial filtering attenuates noise in the estimation of the spatial image gradient; while temporal filtering prevents aliasing in the time domain.

Assuming a small change in x, y and t there is no change in intensity. This constant image intensity assumption yields a good approximation of the normal component of the motion field.

$$f_x u + f_y v + f_t = 0 \tag{A.1}$$

where f_x, f_y, f_t are spatiotemporal derivatives, and u, v are the optical flow components.

A concrete example can be observed by looking at two images, F, the second image, and G, the first image. Considering that a point on the first image $G(x, y)$ has the same intensity as a point $F(x + u, y + v)$ on a second image, the movement of the point between the two images, u and v, can be computed. Defining the error:

$$\varepsilon = \Sigma \left[F(x + u, y + v) - G(x, y)\right]^2 \tag{A.2}$$

$$\varepsilon = \Sigma \left[F(x, y) + F_x(x, y) \cdot u + F_y(x, y) \cdot v - G(x, y)\right]^2 \tag{A.3}$$

where:

$$F_x = \frac{\partial F(x, y)}{\partial x} \tag{A.4}$$

$$F_y = \frac{\partial F(x, y)}{\partial y} \tag{A.5}$$

$$F_t = \frac{\partial F(x, y)}{\partial t} = \frac{G(x, y) - F(x, y)}{1} \tag{A.6}$$

Rewriting ε:

$$\varepsilon = \Sigma \left[F(x, y) - G(x, y) + F_x(x, y) \cdot u + F_y(x, y) \cdot v\right]^2 \tag{A.7}$$

$$\varepsilon = \Sigma \left[-F_t(x, y) + F_x(x, y) \cdot u + F_y(x, y) \cdot v\right]^2 \tag{A.8}$$

The goal is to find u and v, while minimizing the error, ε. So, the partial derivative of ε with respect to u and v should be minimized.

$$\frac{\partial \varepsilon}{\partial u} = 0 \tag{A.9}$$

$$\frac{\partial \varepsilon}{\partial v} = 0 \tag{A.10}$$

This results in (A.11) and (A.12).

$$\frac{\partial \varepsilon}{\partial u} = 2\Sigma[F_x(x,y)u + F_y(x,y)v - F_t]\,{F_x}^2 \tag{A.11}$$

$$\frac{\partial \varepsilon}{\partial v} = 2\Sigma[F_x(x,y)u + F_y(x,y)v - F_t]\,{F_y}^2 \tag{A.12}$$

and

$$\Sigma F_x(x,y)^2 u + F_x(x,y)F_y(x,y)v = \Sigma F_t(x,y)F_x(x,y) \tag{A.13}$$

$$\Sigma F_x(x,y)F_y(x,y)u + F_y(x,y)^2 v = \Sigma F_t(x,y)F_y(x,y) \tag{A.14}$$

Which can be rewritten in matrix form.

$$\begin{bmatrix} \Sigma F_x^2 & \Sigma F_x F_y \\ \Sigma F_x F_y & \Sigma F_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} F_t F_x \\ F_t F_y \end{bmatrix} \tag{A.15}$$

Solving for u and v:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \Sigma F_x^2 & \Sigma F_x F_y \\ \Sigma F_x F_y & \Sigma F_y^2 \end{bmatrix}^{-1} \begin{bmatrix} F_t F_x \\ F_t F_y \end{bmatrix} \tag{A.16}$$

There are two unknowns, u and v, in this equation. However, instead of using one equation for one pixel, we consider a small neighborhood around a pixel and get an over-constrained system, which is solved using a least squares fit. Considering a 2×2 neighborhood, and assuming optical flow to be constant in this neighborhood, we get 4 optical flow equations

$$\begin{aligned} f_{x1}u + f_{y1}v &= -f_{t1} \\ &\vdots \\ f_{x4}u + f_{y4}v &= -f_{t4} \end{aligned} \tag{A.17}$$

The system, written in matrix form, is shown below

$$\begin{bmatrix} f_{x1} & f_{y1} \\ f_{x2} & f_{y2} \\ f_{x3} & f_{y3} \\ f_{x4} & f_{y4} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -f_{t1} \\ -f_{t2} \\ -f_{t3} \\ -f_{t4} \end{bmatrix} \tag{A.18}$$

or

$$AU = F_t \tag{A.19}$$

$$U = \left(A^T A\right)^{-1} A^T F_t \tag{A.20}$$

A linear system is constructed as shown in the above equations, to contain all the gradient information. The 4×2 matrix containing gradient information for x and y and 4×2 matrix containing the gradient information for t can be solved to yield the optical flow components, u and v.

Flow vectors for different motion sequences were computed and the algorithm's accuracy is investigated. The computed optical flow was compared to the actual displacement of pixels between images.

The algorithm accurately computes the optical flow for motion sequences moving at 1 pixel/image. Visually, the flow vectors appear to be accurate. Figure 27 shows the flow field for a one pixel displacement in the x and y directions.

The algorithm is able to detect the corresponding motion, distinguishing between flow in x and y. When pixel displacement is greater than 1 pixel, the algorithm fails to correctly determine the actual displacement in an 8×8 pixel image. It is therefore important to sample the image frequently so that only small, one-pixel changes occur

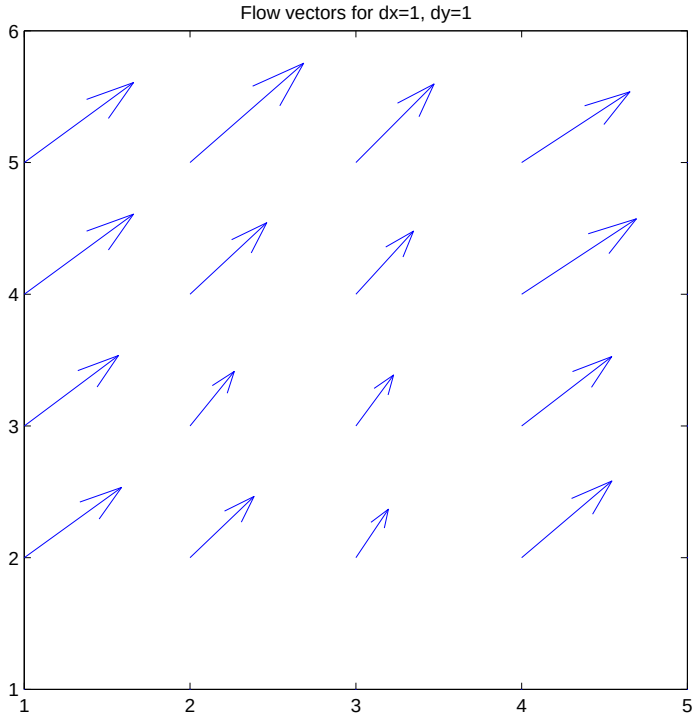

Fig. 27. Estimated Optical Flow

between successive images. Rotation of the image cannot be accurately detected due to the small number of pixels.

If increasing the image size, the rotation of the image can be computed. Figure 28 shows the vector field for a small rotation.

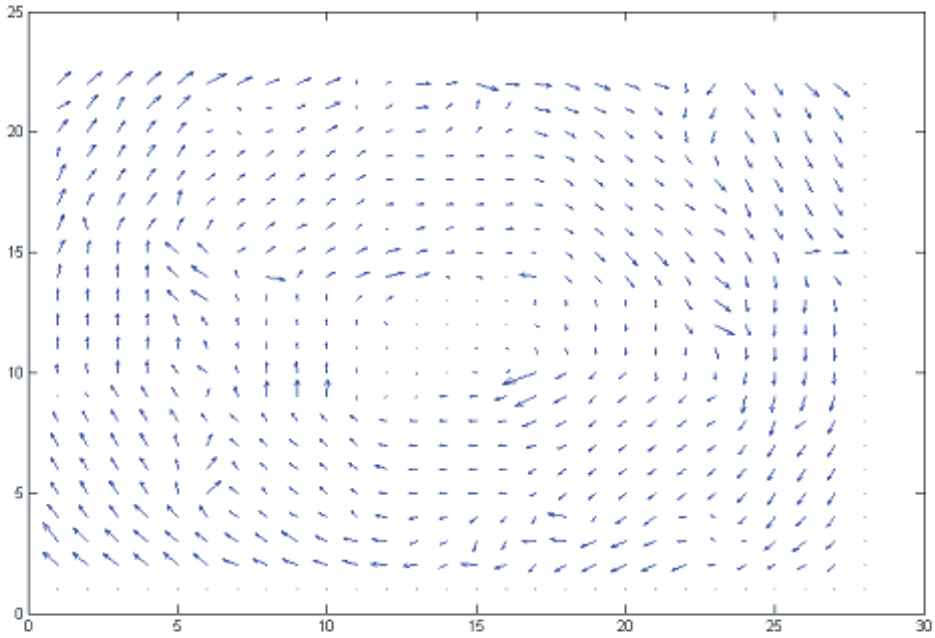

Fig. 28. Large Image Showing Rotation

APPENDIX B

JAVA SOURCE

```
// author: David Sorensen
// Texas A&M Robotics Lab
// PathController.java

import javax.swing.*; import java.util.Date; import
java.awt.event.*;

public class PathController extends JPanel
  implements ActionListener {

  // Robot and Mouse Controllers
  private RobotController rc;
  //private MouseController mc;
  private DualMouseController mc;
```

```
// date for tracking the current running time
private Date startDate;

// timer for control, etc.
private Timer timer;
private int timerDelay;

// desired location
private double xDes,yDes;

// control constants
private double vConstant,omegaConstant;
private double vDesired,omegaDesired;

public PathController() {
  timerDelay = 0;

  // millimeters
  xDes = 0;
  yDes = -2000;

  vConstant = .0002;
  omegaConstant = 80;
  rc = new RobotController("COM1");
  rc.setParameters(148/2,249); // mm - measured
  rc.goExclamation();

  mc = new DualMouseController("COM2");
  mc.setParameters1(-78,203,177.25/(180/Math.PI));
  mc.setParameters2(72,-199,-3.5/(180/Math.PI));
  mc.setAInverse();
  mc.goExclamation();

  timer = new Timer(100,this);
  timer.setInitialDelay(2000);
  timer.start();
}

public void actionPerformed(ActionEvent e){
  System.out.println("time: "+
   (double)timerDelay/(1000/timer.getDelay()) +"s");
  timerDelay++;

  // navigation goes here...
  // calc distance from current location to goal
  // generate desired velocity, omega
  this.vDesired = this.findDistance()
   *this.findDistance()*this.vConstant
   *Math.cos(this.findAngle());
  this.omegaDesired = this.findAngle()
   *this.vDesired; // scale omega according to v
```

```
    System.out.println("vDesired = " +vDesired +
     "  omegaDesired = "+omegaDesired);

    // generate desired VelLeft, VelRight wheel commands
    this.inverseKinematics();
  }

  private double findDistance(){
    return Math.sqrt((this.xDes-this.mc.getX())*
                     (this.xDes-this.mc.getX())+
                      (this.yDes-this.mc.getY())*
                      (this.yDes-this.mc.getY()));
  }
  private double findAngle(){
    return Math.atan2((this.yDes-this.mc.getY()),
    (this.xDes-this.mc.getX()))+Math.PI/2-this.mc.getTheta();
  }

  // find the relation between (V,omega) -> (Vr, Vl)
  // not true inverse kinematics!!!
  private void inverseKinematics(){
    double vL=-this.vDesired-this.omegaDesired;
    double vR=-this.vDesired+this.omegaDesired;
    double max = Math.max(Math.abs(vL),Math.abs(vR));
    double divisor=1;
    if(max > 9){
      divisor = max/9;
    }
    if(vL > 0)
      vL = 0;
    if(vR > 0)
      vR = 0;
    if((vL >= -4) \&\& (vR >= -4)){
      vL = 0;
      vR = 0;
    }

    System.out.println((int)Math.floor(vL/divisor)+
     " "+(int)Math.floor(vR/divisor));
    rc.setWheelSpeeds((int)Math.floor(vL/divisor),
     (int)Math.floor(vR/divisor));
  }

  public static void main(String[] args) {
    PathController pathController1 = new PathController();
  }
}


// author: David Sorensen
// Texas A&M Robotics Lab
```

```
// RobotController.java

import javax.comm.*; import java.io.*;
import java.util.ArrayList;
import java.util.Date;
import java.lang.Math;


/*
  This class takes care of the serial communication
  with the robot controller.
  It also takes care of the Kinematic equations
  necessary to determine the position of the robot.

  Inputs:
    Robot Parameters- wheel diameter, wheel base, ...
    Serial Port Name- identifying the serial port
    Serial stream- taken care of inside the class

  Outputs:
    Current Position- x location, y location, orientation
    Raw Data- Array of all data from the microcontroller
    Path- path of the robot (array of positions and times)
    Wheel Speeds- array of wheelspeeds and
    when they were passed to the wheels */

public class RobotController implements
  Runnable, SerialPortEventListener, Serializable{
  //public static int MAX_SPEED = 15;
  private static double INTERRUPT_TIME = 0.1;

  // for state of robot
  protected boolean isRunning;

  // track the starting time
  protected Date startDate;

  // track if there are new speeds
  private boolean hasNewSpeeds;

  // for saving files
  protected File rawDataFile;
  protected File pathDataFile;
  protected FileWriter rawWriter;
  protected FileWriter pathWriter;

  // communications port info
  static CommPortIdentifier commId;
  protected SerialPort sPort;

  // comm Port string COM1 for example
```

```
protected String commString;

// streams for communication
protected InputStream inStream;
protected OutputStream outStream;

// track the location of the robot
protected double xLoc;
protected double yLoc;
protected double theta;

// robot parameters
private double wheelBase;
private double wheelRadius;


// left and right wheelSpeeds
private int leftWheelSpeed;
private int rightWheelSpeed;

// a thread
protected Thread readThread;

//old left and right encoder values
private int oldLeft;
private int oldRight;

// Constructor Method
// initialize comm port and other variables
public RobotController(){}// does nothing

public RobotController(String comm) {
  try{
    commId = CommPortIdentifier.getPortIdentifier(comm);
  } catch (NoSuchPortException e){e.printStackTrace();}

  try{
    sPort = (SerialPort) commId.open("RobotController",1000);
  } catch (PortInUseException e){e.printStackTrace();}

  // declare default files
  rawDataFile = new File("rawEncData.txt");
  pathDataFile = new File("pathEncData.txt");

  try{
    inStream = sPort.getInputStream();
    outStream = sPort.getOutputStream();
    rawWriter = new FileWriter(rawDataFile);
    pathWriter = new FileWriter(pathDataFile);
  } catch (IOException e){e.printStackTrace();}
```

```
  try{
    sPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
                              SerialPort.STOPBITS_1,
                              SerialPort.PARITY_NONE);
  } catch (UnsupportedCommOperationException e)
   {e.printStackTrace();}

  try{
    sPort.addEventListener(this);
  } catch(Exception e){e.printStackTrace();}

  sPort.notifyOnDataAvailable(true);

  //
  readThread = new Thread(this);
  readThread.start();

  // set parameters
  this.commString = comm;

  this.xLoc = 0;
  this.yLoc = 0;
  this.theta = 0;
  this.wheelBase = 10*25.4;//mm
  this.wheelRadius = 3*25.4;//mm
  this.leftWheelSpeed = 0;
  this.rightWheelSpeed = 0;
  this.hasNewSpeeds = false;
}

// declare the run() method
public void run(){
  try {
    Thread.sleep(20000);
  } catch (InterruptedException e) {}
}

// declare the serialEvent(...) method
// should read data, translate it to
// encoder tics, and then send wheelspeed
// data back to the microcontroller
public void serialEvent(SerialPortEvent event) {
  switch(event.getEventType()) {
    case SerialPortEvent.BI:
    case SerialPortEvent.OE:
    case SerialPortEvent.FE:
    case SerialPortEvent.PE:
    case SerialPortEvent.CD:
    case SerialPortEvent.CTS:
    case SerialPortEvent.DSR:
    case SerialPortEvent.RI:
```

```
    case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
      break;
    case SerialPortEvent.DATA_AVAILABLE:
      // data recieved
      byte[] readBuffer = new byte[10];
      try { // read data
        while (inStream.available() > 0) {
          int numBytes = inStream.read(readBuffer);
        }

        // send forwardKinematics the
        // wheelspeeds from readbuffer
        forwardKinematics(toEncoder(readBuffer));

      } catch (IOException e) {e.printStackTrace();}

      // send wheel speed data
      if(this.hasNewSpeeds){
        this.hasNewSpeeds = false;
        try{
          outStream.write(this.rightWheelSpeed);
          outStream.write(this.leftWheelSpeed);
        } catch (IOException e){e.printStackTrace();}
      }
      break;
  }
}

// updates the position and orientation
// of the robot after each interrupt (~10 ms)
// Inputs: wheel positions- change in wheel
// positions from last interrupt
// Outputs: updates xLoc,yLoc,theta
protected void forwardKinematics(int[] deltaLeftRight){

  if(Math.abs(deltaLeftRight[0]) < 200 &&
     Math.abs(deltaLeftRight[1]) < 200){

    double disL = 2*Math.PI*deltaLeftRight[0]
                  *wheelRadius/2000;
    double disR = 2*Math.PI*deltaLeftRight[1]
                  *wheelRadius/2000;

    xLoc += (disR+disL)*Math.sin(theta)/2;
    yLoc += (disR+disL)*Math.cos(theta)/2;
    theta += (disR-disL)/wheelBase;

    try{
      pathWriter.write("\t" + xLoc + "\t" + yLoc
       + "\t" + theta + "\n");
      pathWriter.flush();
```

```
    } catch (IOException e) {e.printStackTrace();}
  }
}

// tell the microcontroller to begin (G!)
protected void goExclamation(){
  try{
    System.out.println("GO!");
    outStream.write(71);
    outStream.write(33);
    outStream.write(13);
  } catch (IOException e){e.printStackTrace();}
  startDate = new Date();
}

// raw should be a byte array of length 4
// this will convert the data from bytes to encoder ticks
private int[] toEncoder(byte[] raw){
  int left,right;

  // convert to decimal
  if(raw[0] >= 0)
    left = 256*raw[0];
  else
    left = (raw[0]+256)*256;
  if(raw[1] >= 0)
    left += raw[1];
  else
    left += raw[1]+256;

  if (left > 65536/2)
    left -= 65536;

  // convert to decimal
  if(raw[2] >= 0)
    right = 256*raw[2];
  else
    right = (raw[2]+256)*256;
  if(raw[3] >= 0)
    right += raw[3];
  else
    right += raw[3]+256;

  if (right > 65536/2)
    right -= 65536;

  try{
    rawWriter.write("\t" + left + "\t" + right + "\n");
    rawWriter.flush();
  } catch (IOException e) {e.printStackTrace();}
```

```
    int[] out = new int[2];
    out[0] = left-oldLeft;
    out[1] = right-oldRight;

    oldLeft = left;
    oldRight = right;
    return out;
  }

  public void setParameters(int whRad, int whBas){
    this.wheelBase = whBas;
    this.wheelRadius = whRad;
  }

  public void setWheelSpeeds(int left,int right){
    this.leftWheelSpeed = left;
    this.rightWheelSpeed = right;
    this.hasNewSpeeds = true;
  }

  public static void main(String args[]){
    RobotController rc1 = new RobotController("COM1");
    rc1.goExclamation();
    rc1.setWheelSpeeds(-10,-10);
  }
}

// author: David Sorensen
// Texas A&M Robotics Lab
// DualMouseController.java

import javax.comm.*; import java.io.*; import java.util.ArrayList;
import java.util.Date; import java.lang.Math;

public class DualMouseController extends RobotController {
  private static int TICS_PER_MM = 16;

  // save the location and orientation of the mouse
  private int mouse1X;
  private int mouse1Y;
  private double mouse1Theta;

  private int mouse2X;
  private int mouse2Y;
  private double mouse2Theta;

  private Matrix aInverse;
  private Matrix b;
  private Matrix X;

  // xLoc,yLoc,theta are the estimated position
```

```
// for state
private int ohsRecieved;
private boolean m2Received;
private boolean m1Received;

public DualMouseController(String comm) {
  // code left out... much similar to the RobotController class
  ...
  this.isRunning = false;
  this.ohsRecieved = 0;
  this.m2Received = false;
}

// declare the serialEvent(...) method
// should read data, translate it to encoder tics
// and then send wheelspeed
// data back to the microcontroller
public void serialEvent(SerialPortEvent event) {
  switch(event.getEventType()) {
    case SerialPortEvent.BI:
    case SerialPortEvent.OE:
    case SerialPortEvent.FE:
    case SerialPortEvent.PE:
    case SerialPortEvent.CD:
    case SerialPortEvent.CTS:
    case SerialPortEvent.DSR:
    case SerialPortEvent.RI:
    case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
      break;
    case SerialPortEvent.DATA_AVAILABLE:
      try { // read data
        int totalBytes = inStream.available();
        byte[] readBuffer = new byte[totalBytes];
        while (inStream.available() > 0) {
          int numBytes = inStream.read(readBuffer);
        }
        if(this.isRunning)
          forwardKinematics(toEncoder(readBuffer));
        else {
          for(int i=0;i<totalBytes;i++){
            System.out.println(readBuffer[i]);
            if(readBuffer[i] == 79){ // recieved 'O'
              System.out.println("Received one O");
              ohsRecieved++;
            }
            if(ohsRecieved == 2) // received 2 'O's
              this.isRunning = true;
          }
        }
      } catch (IOException e) {e.printStackTrace();}
```

```
      break;
  }
}

// raw should be a byte array of length 2
// this will convert the data from bytes to mouse ticks
private byte[] toEncoder(byte[] raw){
  int[] out = new int[raw.length];
  for (int i=0;i<raw.length-1;i+=2){ // write bytes to file
    int x,y;
    if(m2Received && m1Received){
      try{
        rawWriter.write("\n");
        rawWriter.flush();
      } catch (IOException e){e.printStackTrace();}
      m2Received = false;
      m1Received = false;
    }
    try{
      if(!m1Received){ // just received m1 data
        m1Received = true;
      }
      else if(!m2Received){ // just received m2 data
        m2Received = true;
      }
      rawWriter.write(raw[i] + "\t" + raw[i+1] + "\t");
      rawWriter.flush();
      } catch (IOException e) {e.printStackTrace();}
  }
  return raw;
}

// updates the position and orientation of
// the robot after each new mouse data
// Inputs: mouse delta x and y- change in x
// and y positions according to the mouse
// Outputs: updates xLoc,yLoc,theta
protected void forwardKinematics(byte[] deltaXY){
  if(deltaXY[0] < 70){
    // check that the robot is not sending back characters
      for(int i=0;i<deltaXY.length;i+=4){
      // convert to x,y at center used for a single mouse
      double vX1,vY1,vX2,vY2,vRx,vRy,omegaR,omega1,omega2;

      double norm1 = Math.sqrt(deltaXY[0]*deltaXY[0]
                     +deltaXY[1]*deltaXY[1]);
      double norm2 = Math.sqrt(deltaXY[2]*deltaXY[2]
                     +deltaXY[3]*deltaXY[3]);

      vX1 = (deltaXY[i]*Math.cos(mouse1Theta)-deltaXY[i+1]
                     *Math.sin(mouse1Theta))/TICS_PER_MM;
```

```
vY1 = (deltaXY[i]*Math.sin(mouse1Theta)+deltaXY[i+1]
                *Math.cos(mouse1Theta))/TICS_PER_MM;

vX2 = (deltaXY[i+2]*Math.cos(mouse2Theta)-deltaXY[i+3]
                *Math.sin(mouse2Theta))/TICS_PER_MM;
vY2 = (deltaXY[i+2]*Math.sin(mouse2Theta)+deltaXY[i+3]
                *Math.cos(mouse2Theta))/TICS_PER_MM;

omega1 = -(vX1/this.mouse1Y);
omega2 = -(vX2/this.mouse2Y);

b.A[0][0] = vX1;
b.A[1][0] = vY1;
b.A[2][0] = vX2;
b.A[3][0] = vY2;

X = null;

if(norm1 > 3/16 && norm2 > 3/16){
  // use pseudo inverse
  X = aInverse.multiply(b);
  vRx = X.A[0][0];
  vRy = X.A[1][0];
  omegaR = X.A[2][0];
  //System.out.println("PINV");
}
else if(norm2 <= 3/16){
  // convert to the robot's coordinate system
  omegaR = -(vX1/this.mouse1Y);
  vRy = vY1+(vX1*this.mouse1X/this.mouse1Y);
  vRx = 0;
  //System.out.println("M1");
}
// find omega and velocity of the robot's
// center (forward kinematics)
else{
  omegaR = -(vX2/this.mouse2Y);
  vRy = vY2+(vX2*this.mouse2X/this.mouse2Y);
  vRx = 0;
}

// if the omegas from individual mice are close,
// disallow sideways motion
if(Math.abs(omega1-omega2) < .003){
  vRx = 0;
}

//System.out.println(vRx + " " + vRy + " " + omegaR);
// update the location of the robot in the
// absolute coordinate system
this.xLoc += vRx*Math.cos(theta) - vRy*Math.sin(theta);
```

```
        this.yLoc += vRy*Math.cos(theta) + vRx*Math.sin(theta);
        this.theta += omegaR;

        try{
          pathWriter.write(xLoc + "\t" +
            yLoc + "\t" + theta + "\n");
          pathWriter.flush();
        } catch (IOException e) {e.printStackTrace();}
      }
    }
  }

  public void setParameters1(int mouseX, int mouseY, double mouseTheta){
    this.mouse1X = mouseX;
    this.mouse1Y = mouseY;
    this.mouse1Theta = mouseTheta;
  }

  public void setParameters2(int mouseX, int mouseY, double mouseTheta){
    this.mouse2X = mouseX;
    this.mouse2Y = mouseY;
    this.mouse2Theta = mouseTheta;
  }
  public void setAInverse(){
    Matrix A = new Matrix(4,3);
    A.A[0][0] = 1; A.A[0][1] = 0; A.A[0][2] = (double)-this.mouse1Y;
    A.A[1][0] = 0; A.A[1][1] = 1; A.A[1][2] = (double)this.mouse1X;
    A.A[2][0] = 1; A.A[2][1] = 0; A.A[2][2] = (double)-this.mouse2Y;
    A.A[3][0] = 0; A.A[3][1] = 1; A.A[3][2] = (double)this.mouse2X;

    aInverse = A.pseudoInverse();
  }

  public double getX(){
    return this.xLoc;
  }
  public double getY(){
    return this.yLoc;
  }
  public double getTheta(){
    return this.theta;
  }

  public void setWheelSpeeds(int left, int right){}// does nothing

  public static void main(String[] args) {
    DualMouseController mouseController1 = new DualMouseController("COM2");
    mouseController1.goExclamation();
  }
}
```

VITA

David Kristin Sorensen
PO Box 143162
Austin, TX 98714

EDUCATION

Texas A&M University **College Station, TX**
Master of Science – Mechanical Engineering
Graduated: May 2003
Degree GPA: 3.50

Bachelor of Science – Mechanical Engineering
Graduated: December 2000
Overall GPA: 3.06

WORK EXPERIENCE

Spring 2003 – Present
Texas A&M University – Mechanical Eng. **College Station, TX**
Teaching Assistant
Aid students in learning essential ideas of mechatronics, microcontrollers and C programming.
Conduct lab sessions consisting of hardware and software implementations of concepts taught in class.

Summer 2002 – Present
Texas A&M University – Robotics Lab **College Station, TX**
Research Assistant
Formulated and implemented a kinematic robot model in MATLAB.
Constructed a mobile robot and integrated various sensors.
Implemented embedded and high-level programs which together control the operation of the robot and its sensors.

Summer 1999
BAE Systems **Austin, TX**
Engineering Intern II
Created rocket trajectory visualization using FORTRAN.
Constructed comparative analyses of fin-door designs for a guided rocket.
Assisted in editing and clarifying a proposal that was accepted by the DoD.

ACTIVITIES

Team Leader of the Driver Interface group on the 2000 Texas A&M Formula SAE team (Fall 1999 – Summer 2000)

Member of Texas A&M Men's Soccer Team (Fall 2002 – Spring 2003)