A REAL-TIME POSITIONING SYSTEM FOR MOBILE ROBOTS BASED ON LASER TRIANGULATION

By

Michelle Marie Simi

A thesis submitted to the faculty of
Northwestern University
in partial fulfillment of the requirements for the degree of

Masters of Science

in

Mechanical Engineering

Department of Mechanical Engineering

Northwestern University

December 2000

I would like to dedicate this work to my biggest fan, my Mom.
She always supports me no matter what path in life I choose
and I strive everyday to continue to make her proud.

ACKNOWLEDGMENTS

I would like to begin by thanking my Mom and Joe for their endless encouragement and faith in me. I would also like to thank the rest of my family and friends for their support during the past year especially the Rodgers household, which has become a second home to me.

I would like to thank the members of LIMS for making the lab an atmosphere where ideas thrive. While there are many people I wish to recognize such as Tanya Tickel, Carl Moore, Songho Kim, and Cassandra Eggett, I would like to especially thank Traveler Hauptman. Traveler has been a continual source of support for me since I got here at Northwestern for which I am truly grateful.

Last, but certainly not least, I would like to express my sincerest gratitude to Professor Peshkin for giving me the opportunity to work on this project. I cannot thank him enough for giving me a project that would challenge me and force me to learn new things. Under his tutelage, I have become more of an individual thinker and I am truly fortunate to have had the opportunity to have him as my mentor and advisor.

ABSTRACT

This thesis describes a method for determining the position and orientation of a mobile robot based on laser triangulation. It discusses the governing equations and the numerical method necessary to solve them. Detector devices for sensing the laser beam as it sweeps by are discussed along with the method for manufacturing them. The position and orientation errors of the system are determined and the results are discussed.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 What are Cobots?

A “cobot” is a robotic device designed for collaboration with human operators in a shared workspace. As passive haptic devices, cobots are designed to assist users by providing software-defined guiding surfaces while the operator supplies all the energy to the system. Since a cobot does not generate energy, it is unlikely that a human working with a cobot system will harm themself or damage equipment. It is this passive nature that makes cobots appropriate in applications such as automotive assembly and medical surgery that require the operator to work side-by-side with a robotic device.

In the Laboratory for Intelligent Mechanical Systems (LIMS) two types of cobots have been designed namely wheel based cobots and spherical joint based cobots. Spherical joint based cobots will not be addressed in this paper. Several wheeled cobots have already been designed including a one-wheeled cobot called “Unicycle” and a three-wheeled cobot called “Scooter” (See Figure 1-1 and Figure 1-2 on the following page). The unicycle cobot was designed to illustrate some basic cobot concepts. The unicycle cobot has one wheel, a two-dimensional configuration space, and requires a rail system to keep it upright. Because of these disadvantages, the three-wheeled cobot, Scooter, was developed. Scooter is designed to navigate around a workspace and while two wheels would have been sufficient for this task; an additional wheel was added to eliminate singularities and for balance.

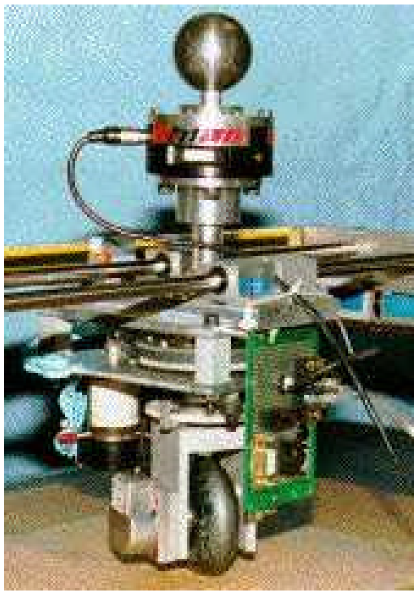
Figure 1-1: Unicycle Cobot

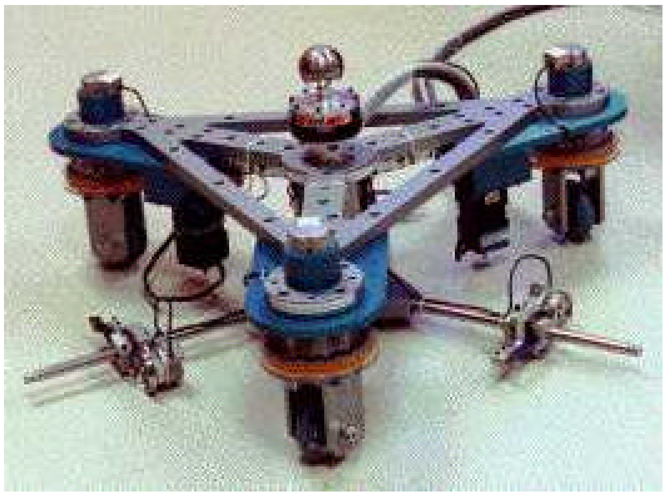
Figure 1-2: Scooter Cobot

Scooter has been the base concept for cobots that are now being used in industry. Scooter's technology is currently being used at General Motors to assist operators with tasks such as removing car doors from empty auto bodies after painting to allow assembly of the rest of the vehicle. It is this second cobot that is of interest for this project and will be discussed further in the next section.

1.2 Current Problem

Scooter currently uses three "glide wheels" to determine its position/orientation and velocity. The glide wheels consist of plastic wheels located at fixed angles with respect to Scooter that have encoders mounted to them that measure their rotation. The rotation of each glide wheel measures the component of Scooter's velocity in its rolling direction, while ignoring the component of velocity in the perpendicular direction (sliding). Scooter's velocity is then determined using the velocities of the three glide wheels. The resulting velocity is integrated to determine an approximation of Scooter's position (dead-reckoning).

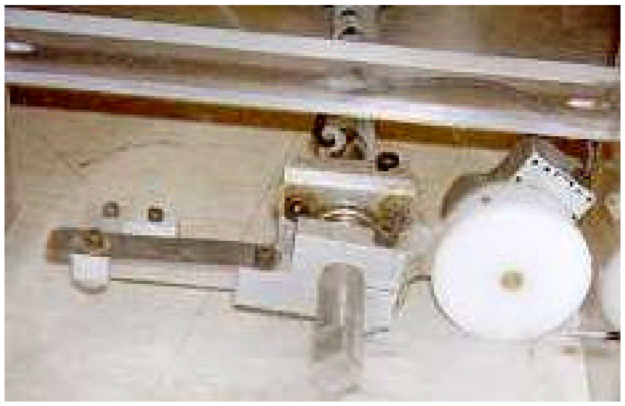

Figure 1-3: One Glide Wheel on Scooter

As with any system that uses dead-reckoning, error will always accumulate over time; such is the case for Scooter. The error buildup is detrimental to accurate representation of a virtual surface because the virtual surface is dependent on Scooter's perceived world coordinates. It is for this reason that the door unloader at General Motors must be zeroed every cycle at the drop off station to eliminate the accumulated error. If the position error were eliminated, programmable virtual surfaces could enable more precise material handling without repeated zeroing.

1.3 Project Goal

The goal of this project is to develop a more accurate way of determining position and orientation of a mobile robot. The method explored in this paper is based on triangulation. A laser mounted on a mobile robot, e.g. Scooter. Using a synchronous motor, the laser beam is swept 360° horizontally around the workspace. Three sensors are placed at known locations in the workspace to detect the laser beam as it sweeps by.

The angle of the beam at the moment that the flashes are detected is used to triangulate the position and orientation of the laser source (see Figure 1-4 below).

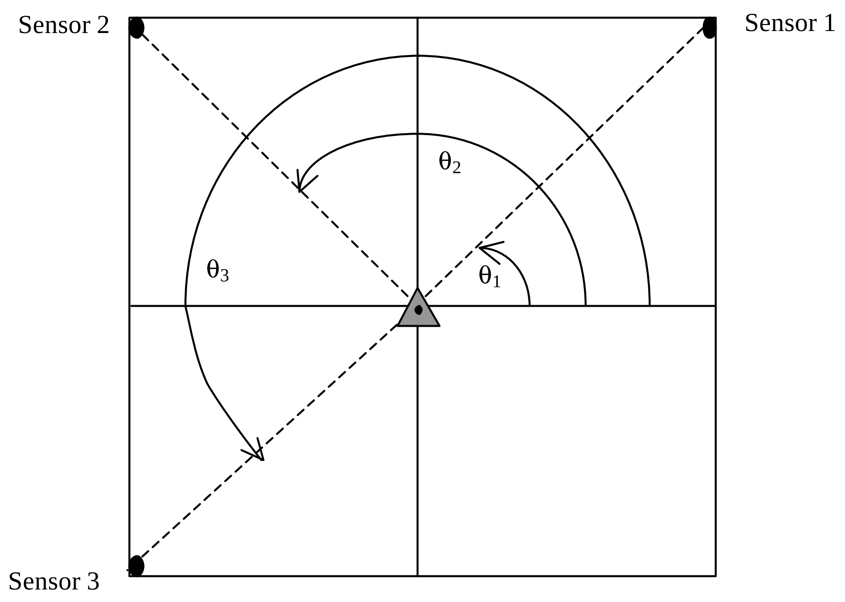

Figure 1-4: System Overview

There are several aspects of this design that should be understood and will be discussed in the subsequent chapters. The next chapter will discuss the development of triangulation equations and the numerical method used to solve them. The design of flash detectors will also be discussed. Chapter 3 will give an overview of the hardware set-up, sensor electronics, and the encoder interface. Experimental results will be discussed in Chapter 4, and the final chapter will summarize the project.

CHAPTER 2: THEORETICAL WORK

In order to begin this project, the first thing to consider was how triangulation would work and what would be needed in order to determine position and orientation. Therefore, the next few sections will discuss the governing equations for triangulation and the numerical method used to solve the equations. The design of flash detectors that enable the necessary inputs to be received will also be discussed.

2.1 Triangulation Equations

To define the governing equations for this project, a few variables must be defined. First, the workspace dimensions must be known. Second, the coordinates of the flash detectors within the workspace must be determined. The case of pure rotation by the mobile robot will be considered first.

Consider the workspace shown in Figure 2-1. If the mobile robot's position is (0, 0) and the location of one flash detector is known, (x_{R1}, y_{R1}), the orientation, θ_s, can be determined if given the angular input, θ_1, that corresponds to the angle at which the laser beam sweeps past the flash detector.

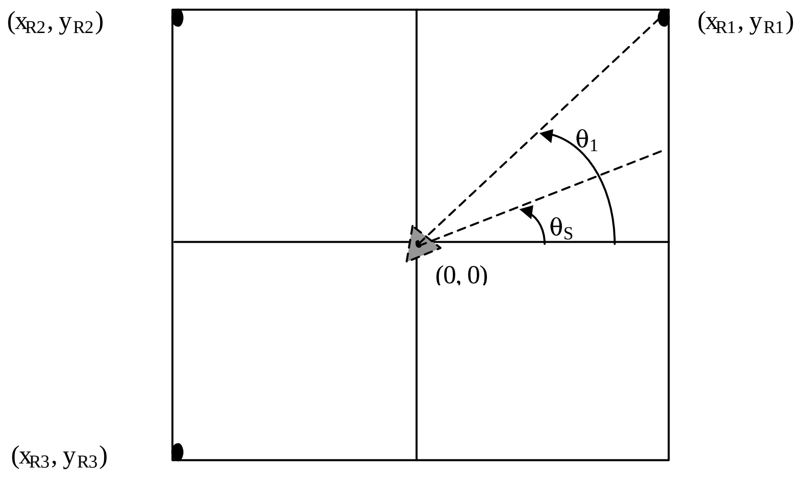

Figure 2-1: Case of Pure Rotation

Therefore, one flash detector resulting in one equation is all that is necessary to determine orientation based on the following equation:

$$\tan(q_s + q_1) = \frac{y_{R1}}{x_{R1}} \qquad \text{where } x_{R1} \text{ and } y_{R1} \text{ are knowns and } q_1 \text{ is an input}$$

Now considering the case of pure rotation as shown in Figure 2-2. The position location contains two unknowns, (x_s, y_s), therefore two flash detectors are needed in order to obtain the necessary equations.

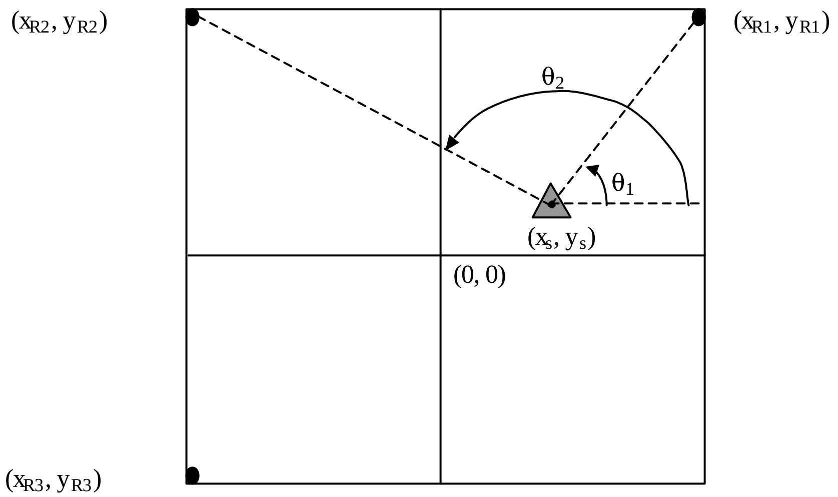

Figure 2-2: Case of Pure Translation

The equations would be of the form:

$$\tan q_i = \frac{y_{Ri} - y_s}{x_{Ri} - x_s} \qquad \text{for } i = 1,2 \text{ where } x_{Ri} \text{ and } y_{Ri} \text{ are knowns and } q_i \text{ is an input}$$

Combining the two cases gives the following equation (See Figure 2-3 on the following page for workspace diagram):

$$\tan(q_s + q_i) = \frac{y_{Ri} - y_s}{x_{Ri} - x_s} \qquad \text{for } i = 1,2,3 \text{ where } x_{Ri} \text{ and } y_{Ri} \text{ are knowns and } q_i \text{ is an input}$$

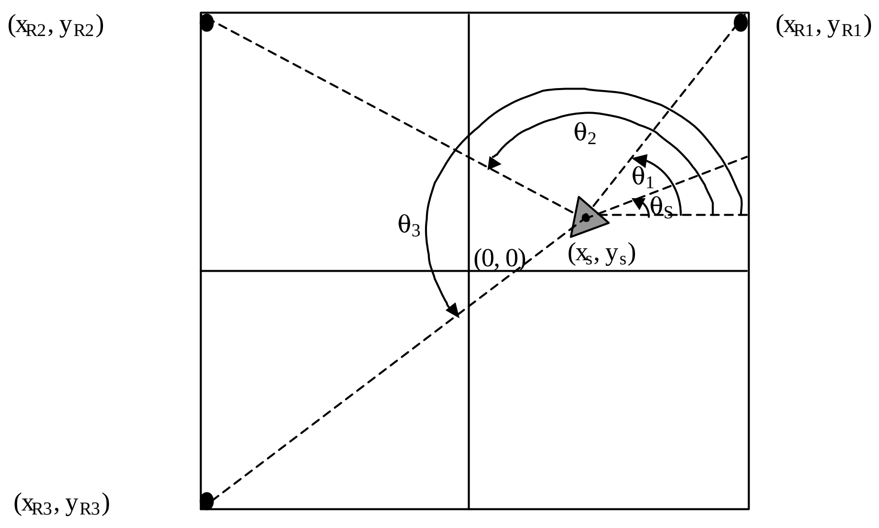

Figure 2-3: Case of Rotation and Translation

Therefore, three equations need to be solved simultaneously to determine the three unknowns: a position (x_s, y_s) and an orientation (θ_s). Since tangent isn't defined at 90° and 270°, the atan2 function was used when programming in C. The system of equations can be represented as:

$$F_i(x_s, y_s, q_s) = \text{atan2}\left(y_{Ri} - y_s, x_{Ri} - x_s\right) - q_s - q_i \qquad \text{for } i = 1,2,3$$

2.2 Numerical Analysis

Since the system of equations discussed in the previous section is nonlinear and not analytically invertible, the Newton-Raphson method was used to solve the system of equations. (For an analytical solution, see Appendix A.) The Newton-Raphson method first calculates $\mathbf{F_i}$ and the Jacobian, **J**, by taking initial guesses for x_s, y_s, and θ_s. For this set of equations, the Jacobian matrix would be as follows:

$$J \equiv \begin{bmatrix} \frac{(y_{R1} - y_S)x_S}{(x_{R1} - x_S)^2} & \frac{1}{x_S - x_{R1}} & -\sec^2(q_S + q_1) \\ \frac{(y_{R2} - y_S)x_S}{(x_{R2} - x_S)^2} & \frac{1}{x_S - x_{R2}} & -\sec^2(q_S + q_2) \\ \frac{(y_{R3} - y_S)x_S}{(x_{R3} - x_S)^2} & \frac{1}{x_S - x_{R3}} & -\sec^2(q_S + q_3) \end{bmatrix}$$

Once F_i and the Jacobian are calculated, the program checks to see if the sum of the F_i solutions (absolute) is within some tolerance value. If so, then the initial guess was accurate and the program is complete. If not, LU decomposition is performed to obtain new values of F_i. Again root convergence is checked and if the sum of the absolute values of these solutions is within tolerance, the program is complete. Otherwise, the entire process is repeated either until a solution is found or the program exceeds a certain number of loops at which time an error message is given. (See Appendix B for program.)

One problem with the Jacobian defined above, however, it that when either of the mobile robots coordinates, x_s or y_s, are at the same position as a flash detector, x_{Ri} or y_{Ri}, the program will not converge to an accurate solution due to zeroes. It was for this reason that the exact solution of the Jacobian as stated above was not used and instead the Jacobian was solved for using a forward difference formula.

2.3 Flash Detector Design

When considering how to get the necessary inputs, θ_i, a couple of things had to be considered. To begin with, it is not practical to just simply place sensors around a room since it is unlikely that the laser beam will be aligned perfectly with them. Therefore, focusing the laser beam seemed like an appropriate method of getting the laser light to a sensor. While there are lenses that will focus light, they all assume that the light is entering "head-on" meaning that the orientation of the incoming light and the focusing device are the same (parallel). This would mean that the mobile robot would have to be directly in line with the sight line of the focusing device. With a mobile robot moving throughout a workspace, the light will not always be "head-on", but perhaps with some incident angle. Hence, a design was sought that would focus the light onto the sensor at an angle.

Another concern was "tilt". The floor on which Scooter rolls may not be ideal; therefore, the detector design needed to account for the light being tilted upwards or downwards slightly due to deviations present in the floor.

To account for tilting deviations, it was desirable for the detectors to have considerable height, but minimal depth to take up the least amount of space. It was for these reasons that a parabolic shape (See Figure 2-4 on the following page). The equation of a parabola is as follows:

$$y = ax^2 \qquad \text{where } a = \frac{1}{4c} \text{ and the focus occurs at } (0, c).$$

A focal point, c, of 2 inches was chosen and the overall dimensions of the parabola became 4 inches tall, y, and 2 inches deep, x.

For the light to get to the sensor no matter what the entrance angle was, the sides of the parabolic shape and the back side were mirrorized (See Appendix C for mirrorizing instructions). This enables the laser beam to essentially ricochet or bounce between the walls and the back side of the parabolic device until they reach the sensor.

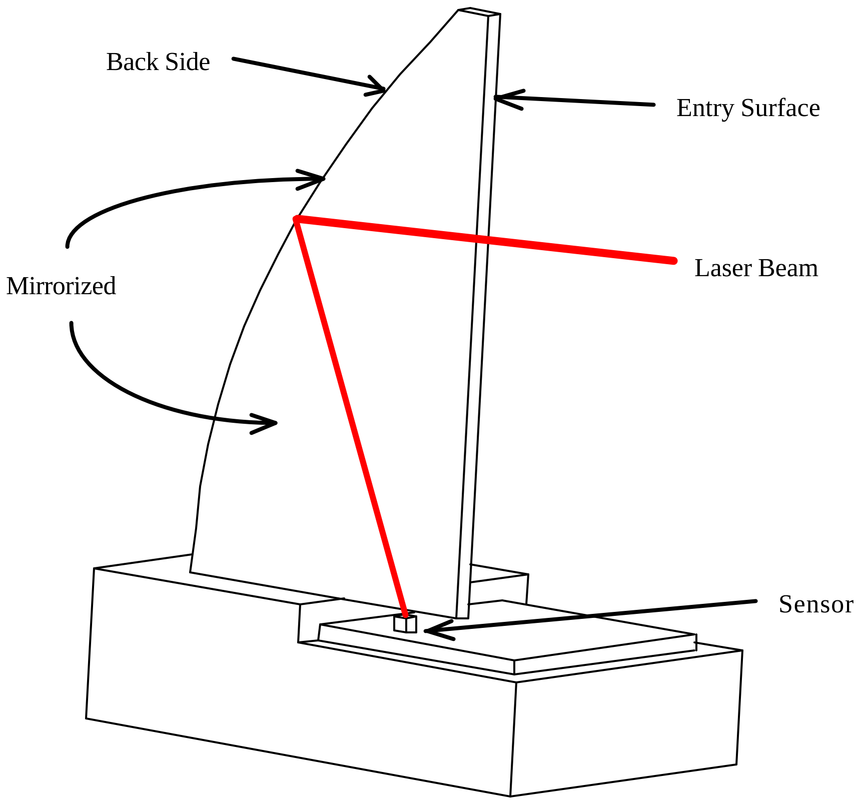

Figure 2-4: Flash Detector Design

Since accuracy is crucial, the width of the entry surface was of great importance. The entry width needed to be large enough to ensure proper data collection but small enough maintain a high level of accuracy. The time that the laser beam shines on the sensor, t_s, at any given distance, r, away from the mobile robot is:

$$t_s = \frac{c_w}{cir * m_R} \qquad \text{where } c_w = \text{entry widt h, } cir = 2pr\text{, and } m_R = \text{motor speed in rev/sec.}$$

Knowing that the sensor (PN127 discussed further in Chapter 3) has a 90% rise time of 2.5 µs, this will become the $t_{s\ min}$. Therefore, the minimum entry width at a distance of 20 feet with a motor speed of 30 rev/sec (typical for AC synchronous motors) would be 0.1131 inches. It is for this reason that the device has an entry surface only

0.125 inches wide. This will provide a sufficient area for the laser beam to enter without decreasing accuracy. Also, since the laser beam light will be bouncing back and forth within the device, the size of the sensor and the laser beam should be such that the sensor detects the laser light no matter its *exit* angle. The active sensing area on the PN127 is half a sphere with a diameter of 0.071 inches (1.8mm). This means that the laser beam diameter needs to be at least 0.054 inches wide to ensure that the sensor will detect the laser beam. The laser beam used has a diameter of 0.087 inches, which is more than sufficient to ensure detection.

The last important aspect of the flash detector design was what material for it to be made out of. Glass would have been the optimal choice, however, it has the major draw back of requiring special equipment in order to cut it; especially in the shape of a parabola. It was for this reason that a more manufacturable material, plexiglass, was used instead. The parabolic collecting devices were machined in a CNC mill and then buffed to remove cloudiness that developed during machining.

A major drawback to choosing plexiglass over glass is that glass mirrorizes better. In performing numerous experiments of mirrorization, it became obvious that the silvering solution preferred to stick to glass than to the plexiglass. While the mirrorizing process did work well on both materials, the reflectivities of the glass specimens were better than the plexiglass specimens.

CHAPTER 3: HARDWARE

3.1 Set-Up

In this section, the apparatus that allows the laser beam to spin will be discussed. The laser cannot actually spin itself for this would cause its wires to knot. Therefore, the configuration shown in Figure 3-1 below was used.

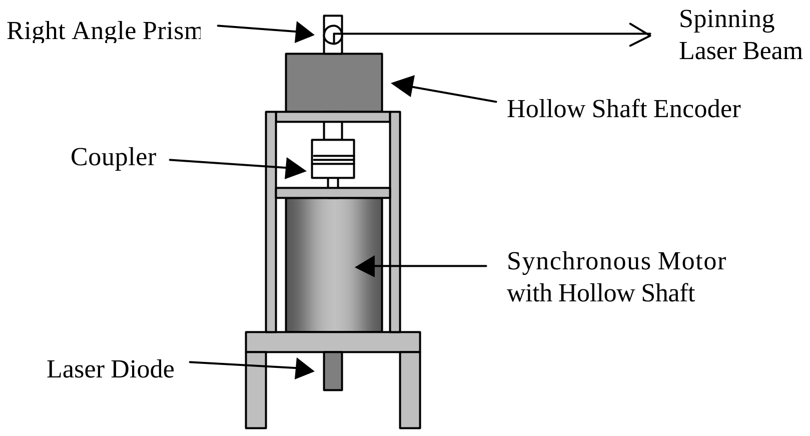

Figure 3-1: Hardware Set-up

A visible red laser beam of wavelength 670nm from Edmund Scientific (P/N H54022) shines vertically up the shaft of an AC synchronous motor from Bodine rated at 1800rpm (P/N 5246). A synchronous motor was chosen because it is crucial that the laser light be spun at the most constant rate possible. This particular motor was chosen, however, because it has a 5/16" motor shaft, which was large enough to drill a hole for the laser beam. Another important aspect of this motor was that the end that does not have the shaft sticking out, does not have a sealed bearing. This was important because a hole was drilled in the motor casing to allow the laser light to shine up the motor shaft.

There is a hollow shaft encoder from Hohner (P/N 8895-0100-12500) rated at 12,500 pulses per revolution used to measure how fast the motor is actually spinning. This particular encoder was chosen for a number of reasons. First of all, it came with a hollow shaft opening of 5/8". This was important because the opening needed to be large enough to allow an aluminum machined coupler to fit inside it and still have enough material for a hole big enough to allow the laser light to pass through.

To ensure that at least 1 encoder count is received when the laser sweeps across the sensor, the following calculation was done to determine the necessary resolution of the encoder, e_r:

$$e_r = \frac{cir}{c_w} \qquad \text{where } c_w = \text{collecting device width and } cir = 2pr.$$

Knowing that $c_w = 0.125$ inches and making the same assumption that r = 20 feet, the resolution of the encoder has to be at least 12064 ppr in order to get at least one count when the laser beam sweeps the sensor. By purchasing an encoder with a resolution of 12500ppr, it eliminated the need to interpolate between encoder counts to determine when the laser beam shined on the sensor.

A flexible coupler from Sterling Instrument (P/N S50HAA-087H1010) is used to account for any angular and parallel misalignments between the encoder and the motor. The coupler attaches directly to the motor shaft at one end and to the machined aluminum coupler at the other end. At the top of the aluminum coupler, there is a right angle prism (Edmund Scientific P/N H32331) inside and a hole to allow the laser beam to escape horizontally from the set-up. With everything assembled, the laser beam sweeps 360° around the workspace.

3.2 Sensor Electronics

The sensor used to detect the laser beam is a PN127 silicon NPN phototransistor. This particular sensor was chosen because it had the greatest spectral sensitivity to laser wavelengths in the visible red region, namely 670nm, and had the shortest rise/fall times (2.5μs / 3.5μs). Figure 3-2 below shows the sensors spectral sensitivity curve.

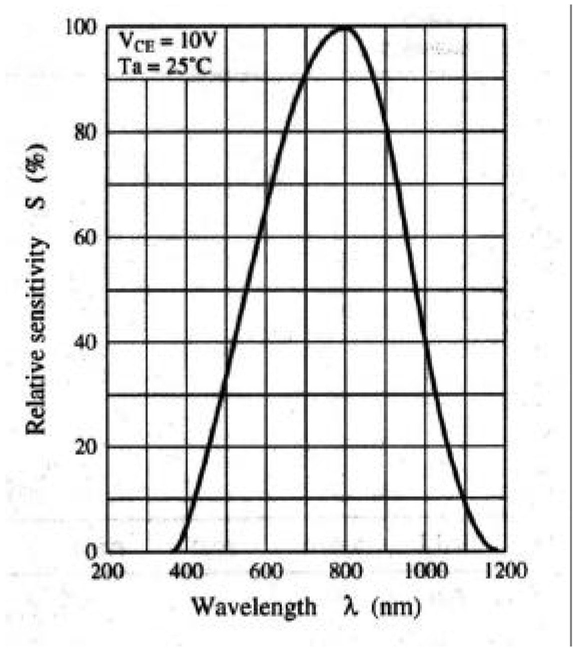

Figure 3-2: PN127 Spectral Sensitivity Characteristics

Once the phototransistor detects the laser beam, a LT1221 operational amplifier, in conjunction with a RC filter of time constant 1μs, is used to amplify the output. A 50KΩ potentiometer, currently setting the threshold to -0.5 volts, feeds into an input of a LM360 high speed differential comparator which eliminates noise and produces a nice square wave. Lastly, a DM74LS123 retriggerable one-shot was used to ensure that the output pulses were always long enough for the encoder interface to detect (10 μs). (See

Figure 3-3 for the schematic. Please note that X1 and X2 are the inverting and noninverting circuit outputs.)

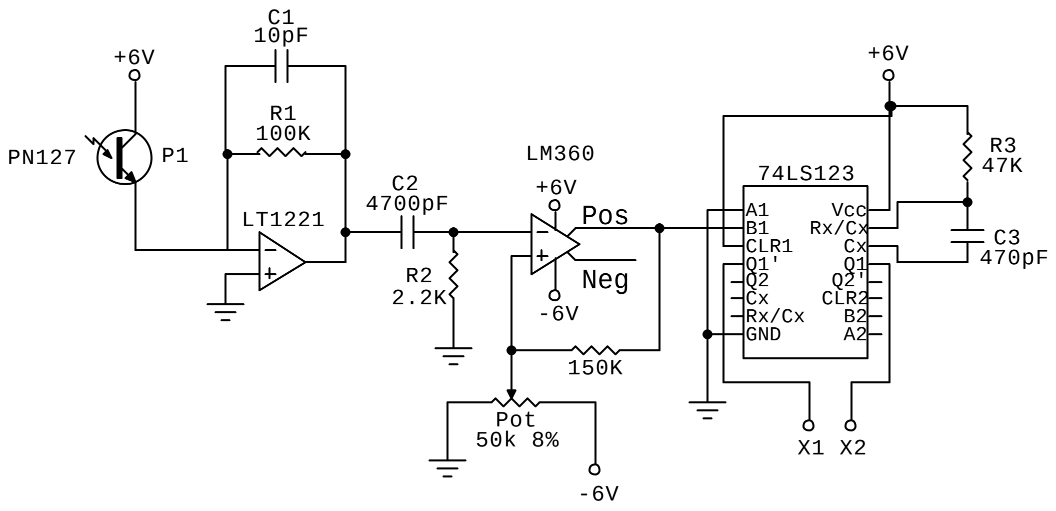

Figure 3-3: Sensor Circuit Schematic

By using this circuit and the flash detectors discussed in Chapter 2, the laser beam can be detected at incident angles up to 44° at 1 foot away and 27° at 7 feet away. This is because of a reduction in the laser beam intensity due to poor reflectivity on the plexiglass and an increase in the number of ricochets.

3.3 Encoder Interface

This is actually the heart of this project. All of the calculations depend on one sensory input, θ_i. In order to get angles at which the phototransistor detects the laser beam, an encoder decoder board was used. A 4 channel, differential, quadrature encoder interface board, PC7266-D, from US Digital was purchased. This particular board uses a LS7266 counter chip which is able to not only decode encoder counts, but load the counter value to an output latch whenever an index pulse occurs. It was this last feature that made this board especially useful. As demonstrated in Figure 3-4 below, the encoder has been wired to all four encoder board channels. All four channels were also configured to load the counter value to the output latch when the index location receives a pulse. One encoder board input channel was wired to the encoder completely. The other three input channels receive index pulses from the output of the flash detector circuits in addition to the encoder's A, B, A-, and B- pulses.

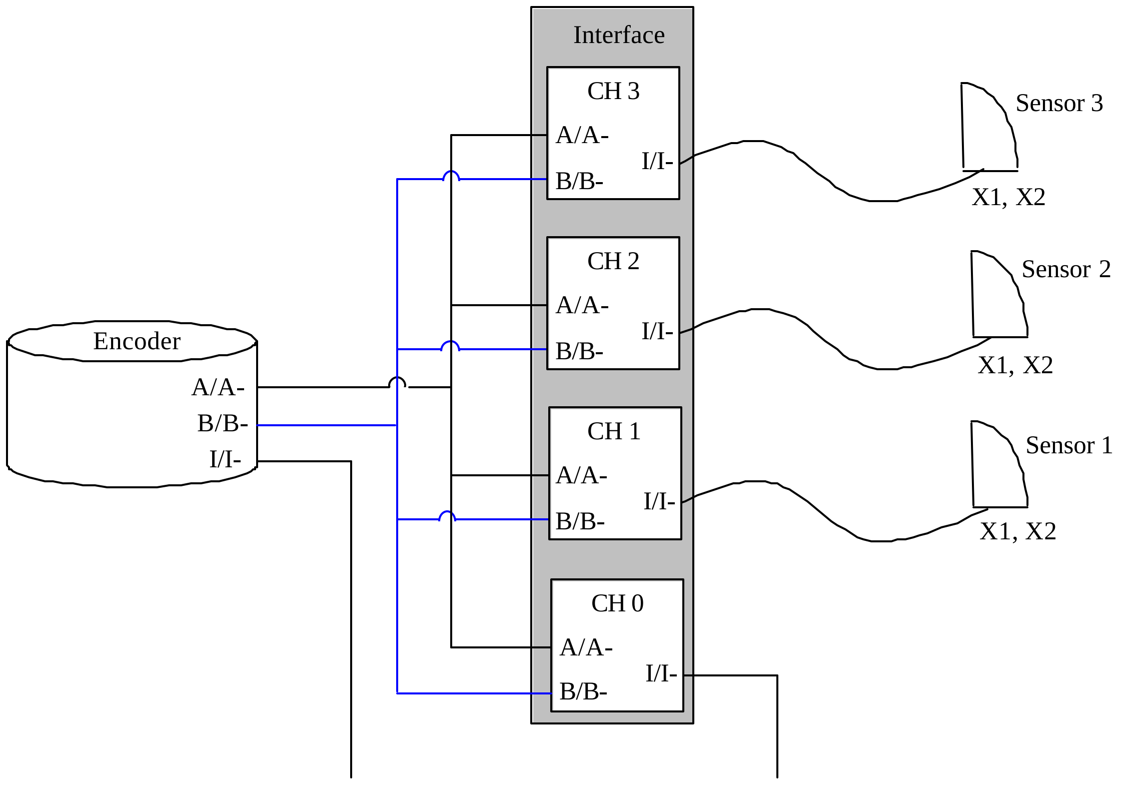

Figure 3-4: Block Diagram Illustrating Wiring Scheme

By loading the counter values to the output latch whenever an index pulse is observed, this allows the computer to come along at any time and get the encoder counts for all four channels without having to use an interrupt service routine. Once the counter values are latched, the sensor angles measured from zero can be calculated by subtracting the encoder's index count from the phototransistors count value and then simply converting encoder counts into angular displacement in radians.

When attempting to drive all four counters with only one encoder, it became obvious that the encoder's signal strength was not great enough to drive all four decoder channels. Therefore, a high current/voltage darlington driver chip (DS2003) was added to make the signal strong enough for accurate pulse recognition (See Appendix D for schematic).

CHAPTER 4: EXPERIMENTAL WORK

4.1 Experimental Protocol

The goal of this experiment was to determine the amount of error that this system is encountering. The error should be minimal and no longer cumulative like the previous approach.

4.2 Experimental Results

For the experimental set-up, the three flash detectors are placed so they form an equilateral triangle with all sides being 119.75 inches long. The triangle was established by taking a rod of known length and positioning the flash detectors equidistant apart. In addition, the flash detectors are angled so their entry surface is approximately aligned with the center of the triangle lending the maximum amount of area visible by the flash detectors. The motor assembly is placed at some location within the triangle.

In determining the error of the system for the x and y components, first the error in the input values of the coordinates of the flash detectors had to be determined. To do this, a geometrical method was used to determine the approximate starting location of the cobot in relation to the three flash detectors. (The procedure for doing this is outlined in Appendix E.) This method showed that there is a flash detector coordinate error less than 0.042 inches and the starting location for the motor assembly is averaged to (57.497, -57.818).

This average was then compared to the optical triangulation solution determined by the program. The starting position under this method was (57.465, -58.116). This gives a difference in values of 0.032 inches in the x-direction and 0.298 inches in the y-direction. This is the measured error of the optical triangulation system.

The next step was to create a map of the workspace. To do this, the floor tiles and the initial optical solution position (57.465, -58.119) were used as a guide for performing12 inch translations in the x and y. The maximum amount of difference between the tile values and the optical values was 0.188 inches in the x-direction and 0.259 inches in the y-direction. The error as it compared to position in the workspace is shown in Figure 4-1 below. (See Appendix F for table with data points).

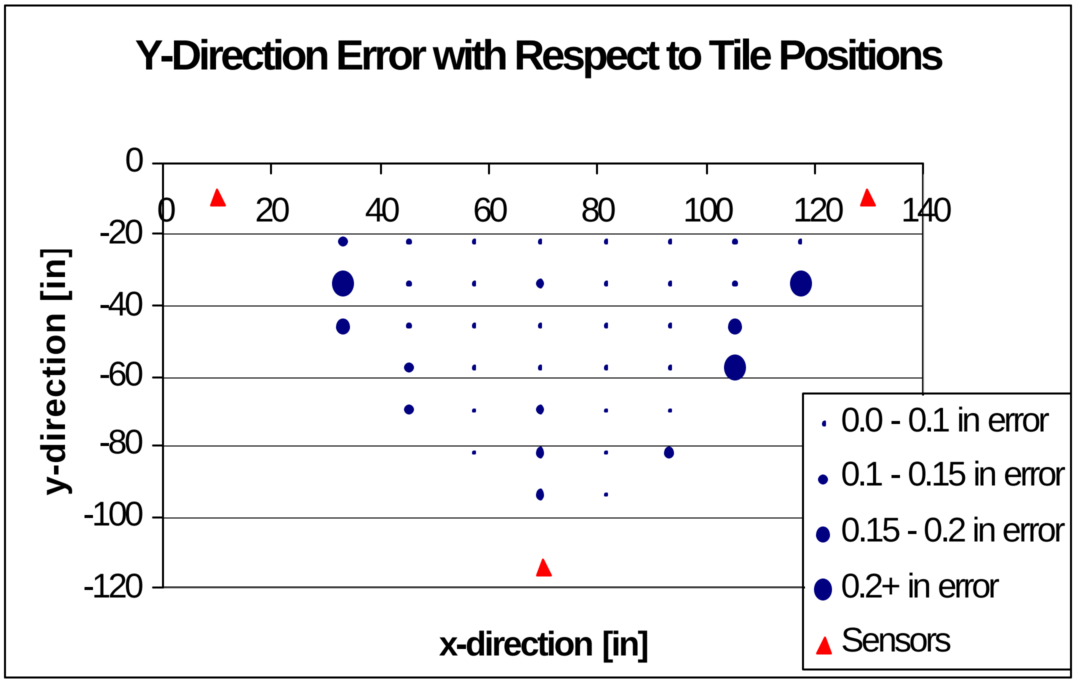

Figure 4-1: Plot of Y-Direction Error with Respect to Positions

In order to determine the error in the orientation, five positions in the workspace were chosen and at each position the motor assembly was rotated 45°, 90°, -45°, and -90°. The maximum error was found to be 0.8 degrees. The results are given in Figure 4-2 on the next page.

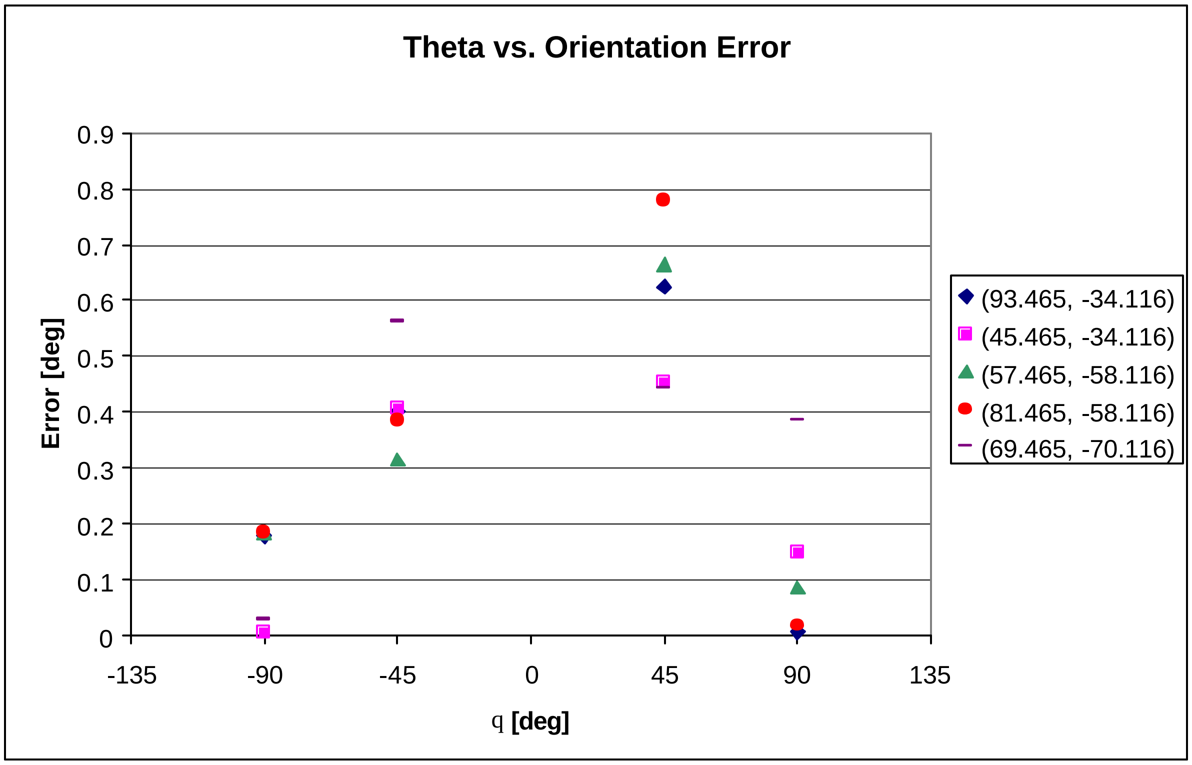

Figure 4-2: Orientation Error Results

4.3 Discussion of Experimental Results and Limitations

The experimental results find errors of up to 0.259 inches for position variables and 0.8 degrees for orientation. While these are slightly larger than would be desired, the error is still minimal. Seeing that the major jumps in error occur on the outer edges of the workspace suggests that the error is probably due to poor data collection by the flash detectors rather than computational error.

In this experiment there are many sources of error that could occur. One source of error is that the floor tiles are not perfectly 12 inches apart and they are not perfectly perpendicular to each other. Another source of error is tilt in the floor causing the entire motor assembly to be tilted as well. This, however, means that the base of the motor assembly is at a different location than the top of the assembly where the laser beam actually exits. Another source of error is the possibility that the encoder could read a variation of one encoder pulse while the unit is stationary which would translate into an error of 0.0048 degrees.

CHAPTER 5: CONCLUSIONS

From the results found in the previous section, it is obvious that measuring position and orientation with this system is not 100% accurate, but since its error does not accumulate it is far better than the current positioning system. With a maximum error of 0.259 inches in position and 0.8 degrees in orientation, these error values are tolerable considering the workspace is 10 feet by 10 feet.

Some suggestions for future experiments would be to manufacture the flash detectors out of glass. Glass would probably increase the incident angle range at which the flash detectors are functional thus allowing for better data collection. Also, since it is intended for the device to be used on a cobot where there is a human operator involved, it will be necessary to have at least one more flash detector located throughout the workspace to allow data collection to continue even when the operator is blocking on of the flash detectors.

APPENDIX A: Analytical Solution

Three sensors are necessary to determine the cobot's position and orientation. These three sensors at known and fixed coordinates form a triangular workspace (See points A, B, and C in Figure A-1 below). The coordinates of a point (P) within the workspace can be determined if the angles between each sensor location are known with respect to the point namely ∠APB, ∠BPC, and ∠CPA. Calculating the position P is possible by finding the intersection of two circles defined by any 2 sensors and the unknown point (P) on the circle's perimeter. (Note: Three circles could be used to determine the same point, but the third circle is unnecessary because while there will be two solutions, one will always be a sensor location and the other will be the unknown point, P.)

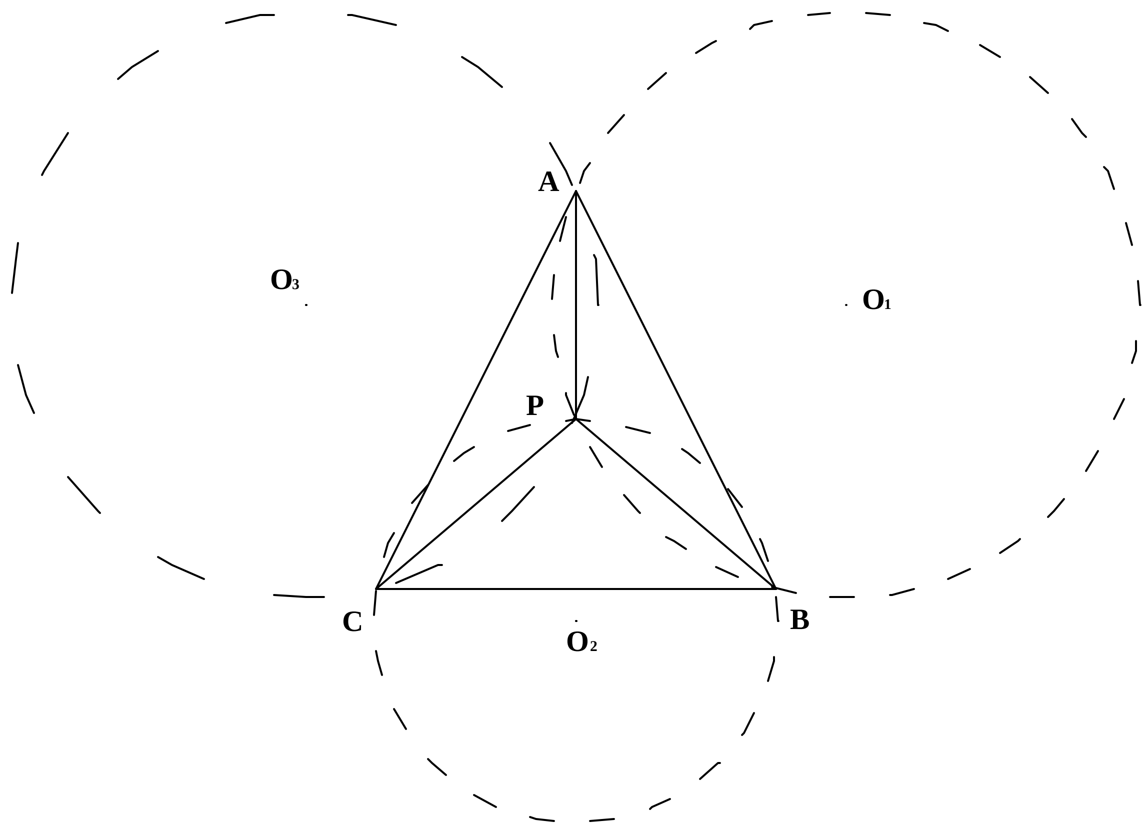

Figure A-1: Triangular Workspace Diagram

The equation of a circle that passes through (P, A, B) with center at (O_x, O_y) as shown in Figure A-2 below will be:

$(x - O_x)^2 + (y - O_y)^2 = R^2$ where R is the radius of the circle

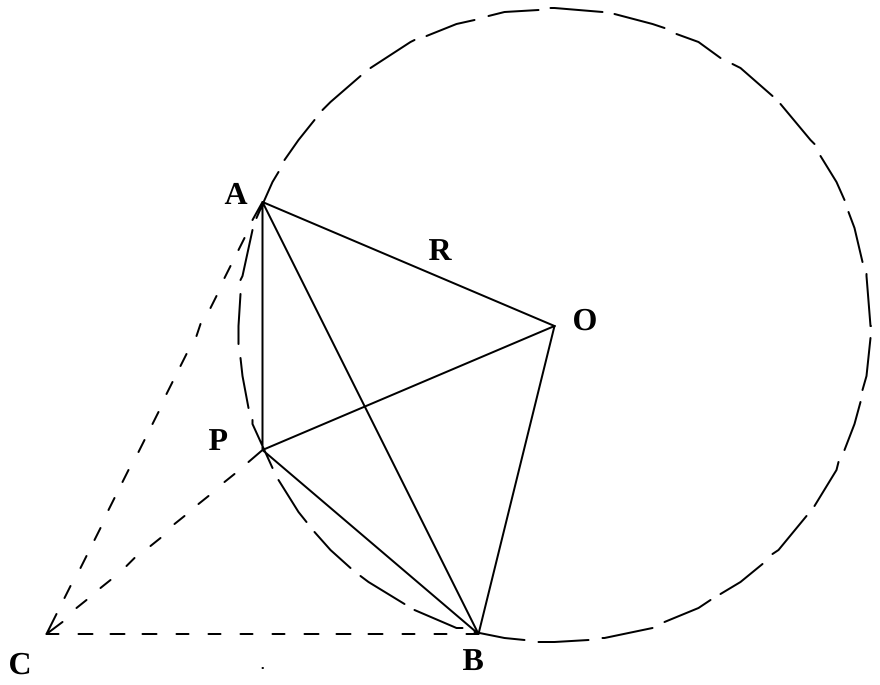

Figure A-2: Isosceles Triangles

Knowing $\angle APB$ and that any triangle made with point O and two other points on the circle's perimeter forms an isosceles triangle, the following relationships are true.

$\angle AOP = 180° - 2\angle APO$

$\angle POB = 180° - 2\ \angle OPB$

$\angle AOP + \angle POB = \angle AOB$

$\angle APO + \angle OPB = \angle APB$

Combining these equations gives:

$\angle AOB = 360° - 2\angle APB$

Knowing ∠AOB and the line segment AB, the radius of the circle can be calculated (See Figure A-3 below for diagram).

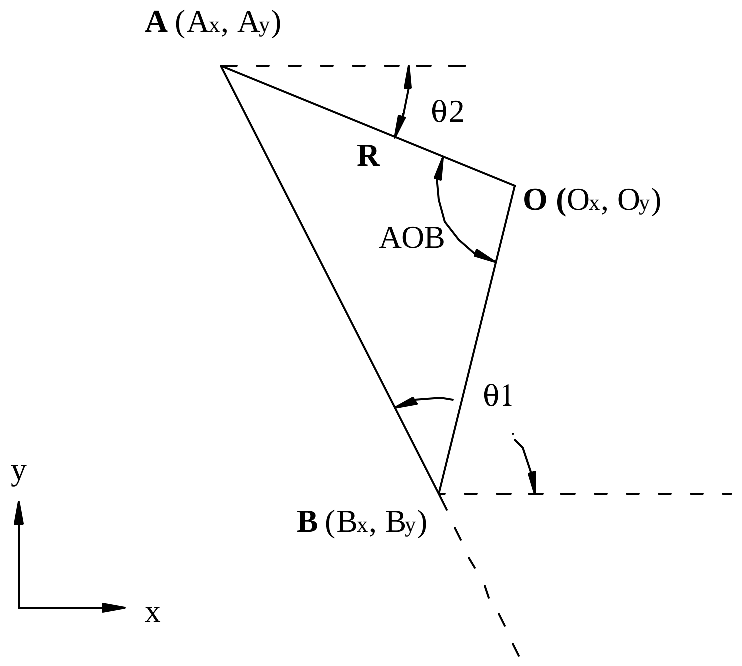

Figure A-3: Triangle Set-up

The equation for the radius is:

$$R = \frac{\left(\frac{\overline{AB}}{2}\right)}{\sin\left(\frac{\angle AOB}{2}\right)} \qquad \text{where } \overline{AB} = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$$

To determine the coordinates of the center of the circle (O_x, O_y), the slope of line segment AB must be determined.

$$\text{slope of } \overline{AB} = \frac{A_y - B_y}{A_x - B_x} = \frac{\sin q_1}{\cos q_1}$$

Rearranging the equations above gives:

$$q_1 = a\tan 2\left(\left(\mathrm{A_y} - \mathrm{B_y}\right), \left(\mathrm{A_x} - \mathrm{B_x}\right)\right) \quad \text{and} \quad q_2 = q_1 - 180 + \frac{180 - \angle \mathrm{AOB}}{2}$$

Therefore,

$$\mathrm{O_x} = \mathrm{A_x} + R\cos q_2 \quad \text{and} \quad \mathrm{O_y} = \mathrm{A_y} + R\sin q_2$$

This procedure must be performed for two of the three circles resulting in two equations that can be solved simultaneously in order to determine point P. In order to determine the cobot’s orientation, one of the three sensors would be used as a reference.

APPENDIX B: Program

```
/*****************************************************************
*       Triangulation Program:  This programs inputs are a user defined        *
*               function (usrfun) that calculates the system of equations      *
*               (fvec) using an inital guess input (x[1..n]). After deter-     *
*               mining the Jacobian (fjac) and performing LU decomposition     *
*               (ludcmp with lubksb), an iterative solution for position       *
*               and orientation (x[]) can be found.                            *
*       Programmer:     Michelle Simi                                          *
*       December 13, 2000                                                      *
*****************************************************************/


#include <math.h>
#include <malloc.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>


#define TINY 1.0e-20;
#define FREERETURN {free_matrix(fjac,1,n,1,n);free_vector(fvec,1,n);\
        free_vector(p,1,n);free_ivector(indx,1,n);return;}
#define NR_END 1

#define X1 129.75                       /* Flash detector coordinates */
#define Y1 -10

#define X2 10
#define Y2 -10

#define X3 69.875
#define Y3 -113.706542103

#define PI 3.14159265359
#define FREQ 0
#define BASE 0x300                      /* Base address of encoder board */

float xr[4], yr[4];
unsigned long count[4];
double theta[4];

/* Subroutine declarations */

void nrerror (char error_text[]);
float *vector(long nl, long nh);
void free_vector(float *v,long nl,long nh);
float **matrix(long nrl,long nrh,long ncl,long nch);
void free_matrix(float **m,long nrl,long nrh,long ncl,long nch);
int *ivector(long nl,long nh);
void free_ivector(int *v,long nl,long nh);
void ludcmp(float **a,int n,int *indx,float *d);
void lubksb(float **a, int n, int *indx, float b[]);
void usrfun(float var[], int n, float *fvec);
void fdjac(int n, float var[], float *fvec, float **fjac);
void mnewt(int ntrial,float x[], int n,float tolx, float tolf);

void find_theta();
void encinit();


/******************************************************************
*       MAIN: Uses the Newton-Raphson method for finding an iterative       *
*               solution for n number of variables by using an initial      *
*               guess, x, to determine the values of the system of          *
*               equations, fvec, and the Jacobian, fjac. It will perform    *
*               ntrial number of iterations until either the sum of the     *
*               magnitudes of fvec are within tolf tolerance or when the    *
*               sum of the absolute values of the changes in each variable  *
*               is less than tolx tolerance.                                *
******************************************************************/

void main()
{
        int *indx, ntrial=100, n=3;                     /* Defines number of trials and variables */
        float errx, errf, d, *fvec, **fjac, *p;
        float tolx=0.00001, tolf=0.00001;               /* Sets tolerances */
        FLOAT X[4];                                     /* Variables to be solved */
        FILE *output;
   output=fopen("output.txt", "w");


        xr[1]=X1; xr[2]=X2; xr[3]=X3;                   /* Sets the flash detector locations */
        yr[1]=Y1; yr[2]=Y2; yr[3]=Y3;
   x[1]=57; x[2]=-57.5; x[3]=0.1;       /* Initial guess */
        theta[0]=0; theta[1]=0; theta[2]=0; theta[3]=0;

        encinit();                                      /* Initializes the encoder decoder board */

        outp (BASE+1, 0x82);    /* resets counters for Ch. 1 & 2 */
        outp (BASE+5, 0X82);    /* resets counters for Ch. 3 & 4 */

        while (!kbhit())
        {
                find_theta();

                mnewt(ntrial, x, n, tolx, tolf);
                printf("Answer: X=%f\tY=%f\tTheta=%f\n", x[1], x[2], x[3]);
                fprintf(output, "%.4f\t%.4f\t%.4f\n", x[1], x[2], x[3]);
        }

        fclose (output);
}
```

```
/*****************************************************************
*          NRERROR: Prints error messages if variables are unsolvable and          *
*                    exits the program.                                             *
*****************************************************************/

void nrerror(char error_text[])
{
          printf("Numerical Recipes run-time error...\n");
          printf("%s\n",error_text);
          printf("...now exiting to system...\n");
          exit(1);
}


/*****************************************************************
*          *VECTOR: Allocates memory space for v[nl..nh] float vector.             *
*****************************************************************/


float *vector(long nl, long nh)
{
          float *v;

          v=(float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
          if (!v) nrerror("allocation failure in vector()");
          return v-nl+NR_END;
}


/*****************************************************************
*          FREE_VECTOR: Frees memory space set aside for *v float vector.         *
*****************************************************************/

void free_vector(float *v,long nl,long nh)
{
          free((char*) (v+nl-NR_END));
}


/*****************************************************************
*          **MATRIX: Allocates memo ry space for m[nrl..nrh][ncl..nch]             *
*                    matrix.                                                        *
*****************************************************************/

float **matrix(long nrl,long nrh,long ncl,long nch)
{
          long i, nrow=nrh+nrl+1, ncol=nch-ncl+1;
          float **m;

          m=(float **) malloc((size_t) ((nrow+NR_END)*sizeof(float*)));

          if (!m) nrerror ("allocation failure 1 in matrix()");
          m += NR_END;
          m -= nrl;
```

```
	m[nrl]=(float *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(float)));
	if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
	m[nrl] += NR_END;
	m[nrl] -= ncl;

	for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;
	return m;

}


/*****************************************************************
*	FREE_MATRIX: Frees memory space set aside for **m matrix.		*
*****************************************************************/

void free_matrix(float **m,long nrl,long nrh,long ncl,long nch)
{
	free((char*) (m[nrl]+ncl-NR_END));
	free((char*) (m+nrl-NR_END));
}


/*****************************************************************
*	*IVECTOR: Allocates memory space for v[nl..nh] int vector.		*
*****************************************************************/

int *ivector(long nl,long nh)
{
	int *v;

	v=(int *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(int)));
	if (!v) nrerror("allocation failure in ivector()");
	return v-nl+NR_END;
}


/*****************************************************************
*	FREE_IVECTOR: Frees memory space set aside *v int vector.		*
*****************************************************************/

void free_ivector(int *v,long nl,long nh)
{
	free((char*) (v+nl-NR_END));
}


/*****************************************************************
*	LUDCMP: Transforms a n by n matrix into rowwise permutation of	*
*		itself using LU decomposition.  Used in conjunction with	*
*		LUBKSB.							*
*****************************************************************/

void ludcmp(float **a,int n,int *indx,float *d)
{
	int i,imax,j,k;
	float big,dum,sum,temp;
```

```
        float *vv;

        vv=vector(1,n);
        *d=1.0;

        for (i=1;i<=n;i++)
        {
                big=0.000;
                for (j=1;j<=n;j++)
                        if ((temp=fabs(a[i][j])) > big) big=temp;
                if (big==0.00) nrerror("Singular matrix routine ludcmp");
                vv[i]=1.0/big;
        }

        for (j=1;j<=n;j++)
        {
                for (i=1;i<j;i++)
                {
                        sum=a[i][j];
                        for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                }
                big=0.0;
                for (i=j;i<=n;i++)
                {
                        sum=a[i][j];
                        for (k=1;k<j;k++)
                                sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
                        if ( (dum=vv[i]*fabs(sum)) >= big)
                        {
                                big=dum;
                                imax=i;
                        }
                }
                if (j != imax)
                {
                        for (k=1;k<=n;k++)
                        {
                                dum=a[imax][k];
                                a[imax][k]=a[j][k];
                                a[j][k]=dum;
                        }
                        *d = -(*d);
                        vv[imax]=vv[j];
                }
                indx[j]=imax;
                if (a[j][j] == 0.0) a[j][j]=TINY;
                if (j != n)
                {
                        dum=1.0/(a[j][j]);
                        for (i=j+1;i<=n;i++) a[i][j] *= dum;
                }
        }
        free_vector(vv,1,n);
}
```

```
/*****************************************************************
*           LUBKSB: Solves the set of n linear equations.                       *
*****************************************************************/


void lubksb(float **a, int n, int *indx, float b[])
{
            int i,ii=0,ip,j;
            float sum;

            for (i=1;i<=n;i++) {
                        ip=indx[i];
                        sum=b[ip];
                        b[ip]=b[i];
                        if (ii)
                                    for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
                        else if (sum) ii=i;
                        b[i]=sum;
            }
            for (i=n;i>=1;i--) {
                        sum=b[i];
                        for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
                        b[i]=sum/a[i][i];
            }
}



/*****************************************************************
*           USRFUN:  Calculates the system of equations, fvec, given an           *
*                       initial guess, x.                                                 *
*****************************************************************/


void usrfun(float var[], int n, float *fvec)
{
            int i;
            for (i=1; i<=n; i++)
            {
                        fvec[i]= atan2((yr[i]-var[2]),(xr[i]-var[1])) - var[3] - theta[i]; /* Fi */
            }
}


/*****************************************************************
*           FDJAC:  Finds the Jacobian matrix, df, using an initial guess,          *
*                       x, and the system of equations, fvec.                           *
*****************************************************************/

void fdjac (int n, float x[], float *fvec, float **df)
{
            int i,j;
            float h,temp,*f;
```

```
        f=vector(1,n);
        for(j=1;j<=n;j++)
        {
                temp=x[j];
                h= 0.0001 * fabs(temp);
                if (h==0) h=0.0001;
                x[j]=temp+h;
                h=x[j]-temp;
                usrfun(x,n,f);
                x[j]=temp;
                for(i=1;i<=n;i++) df[i][j]=(f[i]-fvec[i])/h;
        }
        free_vector(f,1,n);
}


/*****************************************************************
*       FIND_THETA: Reads all four channels output latches to get          *
*               encoder counts that can then be used to find theta[i].     *
*****************************************************************/

void find_theta()
{
        int i, n, m, diff[4];
        double thetapre[4], delta[4];

        diff[1]=12600; diff[2]=12600; diff[3]=12600;
        count[1]=0; count[2]=0; count[3]=0;

        for (i=1; i<4;i++)   thetapre[i]=theta[i];

        while (count[1]==0 || count[2]==0 || count[3]==0)
        {
                for (i=0;i<4;i++)
                {
                        outp (BASE+(i*2)+1, 0x01);              /* Reset byte pointer */

                        count[i] = inp (BASE+(i*2));            /* Reads the 24bit counter */
                        count[i] |= ((inp(BASE+(i*2))) << 8);
                        count[i] |= ((inp(BASE+(i*2))) << 16);
                }
                for (m=1;m<4;m++)
                {
                        diff[m]=count[m]-count[0];
                        if (abs(diff[m])>12500) count[m]=0;     /* Accounts for missing counts and
                                                                 when counter rolls over */
                }
                if (kbhit()) break;
        }

        FOR (I=1;I<4;I++)          /*  This entire for loop accounts for multiple 360deg
                                   rotations by the mobile robot in either direction  */
        {
                if (count[0] > count[i]) count[i]=count[i]+12500;
                diff[i]=count[i]-count[0];
```

```
		theta[i]=((double)(diff[i])/12500)*2*PI;

		if (((2*PI)-theta[i]) < theta[i]) theta[i]=theta[i]-(2*PI);
		delta[i]=theta[i]-thetapre[i];

		if ((abs(delta[i])) > PI)
		{
			if (delta[i] > PI) delta[i]=delta[i] - (2*PI);
			else delta[i]=delta[i] + (2*PI);
		}
		if (delta[i] > PI) delta[i]=delta[i]-(2*PI);
		else if (delta[i] < -PI) delta[i]=delta[i]+(2*PI);

		theta[i]=thetapre[i]+delta[i];
	}

}


/*****************************************************************
*	MNEWT: Calls functions to perforM LU decomposition and checks	*
*		for root and function convergences, tolx and tolf.		*
*****************************************************************/

void mnewt(int ntrial,float x[],int n,float tolx,float tolf)
{
	int k,i,*indx;
	float errx,errf,d,*fvec,**fjac,*p;

	indx=ivector(1,n);
	p=vector(1,n);
	fvec=vector(1,n);
	fjac=matrix(1,n,1,n);
	for (k=1;k<=ntrial;k++)
	{
		usrfun(x,n,fvec);		/* Gets user defined input fvec */
		fdjac(n,x,fvec,fjac);		/* Calculates the Jacobian, fjac */
		errf=0.0;
		for (i=1;i<=n;i++) errf += fabs(fvec[i]);
		if (errf <= tolf) FREERETURN			/* Checks for function convergence */
		for (i=1;i<=n;i++) p[i]= -fvec[i];
		ludcmp(fjac,n,indx,&d);			/* Performs LU Decomposition */
		lubksb(fjac,n,indx,p);
		errx=0.0;
		for (i=1;i<=n;i++)
		{
			errx += fabs(p[i]);
			x[i] += p[i];
		}
		if (errx <= tolx) FREERETURN		/* Checks for root convergence */
	}
	FREERETURN
}
```

```
/****************************************************************
*       ENCINTI: Initializes the encoder board to load the counter          *
*               value to the output latch whenever the index pulse occurs.  *
*               It also set the count mode to normal and X1.                *
****************************************************************/

void encinit()
{
        int i, n, m;
        long int Preset=0;
        unsigned char ByteVal;

        /* Initialize the quad encoder */
        outp (BASE+8, 0x11);        /* Sets Ch.1 and 2 to LCNTR/LOL */
        outp (BASE+9, 0x11);        /* Sets Ch.3 and 4 to LCNTR/LOL */

        for (i=0; i<4; i++)
        {
                outp (BASE+(i*2)+1, 0x04);          /* Reset error counters */
                outp (BASE+(i*2)+1, 0x06);          /* Reset error flags */
                outp (BASE+(i*2)+1, 0x02);          /* Reset counter */

                outp (BASE+(i*2), FREQ);            /* Sets FCK frequency */
                outp (BASE+(i*2)+1, 0x18);          /* Transfers frequency to filter clock */

                /* Configure the registers */
                outp (BASE+(i*2)+1, 0x20 | 0x08);               /* Sets count mode to normal, X1 */
                outp (BASE+(i*2)+1, 0x40 | 0x01 | 0x02);        /* Load to OL, enable A/B */
                outp (BASE+(i*2)+1, 0x60 | 0x01 | 0x02);        /* Positive indexing, LOL */
                outp (BASE+(i*2)+1, 0x00 | 0x02 | 0x10);        /* Reset counter and transfer
                                                                   counter to output latch  */

                /* Preset Counter */
                outp (BASE+(i*2)+1, 0x01);          /* Resets the byte pointer */

                for (n=0;n<4;n++)
                        for (m=0;m<3;m++)
                        {
                                ByteVal = (unsigned char) ((Preset >> (m*8)) & 0xFF);
                                outp (BASE+(n*2), ByteVal);
                        }

                outp (BASE+(i*2)+1, 0x08);          /* Transfer Preset to the Counter */
        }
}
```

APPENDIX C: How to Mirrorize

The following instructions are based on the instructions for silvering given by the Amateur Telescope Making webpage.

(Some notes concerning the mirrorizing process outlined here. After running numerous trials, some things became apparent. First of all, glass mirrorized better than plexiglass in the sense that the coating was more uniform and had greater reflectivity. Also, it is important to use the nitric acid in the cleaning step because the coating clings better with a clean surface. When applying a finishing coat to the silvered sides, make sure that they everything is completely dry and be cautious about what coating you choose. Some coatings may react with the silvering solution such as polyurethane. The crucial step in getting a good finish, however, is making sure that piece is fully immersed when all of the solutions are combined. The first couple of seconds of the reaction really seem to make a difference in the coating results.)

Materials and Equipment needed

- Dishwashing detergent, liquid
- Nitric Acid, specific gravity 1.4
- Ammonium hydroxide, specific gravity 0.9
- Potassium hydroxide, pellet form
- Silver nitrate, crystal form
- Dextrose
- Distilled water
- 4 – 600mL beakers
- Graduated cylinder
- Glass stirring rod
- Glass dropper bottle
- Glass funnel for filling dropper bottle
- Dish big enough to fit piece to be mirrorized
- Scale to weigh out grams
- Rubber gloves (certified to handle the above chemicals)
- Apron (certified to handle the above chemicals)
- Goggles
- Pliers
- Can of spray shellac
- Cotton balls
- Masking tape

Directions

1) Scrub all utensils (glassware, pliers, and gloves) with the dishwashing detergent. Rinse with tap water, then scrub again. Dry everything.
2) Make up the following solutions:
   a) Dissolve 11 grams of potassium hydroxide in 185mL of distilled water.

b) Dissolve 11 grams of dextrose in 90mL of distilled water.
c) Dissolve 16.5grams of silver nitrate in 185mL of distilled water.
d) Put 75mL of distilled water in the fourth beaker.

Stir each solution until each ingredient is completely dissolved. Let these solutions sit while doing the rest of the procedures so they can cool. (In the chemical reaction of dissolving the chemicals in the distilled water, the solutions will heat up slightly.)

3) Use the glass funnel to fill the glass dropper bottle with ammonium hydroxide. Wash the funnel immediately and make sure the dropper bottle is stoppered until use.
4) Scrub the piece to be mirrorized thoroughly. Rinse with tap water.
5) Put some nitric acid on a cotton ball and clean the piece. When done, rinse with large amounts of water flushing everything down the sink. *Concentrated nitric acid is a dangerous chemical: it will attack anything except glass and rubber, so keep it away from contact with skin, clothing, or anything else.*
6) Rinse piece with distilled water and place in the dish. Fill dish with enough water to cover piece completely.
7) Now begin to add ammonium hydroxide a couple drops at a time to the silver nitrate solution. The whole solution will turn brown. Stir constantly and continue to add ammonium hydroxide until the solution shows signs of clearing. As the solution clears, add only one drop at a time. When the solution appears grayish or slightly cloudy, stop. *An excess of ammonium hydroxide will ruin all subsequent operations*. No harm is done if the solution contains more silver nitrate than theory permits, but too much ammonium hydroxide is fatal.
8) Add all of the potassium hydroxide solution to the silver nitrate / ammonium hydroxide solution, a little at a time, with constant stirring. Goggles should be on for this operation. *Too rapid an addition of the potassium hydroxide may cause an explosion since a small amount of silver fulminate is formed at this stage.* However, there is little likelihood of such an event if the potassium hydroxide is added slowly enough.
9) Once more add ammonium hydroxide, until the brown or black appearance of the solution produced in the previous step begins to clear up. When the clearing starts, proceed very cautiously, again adding the ammonium hydroxide drop by drop. You are aiming for a solution that will look like weak tea and which more than likely will contain a myriad of small black specks. The silvering solution is now complete and ready to use.
10) Pick up the piece to be mirrorized out of the dish. Put pliers on the end that is not going to be mirrorized and place masking tape along the front edge and around to bottom so they don't get mirrorized.
11) Place piece, while holding it with the pliers, in the beaker that only contains 75mL of water.
12) Pour the dextrose solution into the beaker.
13) Pour the silvering solution into the beaker.
14) Keep piece from touching the walls, but wiggle the piece slightly.
15) The solution will first turn dark brown, then lighter brown, and then you should be able to see that the sides of the beaker are starting to mirrorize. Sliver-brown flakes will form on the solution surface. When the outside of the mirror looks like a good mirror (about 5 minutes) remove the piece.

16) Rinse it with distilled water and let dry in upright position so that no silverized portion is touching anything (do not touch silver, it will flake right off).
17) When dry, spray piece with shellac. When dry, remove tape.

APPENDIX D: Encoder Output Amplification Schematic

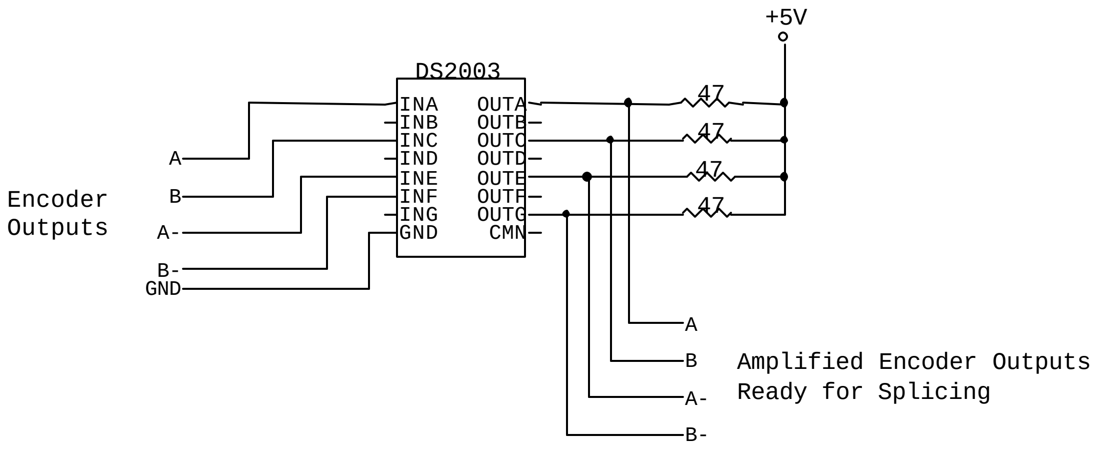

Figure D-1: Schematic for Amplifying Encoder Outputs

APPENDIX E: Flash Detector Set-up Procedure

For an equilateral triangle with side A and the motor assembly as some location within the triangle, the position of the motor assembly can be calculated. This requires measuring the distances between the motor assembly and the three flash detectors (B, C and D) and defining an arbitrary reference frame. (See Figure D-1 below.)

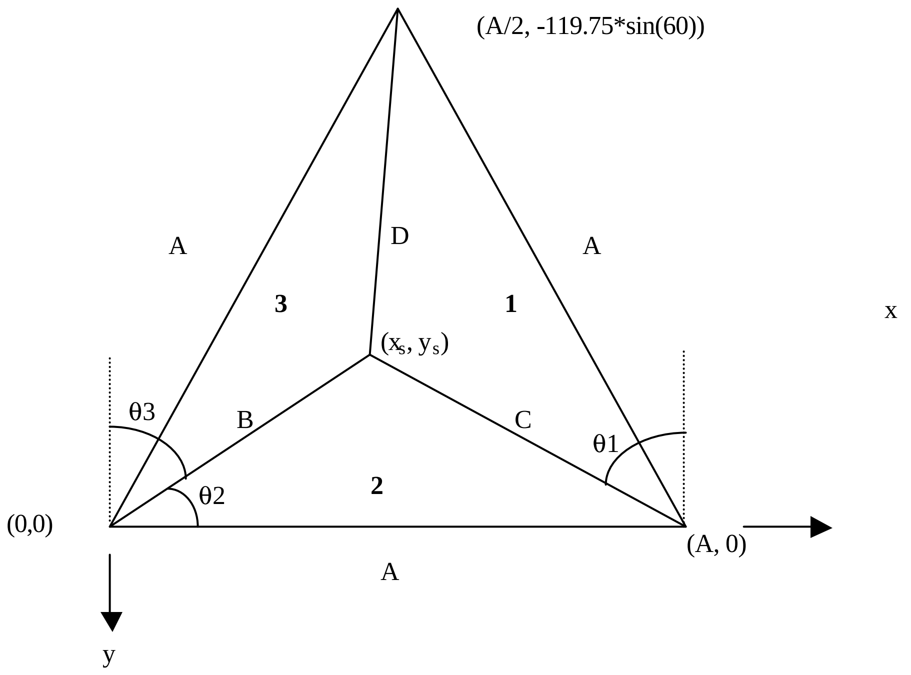

Figure E-1: Flash Detector Coordinate Set-Up

So using the law of cosines, the necessary angles can be found as follows:

$$q1 = 30 + \cos^{-1}\left(\frac{C^2 + A^2 - D^2}{2AC}\right)$$

$$q2 = \cos^{-1}\left(\frac{B^2 + A^2 - C^2}{2AB}\right)$$

where qi is in degrees for $i = 1,2,3$

$$q3 = 30 + \cos^{-1}\left(\frac{B^2 + A^2 - D^2}{2AB}\right)$$

Knowing the angles, you can now calculate the position (x_s, y_s) using three different triangles by using the following equations:

Triangle 1:

$$x_s = A - C\sin(q1)$$
$$y_s = -C\cos(q1)$$

Triangle 2:

$$x_s = B\cos(q2)$$
$$y_s = -B\sin(q2)$$

Triangle 3:

$$x_s = B\sin(q3)$$
$$y_s = -B\cos(q3)$$

These three values can be compared and their difference is the error in the flash detector positions. For the set-up, the following measurements in inches were used:

A=119.75, B=67.40625, C=86.65625, D=57.25

The result is three different coordinates for x_s and y_s. Using the numbers given above, the coordinates where (47.479, -47.815), (47.492, -47.834) and (47.521, -47.805). Therefore, average location is (47.497, -47.818) with an absolute difference of 0.04185 inches in the x-direction and 0.028756 inches in the y-direction.

APPENDIX F: Tabulated Results

Tile Values		Optical Values		Difference		Absolute % Diff.	
x [in]	**y [in]**	**x[in]**	**y[in]**	**Dx [in]**	**Dy [in]**	**Dx [%]**	**Dy [%]**
57.46535	-58.11582	57.46535	-58.11582	0.000	0.000	-	-
57.465	-46.116	57.553	-46.103	0.087	0.013	0.152	0.029
57.465	-34.116	57.594	-34.031	0.128	0.085	0.223	0.250
57.465	-22.116	57.601	-22.116	0.136	0.000	0.236	0.001
45.465	-22.116	45.562	-22.051	0.096	0.065	0.212	0.294
33.465	-22.116	33.598	-22.010	0.133	0.106	0.397	0.480
33.465	-46.116	33.540	-45.955	0.074	0.161	0.222	0.349
45.465	-46.116	45.525	-46.035	0.060	0.080	0.132	0.174
45.465	-34.116	45.531	-34.043	0.065	0.073	0.143	0.213
33.465	-34.116	33.653	-33.883	0.188	0.233	0.562	0.682
45.465	-58.116	45.515	-58.016	0.049	0.100	0.109	0.172
45.465	-70.116	45.532	-69.998	0.067	0.118	0.147	0.168
57.465	-70.116	57.490	-70.019	0.024	0.097	0.042	0.138
57.465	-82.116	57.534	-82.043	0.069	0.073	0.119	0.088
81.465	-58.116	81.539	-58.145	0.073	-0.029	0.090	0.049
81.465	-82.116	81.556	-82.136	0.091	-0.021	0.112	0.025
81.465	-94.116	81.385	-94.133	-0.080	-0.017	0.098	0.018
81.465	-46.116	81.570	-46.091	0.104	0.025	0.128	0.054
81.465	-22.116	81.536	-22.024	0.070	0.092	0.086	0.415
81.465	-34.116	81.619	-34.077	0.154	0.038	0.189	0.113
81.465	-70.116	81.541	-70.112	0.076	0.003	0.093	0.005
93.465	-70.116	93.438	-70.183	-0.027	-0.067	0.029	0.096
93.465	-82.116	93.332	-82.249	-0.134	-0.133	0.143	0.162
93.465	-46.116	93.539	-46.142	0.074	-0.026	0.079	0.056
93.465	-34.116	93.524	-34.112	0.059	0.004	0.063	0.011
93.465	-22.116	93.473	-22.072	0.008	0.043	0.008	0.196
105.465	-22.116	105.399	-22.165	-0.066	-0.049	0.063	0.223
105.465	-34.116	105.529	-34.192	0.063	-0.076	0.060	0.224
105.465	-46.116	105.536	-46.296	0.070	-0.180	0.067	0.391
105.465	-58.116	105.527	-58.375	0.062	-0.259	0.058	0.446
93.465	-58.116	93.539	-58.178	0.073	-0.062	0.078	0.107
117.465	-34.116	117.477	-34.354	0.012	-0.238	0.010	0.697
117.465	-22.116	117.487	-22.167	0.021	-0.052	0.018	0.233
69.465	-46.116	69.547	-46.092	0.082	0.023	0.117	0.051
69.465	-34.116	69.580	-33.981	0.115	0.134	0.165	0.394
69.465	-22.116	69.549	-22.078	0.084	0.038	0.120	0.173
69.465	-58.116	69.572	-58.054	0.107	0.062	0.154	0.107
69.465	-70.116	69.605	-69.993	0.140	0.122	0.201	0.175
69.465	-82.116	69.531	-82.002	0.066	0.114	0.095	0.139
69.465	-94.116	69.455	-93.997	-0.010	0.118	0.015	0.126
81.465	-94.116	81.358	-94.064	-0.107	0.052	0.131	0.056

Figure F-1: Table of Tile Values vs. Optical Values

BIBLIOGRAPHY

Colgate, J. E., Wannasuphoprasit, W., and Peshkin, M. A. “Cobots: Robots for Collaboration with Human Operators,” *ASME International Mechanical Engineering Congress and Exposition*, Atlanta, Vol. 58, pp. 433-439, 1996.

Flannery, Brian P., Press, William H., Teukolsky, Saul A. and Vetterling, William T., Numerical Recipes in C The Art of Scientific Computing. 2nd ed. New York: Cambridge University Press New York, 1992.

Halliday, David, Robert Resnick and Jearl Walker. Fundamentals of Physics Extended. 5th ed. John Wiley & Sons, Inc., USA, 1997.

Mills, A. L. “An Analysis of Position and Velocity Sensor Systems for a 3-Degree-of-Freedom Planar Collaborative Robot” Master of Science in Mechanical Engineering, Northwestern University, 1998.

Parabolic Reflectors Page. Mississippi State University. May 2000. <http://csmt.msstate.edu/lw/html/resources/parab.html>.

Silvering Page. Amateur Telescope Making. June 2000. <http://www.atmpage.com/silver.html>.

Wannasuphoprasit, W., Akella, P., Peshkin, M., and Colgate, J. E. “Cobots: A Novel Material Handling Technology,” *ASME International Mechanical Engineering Congress and Exposition*, Anaheim, pp. 1-7, 1998.

Wannasuphoprasit, W., Gillespie, R. B., Colgate, J. E., and Peshkin, M. A. “Cobot Control,” *IEEE International Conference on Robotics and Automation*, Albuquerque, Vol. 4, pp. 3571-3576, 1997.

Zwillinger, Daniel. Standard Mathematical Tables and Formulae. 30th ed. Boca Raton: CRC Press, Inc., 1996.