Mobile Robot Navigation with Human Interface Device

David Buckles
Brian Walsh

Advisor: Dr. Aleksander Malinowski

ABSTRACT

This project utilizes a Pioneer 3D-X robot chassis equipped with ultrasonic and infrared sensors to map its environment either autonomously or manually. There are two autonomous modes: random exploration and potential field planning to a particular goal. The mapped area is then animated in 3D using OpenGL and transmitted to an LCD eyepiece where it is displayed. The manual override functionality utilizes a sensor glove to direct the movement of the robot. The glove can also be used to control a robotic arm with grasper mounted on the front of the 3D-X chassis. Toggling between modes is handled by feature recognition algorithms that detect finger location and movement from the sensor glove while head movement is detected by a gyroscope, magnetometer, and accelerometer mounted on the LCD eyepiece. Sensor fusion is achieved via an adaptive filter applied to the sensors mounted on the LCD eyepiece to reduce sensor drift.

TABLE OF CONTENTS

INTRODUCTION

The Pioneer 3D-X series robot has been used at Bradley University for previous senior projects; the most recent being a project that focused on mapping and navigating a combat environment. This project expands upon said work by retaining the automated functionality and adding an override feature that will enable users to control the robot with feedback from a glove with gyroscopic and other forms of sensors. Also, data from the sensors will be mapped in 3D in real time and displayed to the user via an LCD eyepiece.

The robot currently uses 8 sonar sensors to gather data from plus or minus ninety degrees, or 5 infrared sensors already mounted on the Pioneer chassis with the same range.

The overall purpose for integrating these features into the Pioneer chassis is to create an agent that can operate in a combat environment autonomously or be controlled by the user in real time while mapping the environment and relaying sensor information in an easily comprehensible visual format.

SYSTEM DESCRIPTION

HUMAN MOUNTED SUBSYSTEMS

The first subsystem is the sensor glove. The sensors mounted on the glove will provide pitch, yaw, and rotational feedback, as well as finger position and tracking for overall hand movement.

The second subsystem is the human mounted laptop, which will interpret sensor data from the sensor glove, wirelessly control robot movement, and take the mapped data from the robot mounted laptop and display it on a user eyepiece.

The LCD eyepiece displays the OpenGL environment. The yet to be implemented eyepiece sensors would measure head angle and then modify the users view of the robot's perceived environment.

ROBOT MOUNTED SUBSYSTEMS

The robot mounted laptop takes the control signal from human mounted laptop and directs the robot. It also takes position and map data from the robot and sends it back to the human mounted laptop. The laptop also sends the grasping device microcontroller commands to move the actual grasping device.

The Pioneer robot subsystem gathers data about its surroundings using ultrasonic and infrared sensors and transmits this data to the mounted laptop, as well as receives commands from the laptop for either navigational mode.

The infrared microcontroller takes a command from the robot to begin measurements using the infrared sensors. The received data is then sent back to the robot.

The infrared sensors send beams of light, and measures the distance of nearby walls by measuring the time elapsed.

The grasping device microcontroller takes a command signal from the laptop, interprets it, and then sends out 5 PWM signals to the robotic arm grasping device, to control the 5 servo motors.

Finally, the grasping device takes the PWM signals from the grasping device microcontroller, and changes position due to the PWM controlled servo motors.

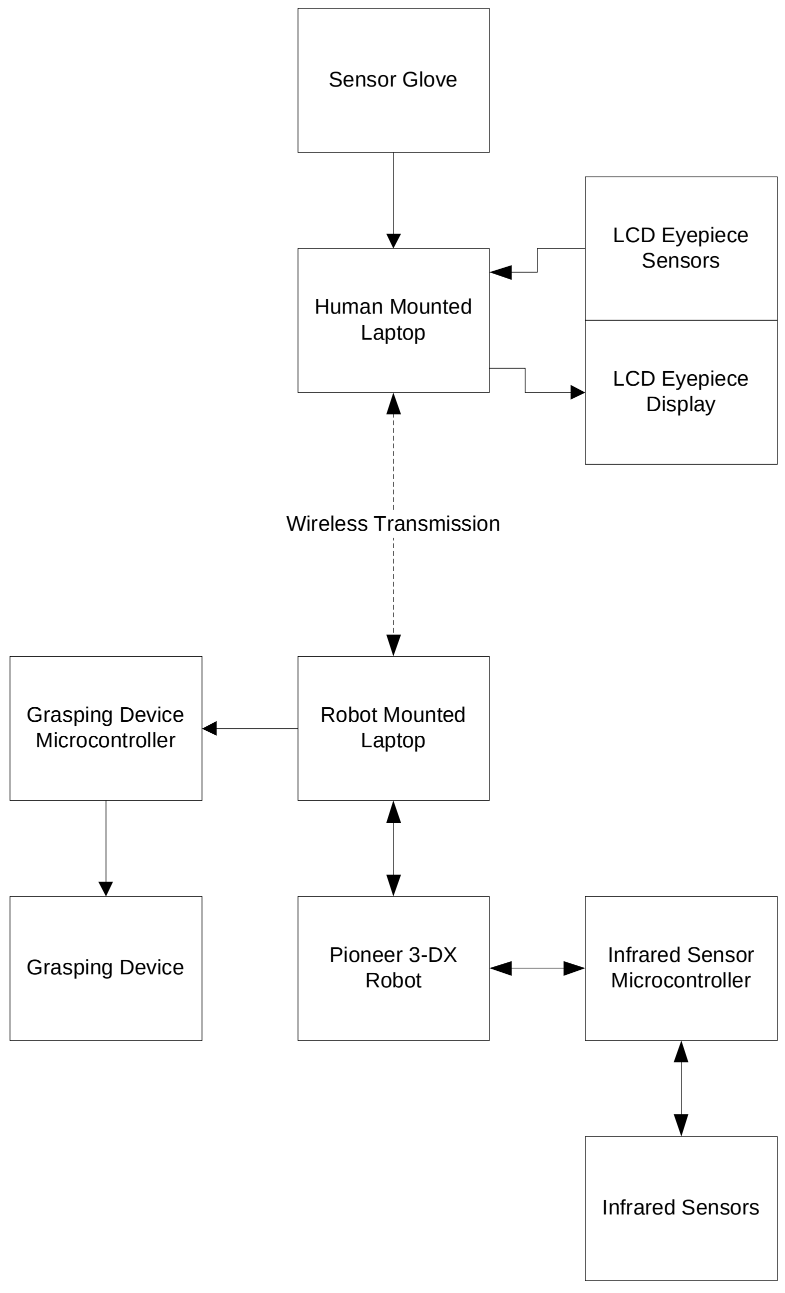


Fig. 1 – Functional System Block Diagram

DESIGN EQUATIONS, CALCULATIONS, AND FLOWCHARTS

DATAGLOVE

One goal of this project was to incorporate a reasonably intuitive human interface into the control scheme of an autonomous mobile agent in order to allow manual override with visual feedback in real time. To this end, a DG5-V dataglove was employed. This particular dataglove was selected based on its low cost, wide range of versatility, ease of communication, and sensing ability. The DG5-V has five potentiometers each with 1024 positions per finger as well as a 3-axis accelerometer with 3g sensitivity. There is also an on-board microcontroller that relays the data in 20-byte cycles through a serial or USB interface, though there is also a Bluetooth model. The protocol of the UART communication is shown below in Figure 2, as well as the values of the 20-byte cycle. An image of the dataglove is shown page 9 in Figure 3. A flowchart outlining the procedure of the software for interfacing the glove is on page 9 and the serial port code is located in Appendix A.

RS232 Serial Port Setting:
Baud Rate: 115200 BPS
Data Bit: 8
Stop Bit: 1
Parity: NONE
– start transmission (PC to datagllve): send 's' to the glove;
– (Data glove to PC): the glove transmits the package continuously;
– stop transmission (PC to dataglove): send 'e' to the glove;
Package structure:
The glove continuously transmits the PC the following 20 byte package:
1 - header = 0x20
2 - header = 0x0A
3 - lenght = 0x14 (20 byte)
4 - acc axis x_l
5 – acc axis x_h
6 - acc axis y_l
7 - acc axis y_h
8 - acc axis z_l
9 - acc axis z_h
10- bend 0_l
11 - bend 0_h
12 - bend 1_l
13- bend 1_h
14- bend 2_l
15- bend 2_h
16 - bend 3_l
17 - bend 3_h
18 - bend 4_l
19 - bend 4_h
20 – crc
CRC represents the XOR of the first 19 bytes.
Bend value are from 0 to 1023 so only 10 bit are used.

Fig. 2 – UART Communication Protocol

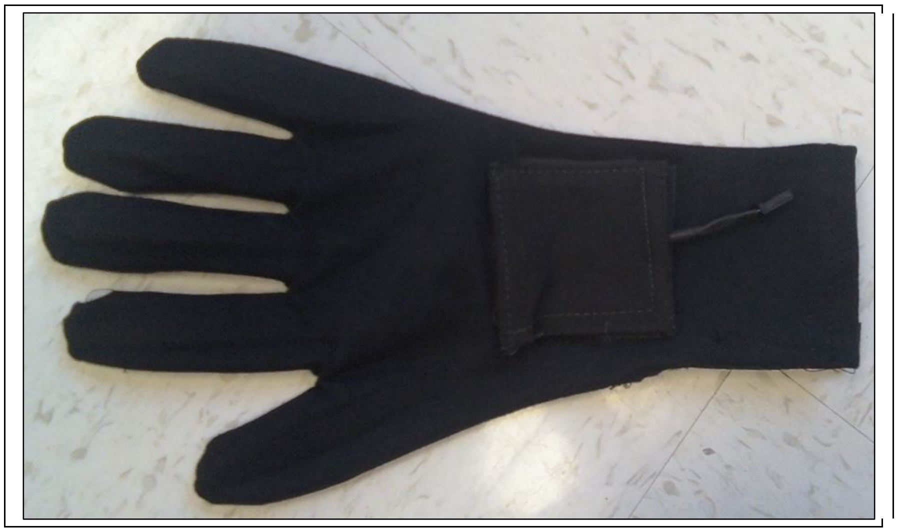


Fig. 3 – Data (Sensor) Glove

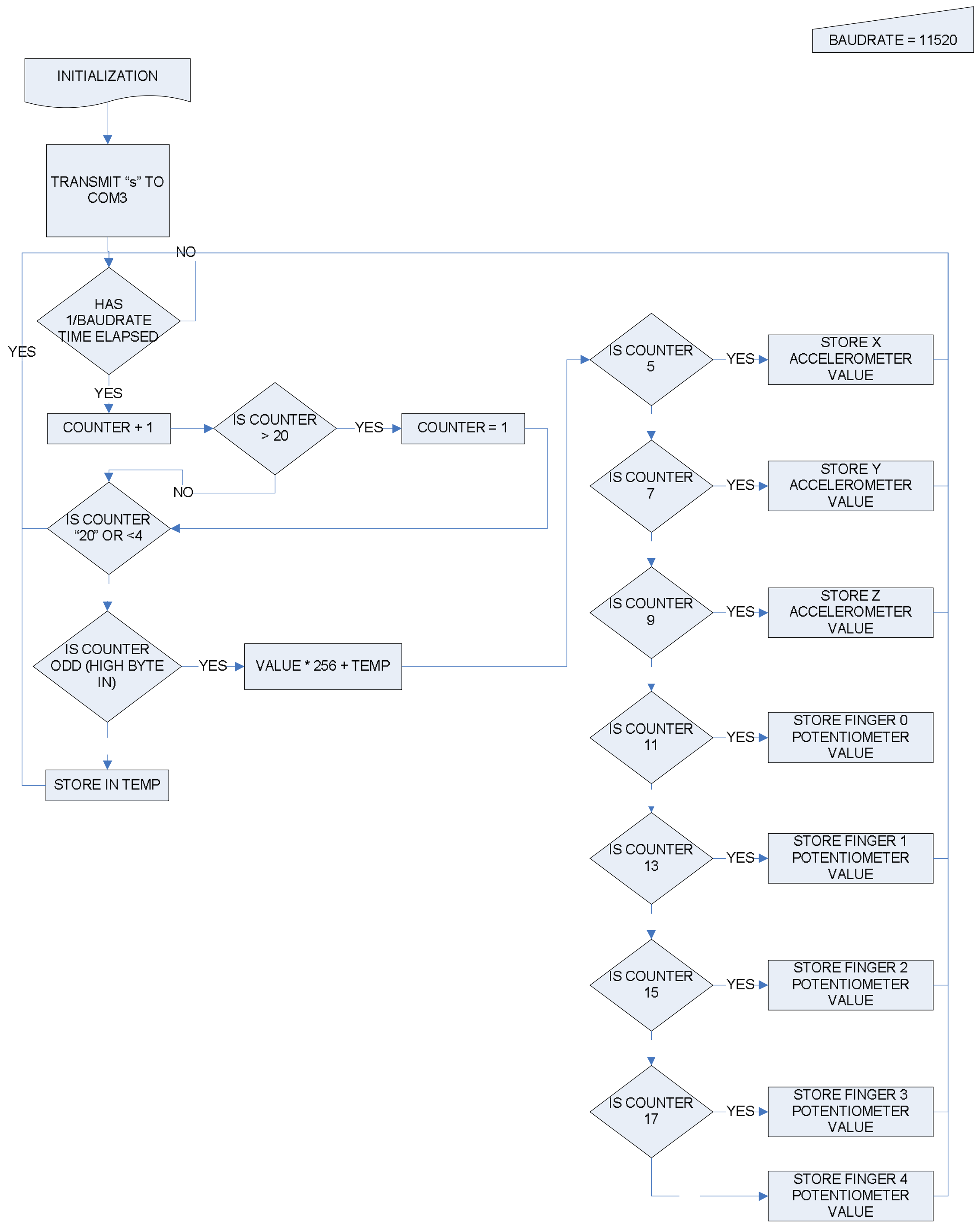

BAUDRATE = 11520
INITIALIZATION
TRANSMIT "s" TO COM3
NO
HAS 1/BAUDRATE TIME ELAPSED
YES
YES
COUNTER + 1
IS COUNTER > 20
YES
COUNTER = 1
NO
IS COUNTER "20" OR <4
IS COUNTER ODD (HIGH BYTE IN)
YES
VALUE * 256 + TEMP
STORE IN TEMP
IS COUNTER 5
YES
STORE X ACCELEROMETER VALUE
IS COUNTER 7
YES
STORE Y ACCELEROMETER VALUE
IS COUNTER 9
YES
STORE Z ACCELEROMETER VALUE
IS COUNTER 11
YES
STORE FINGER 0 POTENTIOMETER VALUE
IS COUNTER 13
YES
STORE FINGER 1 POTENTIOMETER VALUE
IS COUNTER 15
YES
STORE FINGER 2 POTENTIOMETER VALUE
IS COUNTER 17
YES
STORE FINGER 3 POTENTIOMETER VALUE
STORE FINGER 4 POTENTIOMETER VALUE

Fig. 4 – Glove Interface Software

Once the framework was in place to continuously retrieve data from the sensor glove in real time the feature recognition implementation could begin. The feature recognition algorithm begins by checking if the user desires to control the mobile agent in manual override mode, access the grasping device, or to have the mobile agent operate autonomously. The commands for each of these actions are given through the potentiometers located on the fingers and the accelerometer located on the back of the sensor glove. If the hand is closed in a fist, or pointed downward at the user's side then the algorithm executes the autonomous mode. The program detects the fist if the potentiometers on the fingers drop below a certain threshold and it detects that the arm is at the user's side if the x-axis value is ~1g and the y-axis value is ~-1g on the accelerometer. If the pinkie and ring finger are below a certain threshold then the program gives the user control of the grasping device, and the default mode is manual override. The feature recognition flowchart is shown in Figure FEATREFLOW on page P, and all of the feature recognition and dataglove code is located in Appendix CD.

MANUAL OVERRIDE MODE

In manual override mode, pitch and roll are calculated using the values from the x-, y-, and z-axis of the accelerometer. The equations for calculation are as follows:

Pitch = arctan(x-value/(sqrt(y-value^2 + z-value^2)) (1)
Roll = arctan(y-value/(sqrt(x-value^2 + z-value^2)) (2)

The pitch value controls the speed of the agent. If the pitch is oriented downward then the degree to which the user's hand is downward oriented determines the forward speed of the mobile agent. If the pitch is oriented upward, the agent stops. Similarly, the magnitude of the roll to the right or left determines the banking rate of the agent to the respective direction of the roll.

ROBOT ARM AND GRASPING DEVICE CONTROL MODE

In order to implement grasping functionality, a robotic arm was to be mounted on the front of the mobile agent. An image of the arm is shown in Figure 6 on page 11. The arm consisted of three joints with servomotors providing a vertical axis of rotation, one two-pronged grasping device controlled by a servomotor, and one servomotor controlling the horizontal angle of rotation of the entire arm.

The 645MG servomotor was used everywhere except the lowest joint with a vertical axis of rotation; an 805BB servomotor was used there because of its higher torque. Both types of servomotors were controlled with the same theoretical range of via Pulse Width Modulated (PWM) signal. By adjusting the length of a 20ms period (50Hz) that was "high" the servomotor would move to an angle that corresponded with the high portion of the period. Once there, it would utilize its own closed loop control circuitry to remain fixed at that position. Figure 5 illustrates this concept.

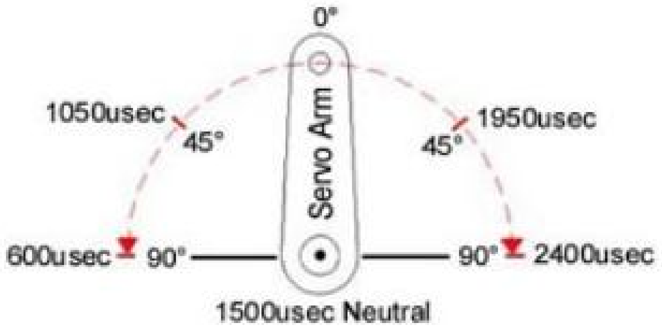


Fig. 5 – Range of Period of PWM for Servo Motors

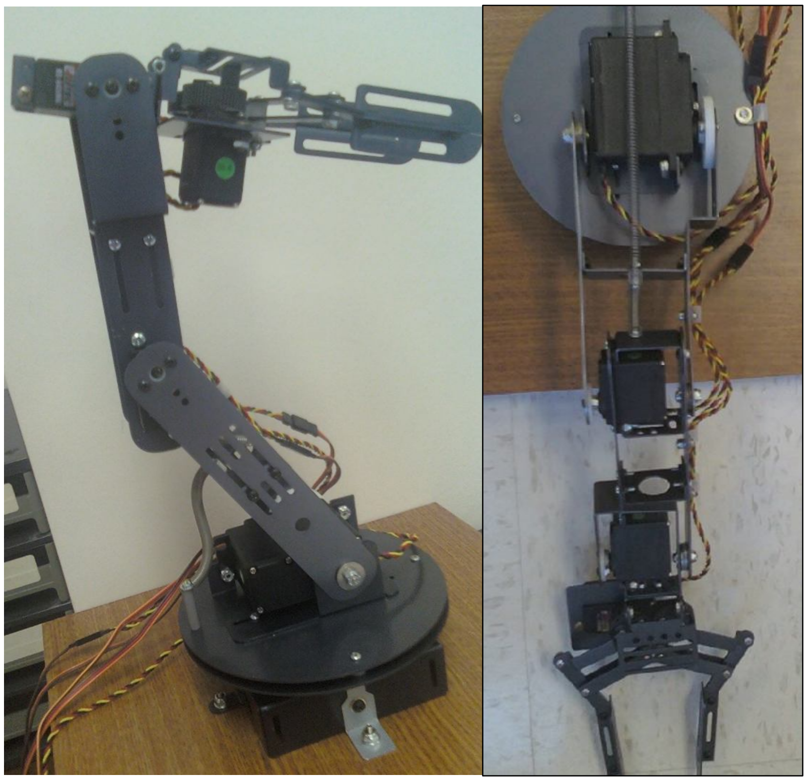
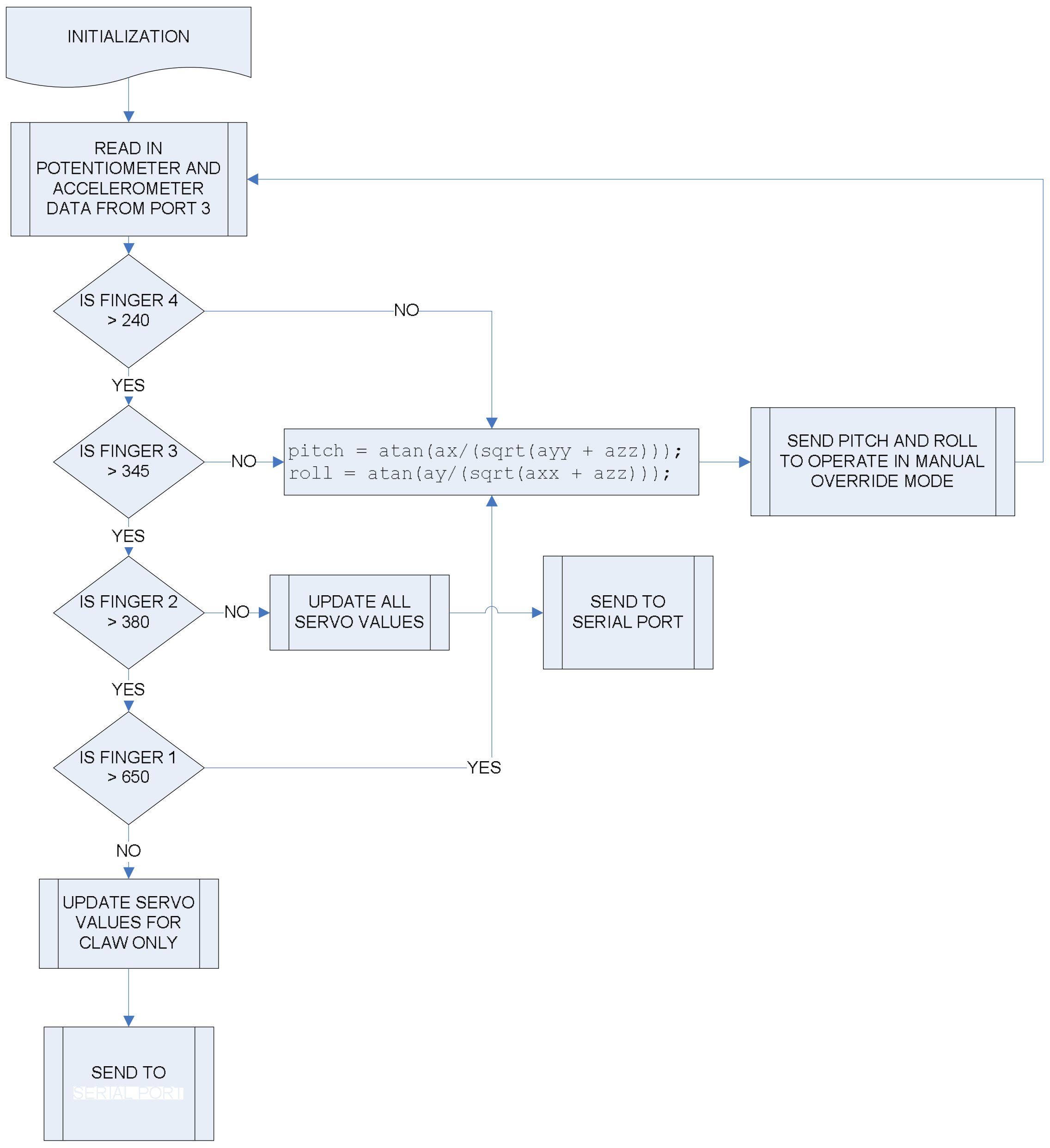


Fig. 6 – Robot Arm to be Mounted on Robot

The servomotors mounted on the arm were individually measured to determine their maximum and desired ranges. The ranges for each servomotor are shown in Figure 5. These ranges were then hardcoded into the robotic arm control algorithm. The robotic arm control algorithm takes the real time potentiometer and accelerometer readings from the dataglove and multiplies them by a previously calculated scaling factor to compute the appropriate position of each servomotor. The flowchart for the arm control algorithm is shown in Figure 7 and the code is located in Appendix B. The scaling factors were initially computed by taking a simple ratio of the accelerometer and potentiometer ranges and the servomotor ranges. The final scaling factors are a result of tuning. Once the appropriate position has been calculated, the number of interrupts in the 20ms PWM period to remain high is relayed to the microcontroller that sends the PWM signals to the servomotors in the robot arm via 8-bit UART protocol at 115200 bps. The interrupt time of the microcontroller was chosen to be 10 us because it provides good resolution.



```
pitch = atan(ax/(sqrt(ayy + azz)));
roll = atan(ay/(sqrt(axx + azz)));
```

Fig. 7 – Sensor Glove Control Algorithm for Robot Arm

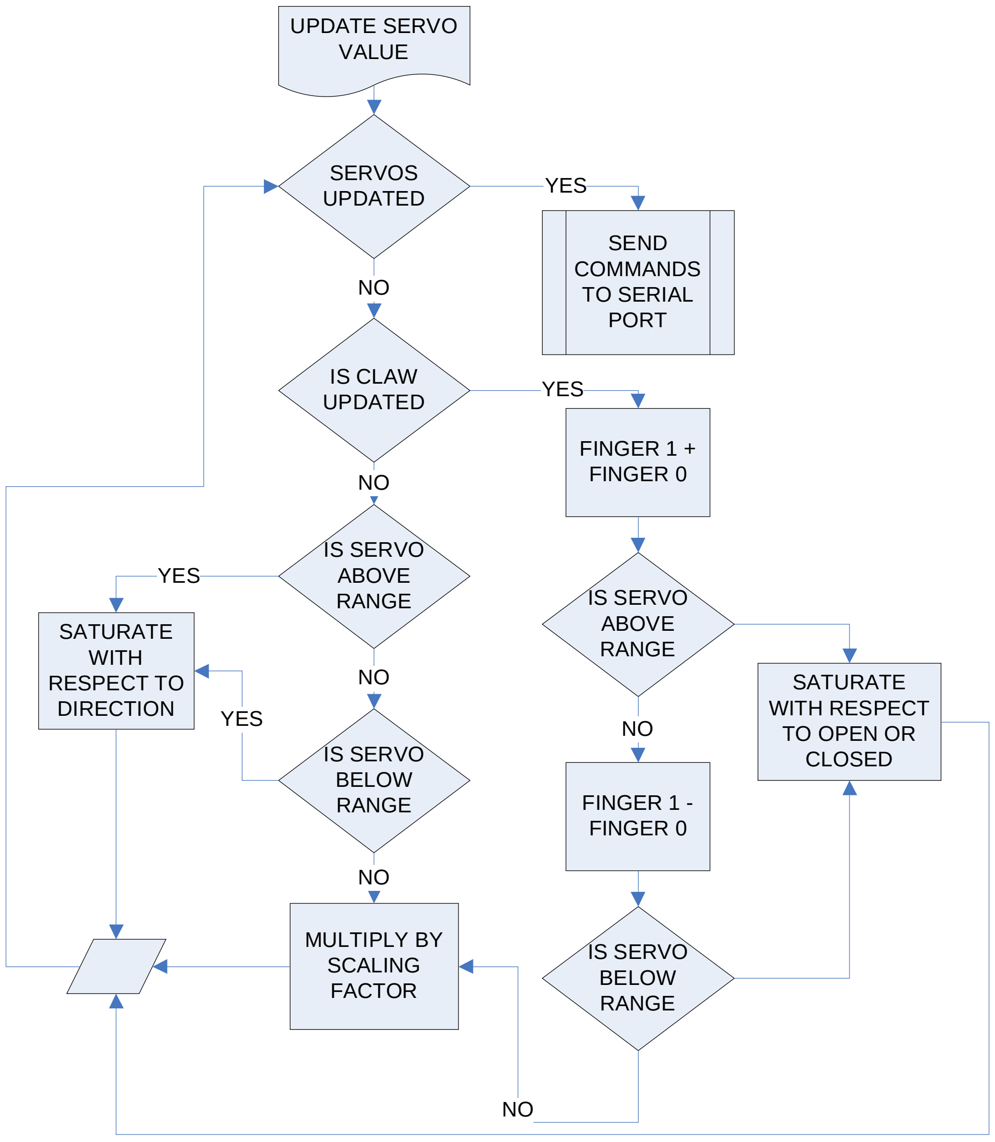


Fig. 8 – Servo Motor Updates

The robot arm receiver program is run by the microcontroller that sends the PWM signals to the servomotors mounted on the robot arm. The algorithm operates by continuously sending a PWM signal with a 20 ms period (50Hz) to each of the servomotors mounted on the robot arm. The amount of time that each of the servomotor PWM signals is "high" is determined by a counter unique to each servomotor. This counter is the number of 10 us interrupts that the microcontroller should send a "high" signal. After the count value has been reached the signal to that counter's particular servomotor is drawn low for the remainder of the 20 ms period. The updates for the count values are continually received from the serial port via UART protocol with 8-data bits at a rate of 115200 bps in the byte cycle illustrated in Figure 9. The flowchart for the robot arm receiver program is located in Figure 10 and the code is located in Appendix C.

BYTE CYCLE FOR SERVO CONTROL OF ROBOTIC ARM		
Order of Byte Cycle for Transmission	CLAW	UART PROTOCOL 8 data bits 115200 bps
	WRIST	
	ELBOW	
	SHOULDER	
	ROTATION	

Fig. 9 – Byte Cycle for Servo Control

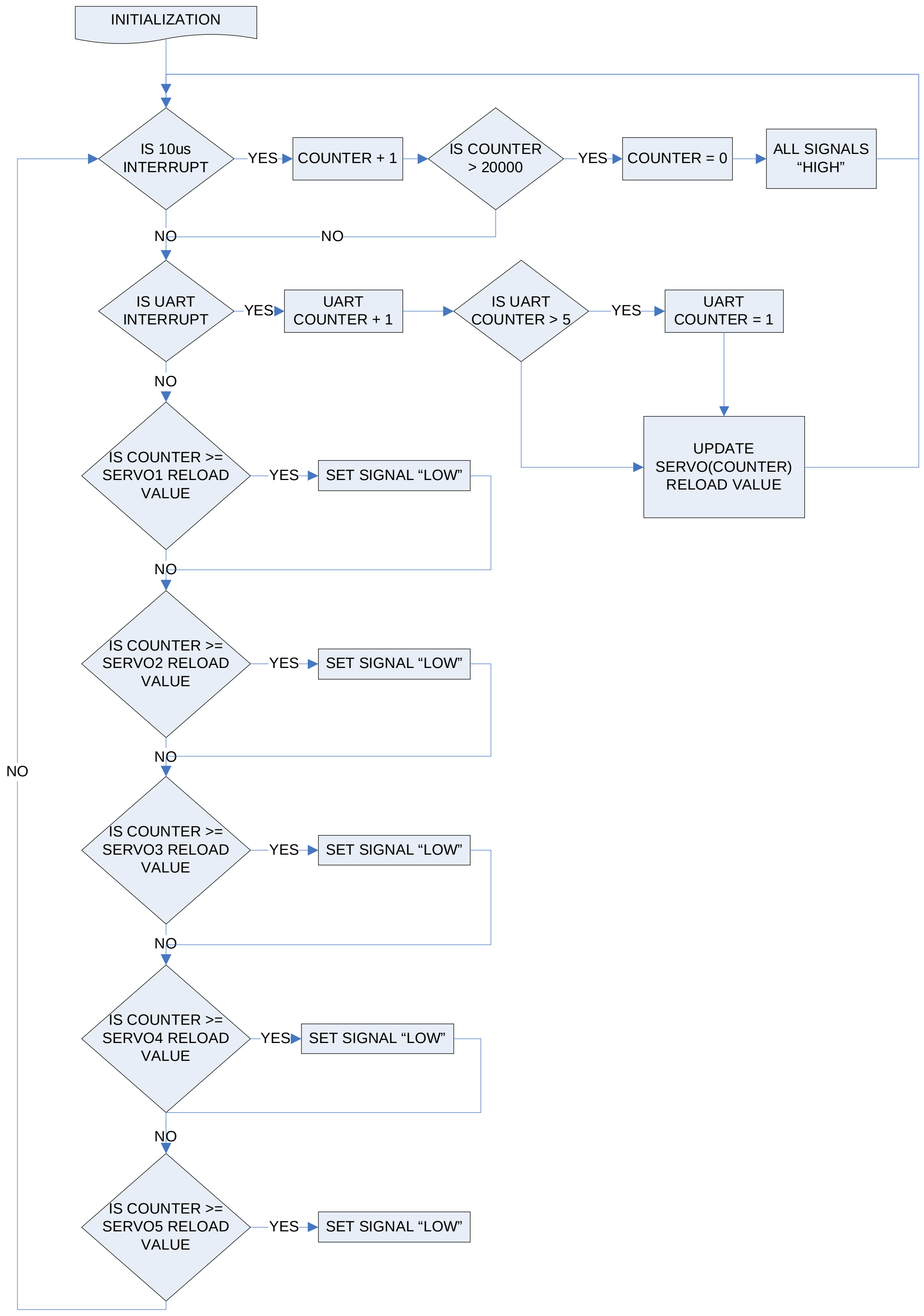


Fig. 10 – Control Receiver for Robot Arm

GYROSCOPIC DRIFT REDUCTION

In order to track movement of the LCD headpiece so as to provide visual feedback to the user, sensors should be mounted atop the display screen. In order to be low cost and flexible, the sensors should be MEMS devices. MEMS stands for micro-electro-mechanical, and MEMS devices are inexpensive and flexible. However, MEMS devices do have some inherent imperfections. Magnetometers have distortion around magnetic devices, accelerometers provide poor resolution for short movements, and gyroscopes have drift. In order to cancel these imperfections, the best solution would be a Kalman filter. Kalman filters update their coefficients in real time based on a statistical model in order to minimize the error of the system. However, a Kalman filter was too complex for the purposes of this project. Therefore, a solution was computed that solved a problem more narrow in scope than applying a Kalman filter to the problem of sensor fusion to a gyroscope, accelerometer, and digital compass. This solution was the design and computation of an FIR Wiener filter to cancel nonlinear gyroscopic drift.

A Wiener filter uses second order statistics in order to compute the coefficients of the filter. For this application, an FIR filter was designed because it is inherently stable and simpler to design than an IIR filter. The statistics required by the Wiener filter are the autocorrelation of the input and the cross-correlation between the input and the desired output. Signals simulating the drifty input and the desired output were generated in MATLAB and the auto and cross-correlations were taken. The respective equations for the auto and the cross-correlation are:

$$rxx(k) = E\{x(n+k)x(k)\} \tag{3}$$
$$rxy(k) = E\{x(n+k)y(n)\} \tag{4}$$

Next, the correlation statistics are arranged in a Toeplitz matrix:

$$\begin{bmatrix} rxx(0) & rxx(1) & \dots & rxx(M) \\ rxx(1) & rxx(0) & \dots & rxx(M) \\ . & . & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ rxx(n) & rxx(n-1) & \dots & rxx(0) \end{bmatrix} \begin{bmatrix} c0 \\ c1 \\ . \\ . \\ . \\ cM \end{bmatrix} = \begin{bmatrix} rxy(0) \\ rxy(1) \\ . \\ . \\ . \\ rxy(M) \end{bmatrix} \tag{5}$$

The Toeplitz matrix can be solved using the Levinson-Durbin algorithm for n^2 speed: a popular method in industry.

After that, the Wiener FIR filter is applied to the incoming data. From the output of the filter, the error can be computed. The equation is as follows:

$$e(n) = d(n) – y(n) \tag{6}$$

where e(n) is the error, d(n) is the desired output, and y(n) is the estimated output from the filter.

Then the prerequisites for solving for Minimum Mean Square Error are computed:

Py = E{e(n)^2} (7)

The Minimum Mean Squared Error (MMSE) may now be computed:

Po = Py – [REAL(d(n)) – i*IMAG(d(n))] * c(n) (8)

For this project an 11-tap Wiener filter was designed using the methods described above. First, a sinusoidal signal with nonlinear drift was simulated. The characteristic equation and graph from MATLAB of the drifty filter input signal are shown below.

```
x = (m.*m)+sin(t * (2 * pi * frequency));    (9)
```

where the characteristic code for m is

```
m((tl/div)+1:tl) = 1:tl-(tl/div);       (10)
```

where tl/div delays the start of the drift and tl is a length 0 to time duration by 1/samplerate steps.

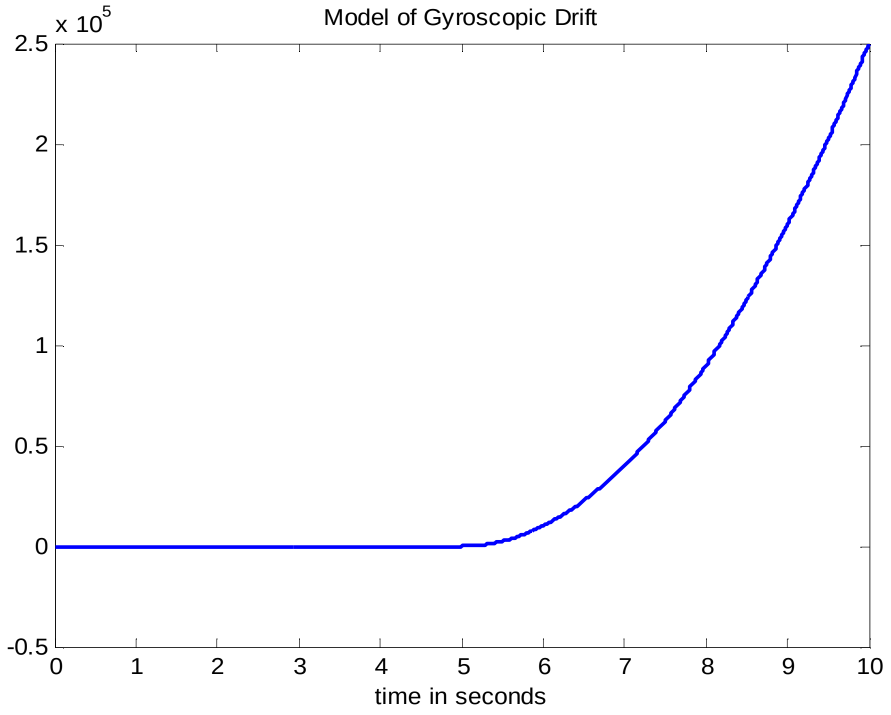


Fig. 11 – Gyroscopic Drift Model

Next, the auto and cross-correlation are computed using equations (3) and (4) and the "xcorr" function in MATLAB. The respective graphs are shown below:

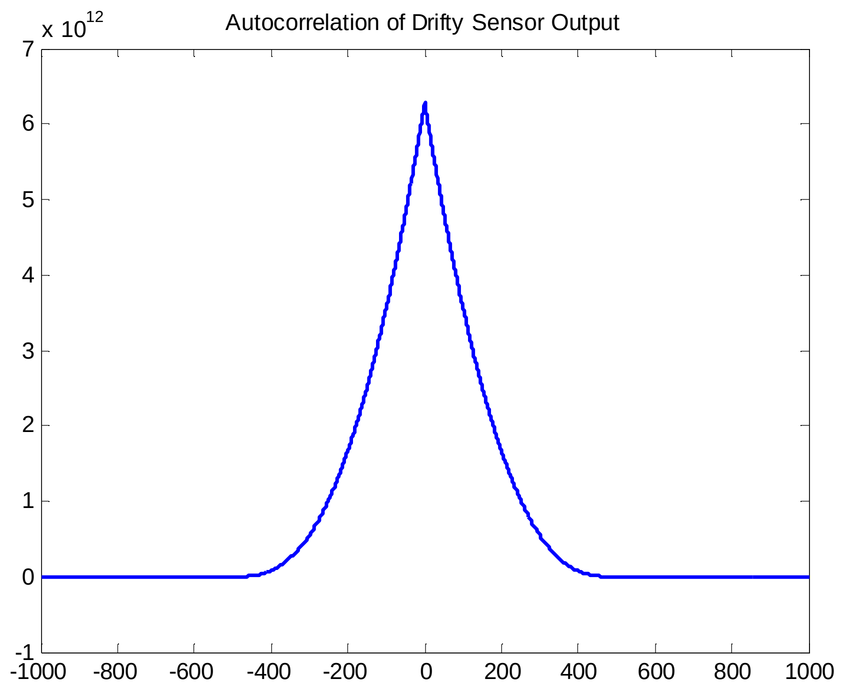


Fig. 12 – Drifty Sensor Autocorrelation

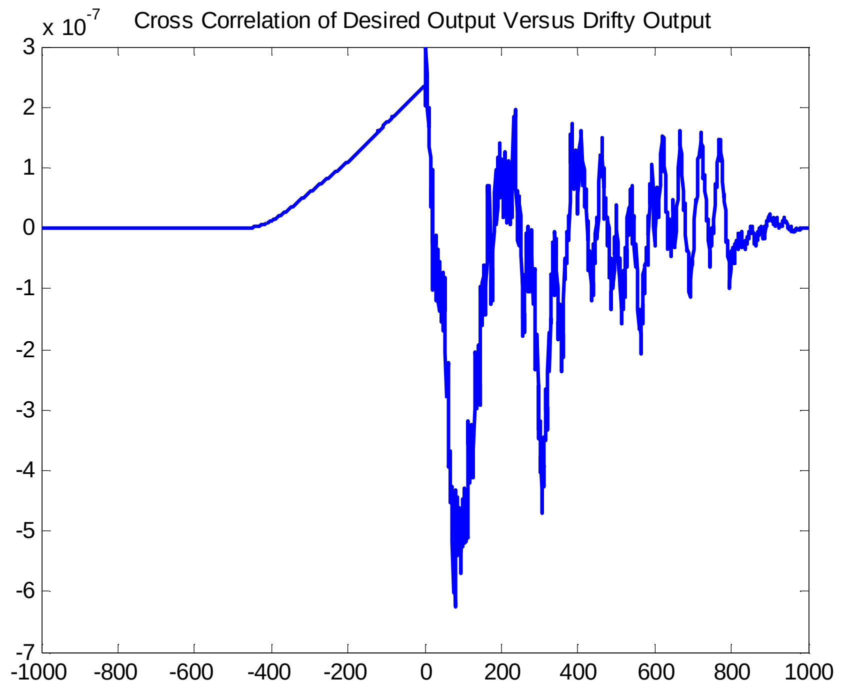


Fig. 13 – Cross Correlation of Desired Vs Drifty Outputs

It can be seen from the figure above that the drifty input and desired output have a cross-correlation of a magnitude 10^ -7.

Next, the "toeplitz" command was used in MATLAB to set up the Toeplitz matrix per equation (5). The cross-correlation matrix was also computed per equation (5), and the filter coefficients were transposed into a column matrix. Although the Levinson-Durbin could be used to solve for the coefficient matrix, simple matrix division was utilized in MATLAB for the purposes of this project. Once the coefficients were calculated, they were run through the FIR filter code shown below.

```
for k = 1:M                                    (11)
     y(n) = y(n) + c(k)*x(n-k+1);
end
```

where y(n) is the estimated output of the filter, c(n) are the coefficients, x(n) is the drifty sinusoidal input, and k is the number of taps in the filter. The output signal of the Wiener filter with reduced simulated drift is shown below.

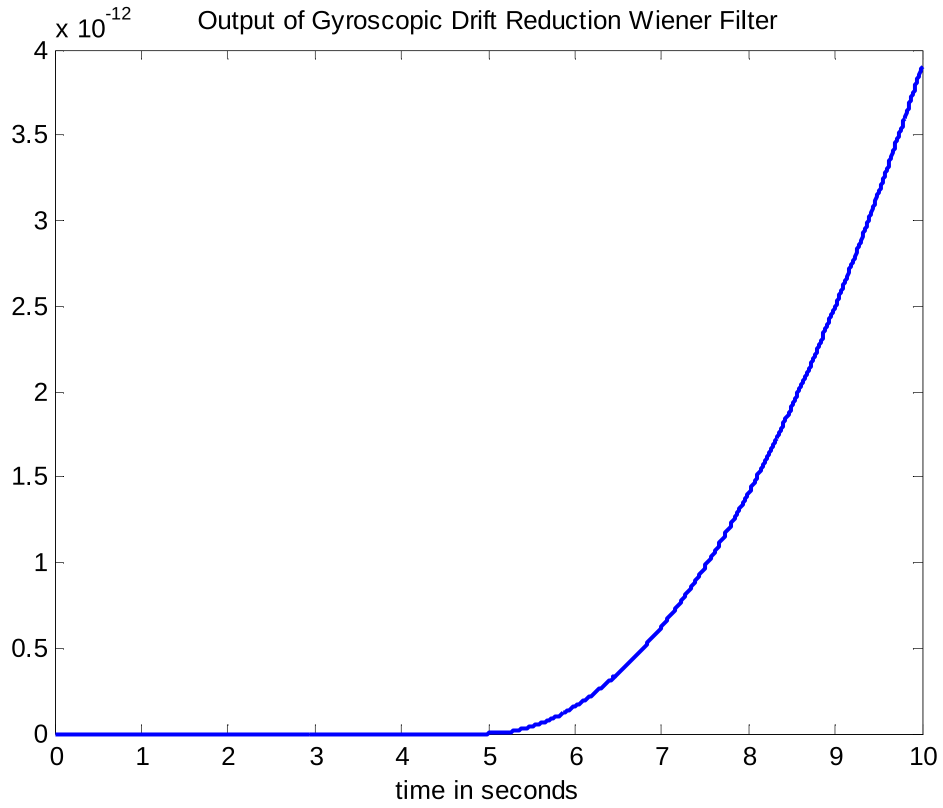


Fig. 14 – Gyroscopic Drift Reduction from Wiener Filter

From the Figure above it can be seen that the simulated nonlinear drift is now of the order of 10^ -12; a significant reduction.

Finally, equations (6), (7), and (8) were used to compute the Minimum Mean Squared Error (MMSE). The MMSE was calculated to be 8.1623 * 10^ -37 using the MATLAB code shown below.

```
     Po = Py - dot(conj(d),c);                 (12)
```

The Wiener filter derived above was of the order M = 11. The number of taps was chosen experimentally beginning with a 4-tap filter and gradually increasing the order until there were no more significant changes. As such, there is little difference between the 11th order filter designed and a 100-tap filter.

The Wiener filter itself was selected as a means to reduce gyroscopic drift because it can compute an optimal solution statistically for a non-time-varying input. Since the simulated gyroscopic input of the filter would be the same regardless of the time in which it was filtered, a Wiener filter could be implemented. If the drifty gyroscopic input was time-varying, then an adaptive filter that continuously computed optimal coefficients would be used. Though the Wiener filter was successful in reducing the simulated gyroscopic drift, a Kalman filter is still the best means to achieve sensor fusion to relay an accurate position of the LCD headpiece.

ROBOT CONTROL

The Observer program from the human mounted laptop tells which navigation mode the robot should be in, as well as a direction signal.

If the mode is manual (Joystick or Sensor Glove control), then the robot will move according to the input and wait for its next signal. If it is autonomous (Wandering, Erratic Movements, Potential Fields), then it does autonomous actions.

If there is an obstacle close by, the robot will try to avoid it and wait for a new mode. Else it will simply continue what it was doing and then wait for a new mode from the Observer.

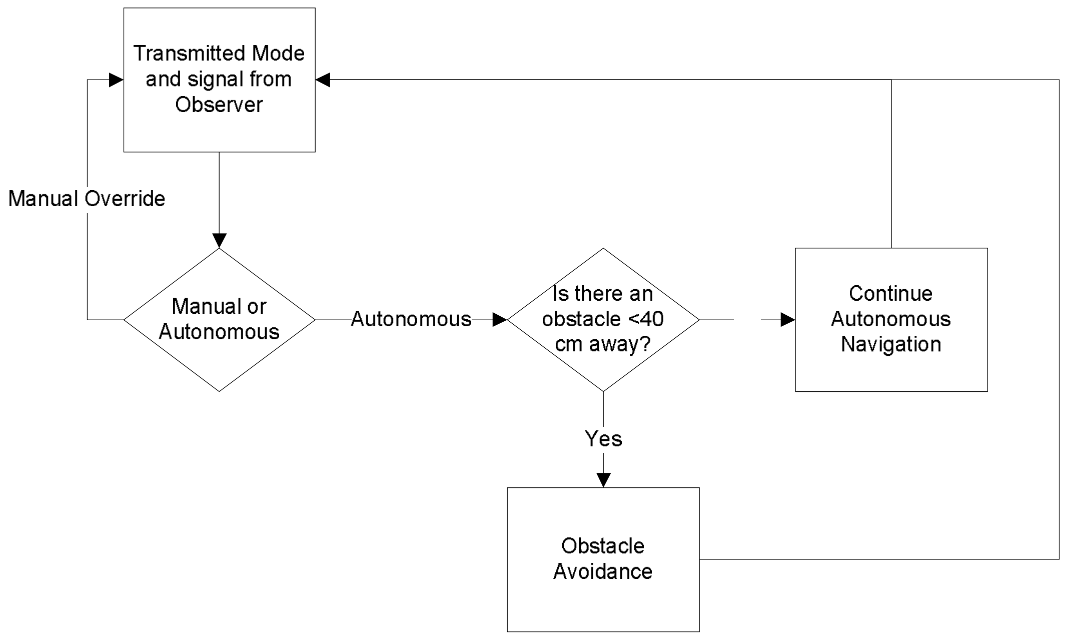

Transmitted Mode and signal from Observer
Manual Override
Manual or Autonomous
Autonomous
Is there an obstacle <40 cm away?
Continue Autonomous Navigation
Yes
Obstacle Avoidance

Fig. 15 – Software Flowchart for Robot Control

ROBOT OBSERVER PROGRAM

The program used to run the robot is built to allow multiple robots to be controlled. As such, there is an Observer program that oversees all the potential robots and compiles all the maps together.

The Initialize Robot function for the observer simply starts a clock, which is used in map creation and updating.

Start Communications then opens up the network to allow oversight of the robots.

Read Map simply looks at the current state of the map, and if there is no map, creates one.

Then the program goes into multithreading, with Run Database, Run Joystick, and Run Commander.

Run Database keeps track of the robot(s), in particular their positions.

Run Joystick allows for joystick manual control if a joystick is activated and the correct mode is selected.

Run Commander is for changing the various control modes of the robot. It looks at the input in the console window and determines how the robot should act from there.

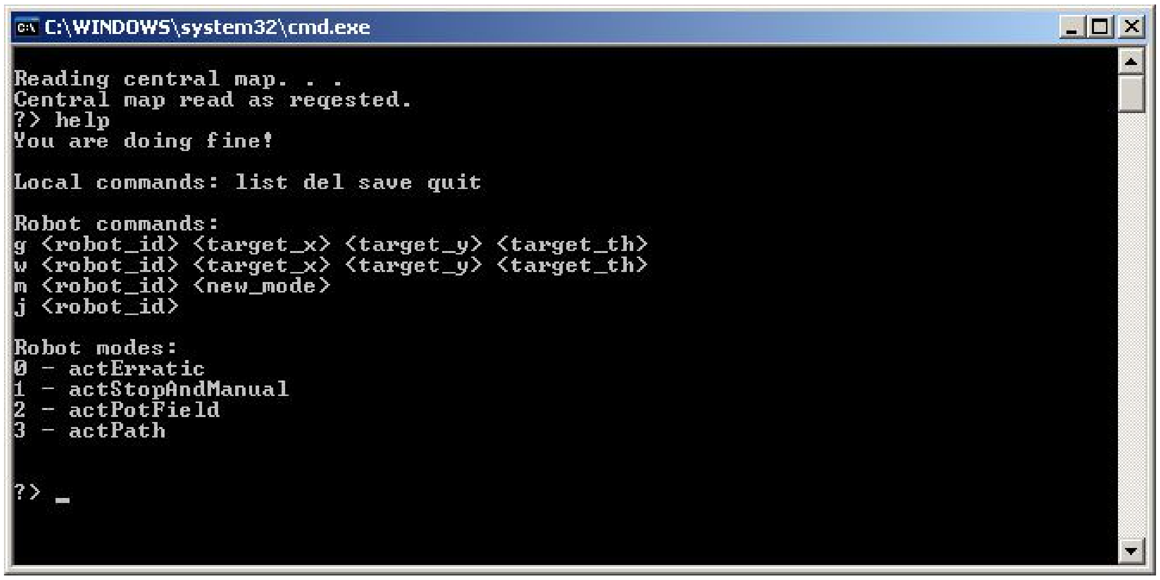
Get Map Data updates the map with data gathered from the robots at regular intervals.

Has Communication Stopped really only occurs at shut down of the program. If it has not stopped, the multiple threads simply continue. Else it proceeds on.

The Write Map section simply updates the map one last time before the program ends at Stop.

The robot observer had minimal modifications made to it throughout this project. The only noteworthy change was the expansion of the help function which was under the Run Commander thread. When 'help' was input in the console window, the various modes that the robot could take with a mode change were displayed.

Code in Appendeix E.



```
Reading central map. . .
Central map read as reqested.
?> help
You are doing fine!

Local commands: list del save quit

Robot commands:
g <robot_id> <target_x> <target_y> <target_th>
w <robot_id> <target_x> <target_y> <target_th>
m <robot_id> <new_mode>
j <robot_id>

Robot modes:
0 - actErratic
1 - actStopAndManual
2 - actPotField
3 - actPath


?> _
```

Fig. 16 – Observer Help Menu

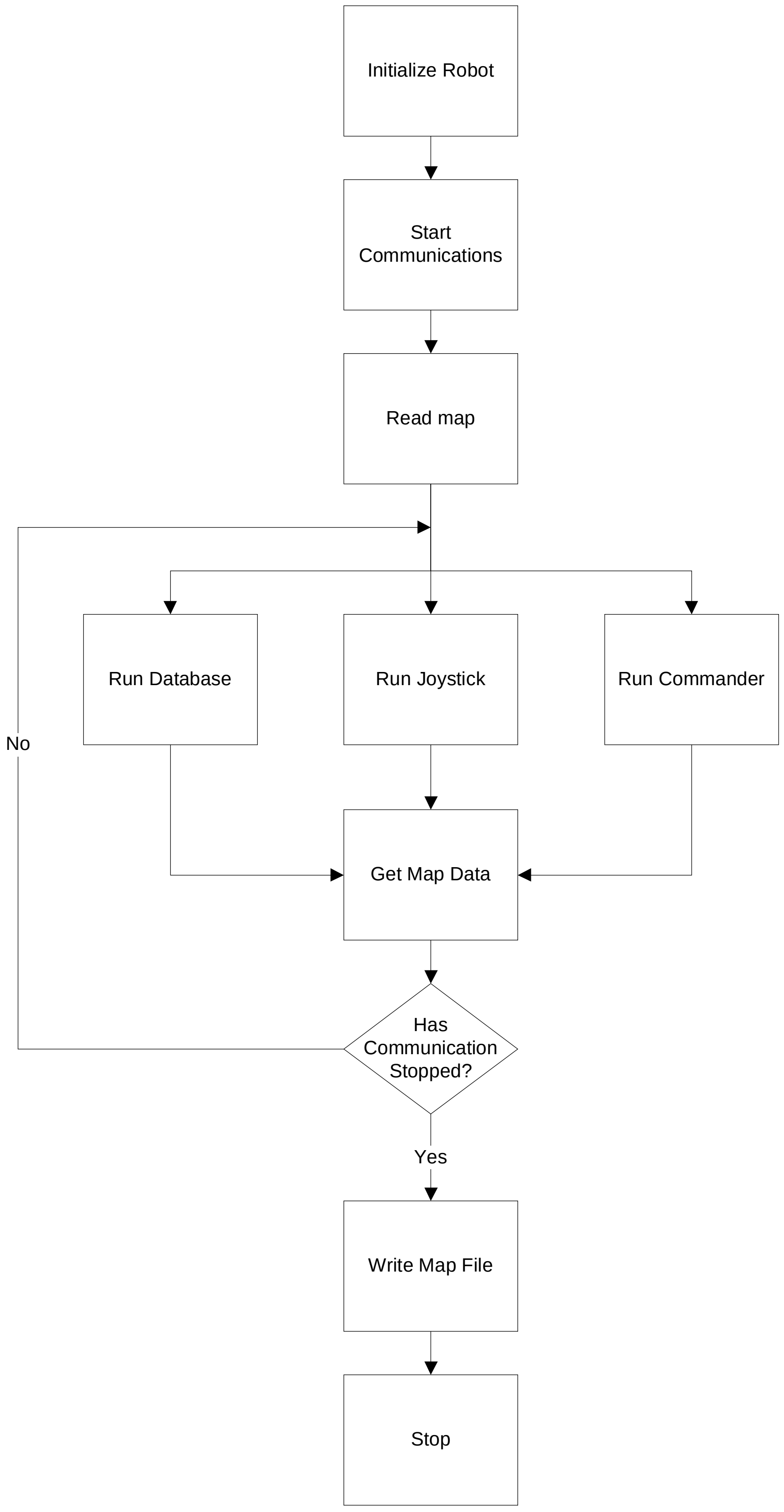


Fig. 17 – Robot Observer Flowchart

ROBOT FRAMEWORK PROGRAM

Initialize Robot first sets up time for map functionality. It then asks the user for a name for it to referred as. It then sets up that robot to be at a default position, in this case x = 0, y = 0, theta = 0. It then adds itself to the known robots list for the Observer.

Start Communications checks if there is a robot to connect to and if it is able connect to an Observer network.

Was Communication Successful checks if the robot connection and the connection to the Observer connection were successful. If they were, the program continues on to read the map, else it tries to reconnect.

Read Map checks the map. If there is no map, it creates a new one.

The program then proceeds to go into multithreading with Run Sensors, Run Database, Run Controller, and GL Output.

Run Sensors takes the data retrieved from the sensors and is capable of sending that data, as well as its position, back to the Observer.

Run Database takes its known sensor data and can compare it with Observer or other robots, to produce a more complete map.

Run Controller is the thread that actually controls robot movement. Depending on the mode sent by the observer, it will take one of four different actions. First, the robot might go forward until it encounters and obstacle. Then it will turn a random direction, and continue forward until it finds another wall, repeating itself. It can also wander erratically, where it will drive forward for a bit, rotate randomly, and then drive forward a bit more, repeating itself. Then there is potential field planning, in which it will take the lowest potential path to a point determined by the Observer. The last method is Manual Override, which is currently calibrated for Joystick control, though would be easily enough be modified for sensor glove input. Using values from the manual control, the robot will go in the direction and speed relative to max speed, the user requests. Run controller also can manually save its robot’s current map.

GL Output takes position data, and potentially sensor data, and sends it to the openGL program.

Still Communicating only happens when the threads have stopped, and it checks to see if the program is still communicating with the robot.

Save Map File saves the map. And then the program closes.

Code in Appendix E.

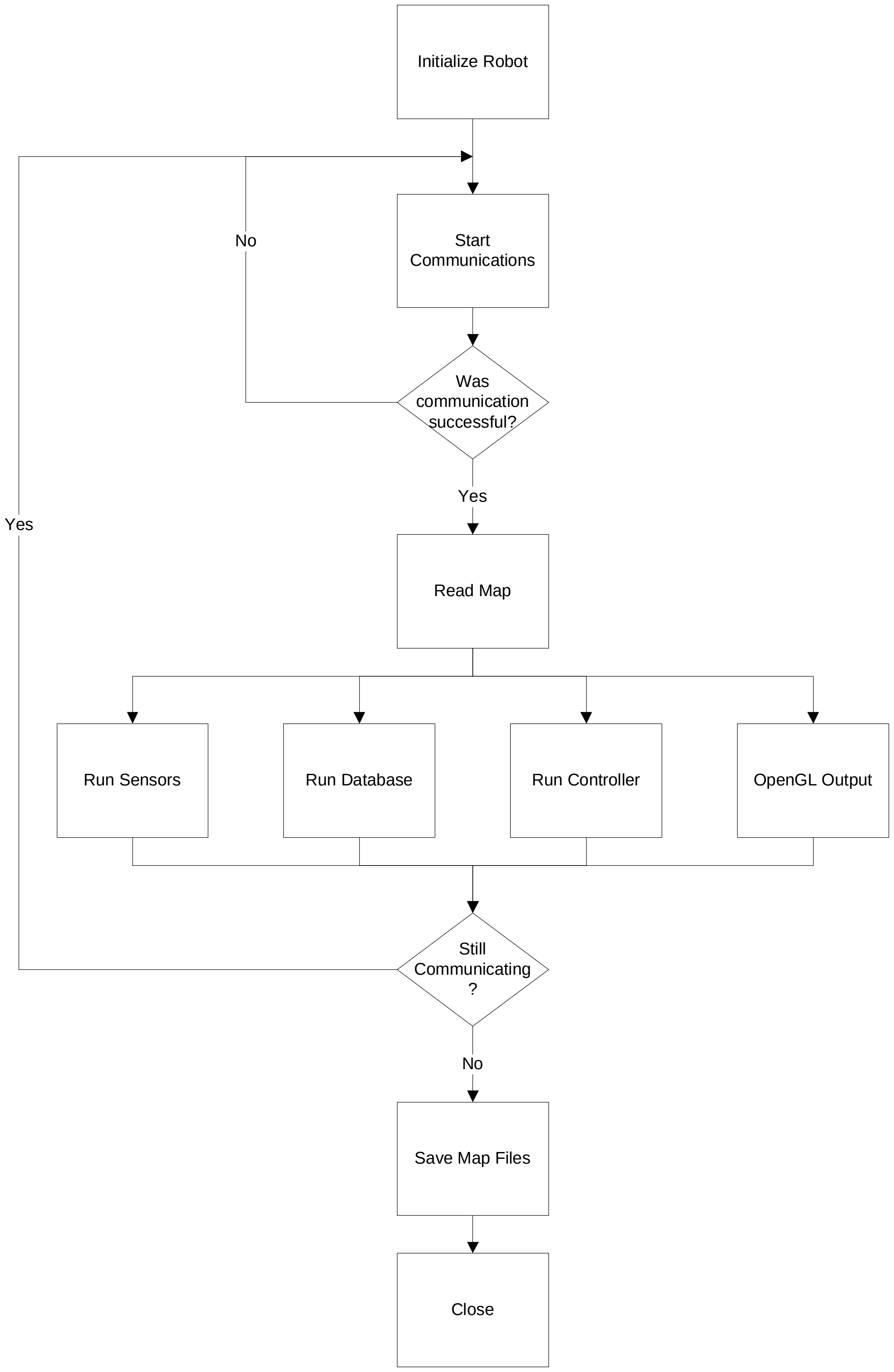


Fig. 18 – Robot Framework Flowchart

OPENGL PROGRAM

When OpenGL first starts up, it goes through some set up.

```
Int main_window;
glutInit(&argc, argv);
glutInitWindowPosition(1, 1);
glutInitWindowSize(800 ,600 );
glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
main_window = glutCreateWindow("Grid Explorer");
glutDisplayFunc(display);
glutReshapeFunc(reshape);
initialize();
```

Fig. 19 – OpenGL Start-Up

It first declares a main window. It then takes some values from the program being executed to help in Glut Initialization. Then the initial position of the window is set for the upper-left hand corner. The window size is 800 pixels wide by 600 high. The display mode is RGB, (red, blue, yellow) with double buffer and depth. The window is named “Grid Explorer”, then the window is displayed, and the world is ready to be shown. Initialize sets up variables that determine display properties and background color.

Update loop simply takes position data from the robot and transfer them to OpenGL.

```
if (NetComRecv(buffer, sizeof(buffer))>0 && buffer[0]=='P') {
      char c;
      char robotname[32];
      double inRX=0, inRY=0, inRT=0;
      std::istrstream ISTR(buffer);
      ISTR >> c >> robotname >> inRX >> inRY >> inRT;
      if (!ISTR.fail()) {
            z = inRX/10000;
            x = inRY/10000;
            horizontalAngle = inRT;
      }
}
```

Fig. 20 – OpenGL Update Code

A position buffer is taken from the robot and X, Y, and Theta values are taken from it. Based on how this OpenGL program was built, Robot(x) got mapped to z, and Robot(y) got mapped to x. The values are divided by 10,000 to accommodate the draw distance. If the data was not received, then OpenGL simply displays the last location.

Draw Walls simply takes two points, draws a line between them, and adds some height.

```
glBegin(GL_QUADS);
glColor3f(1,0.5f,1);
glVertex3f(-x1, -2.0f, z1);
glVertex3f(-x2, -2.0f, z2);
glVertex3f(-x2, -0.5f, z2);
glVertex3f(-x1, -0.5f, z1);

glEnd();
```

Fig. 21 – DrawWall Algorithm

GlBegin allows for quadrilaterals to be built from 4 points. From tuning it was discovered that it was best to have the bottom of the floor at height -0.5. For testing purposes, a height of -2 was used. The negatives are based off of development of the program with the x-axis being reversed. With the height taken care of, all that was needed was two points for a line. glColor3f takes RGB values and makes the walls that color. This particular combo gave pink. GlEnd tells OpenGL to stop drawing new shapes.

Draw Environment actually encompasses Draw Walls, and takes all the drawn walls and displays them on the LCD eyepiece.

Code in Appendix E.

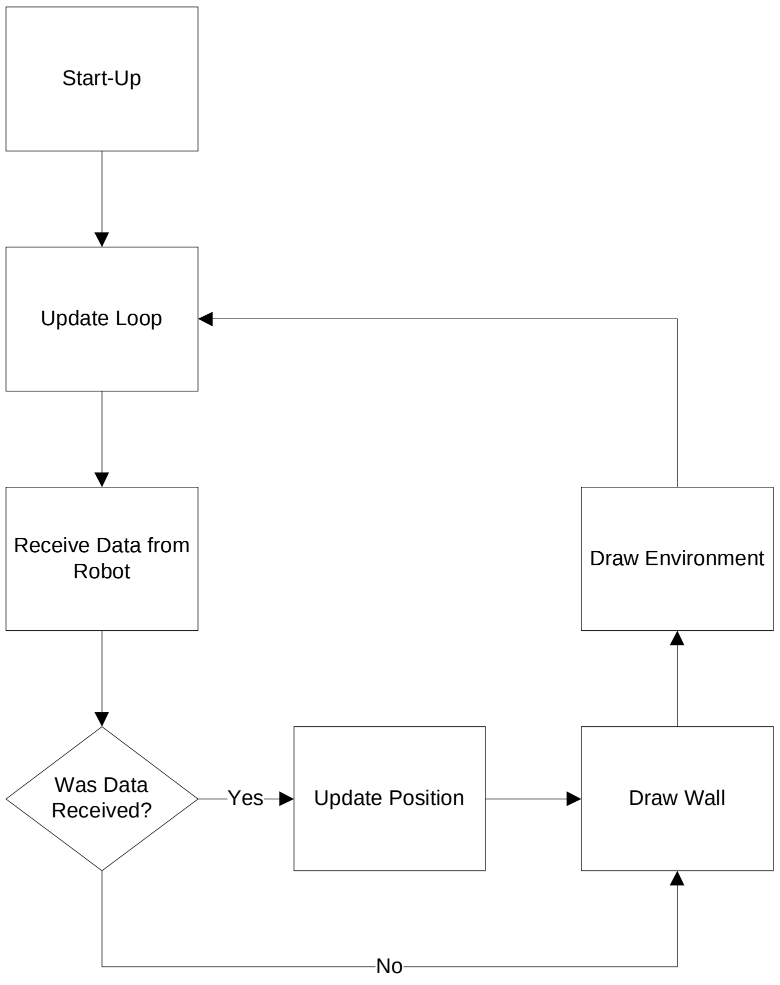


Fig. 22 – OpenGL Software Flowchart

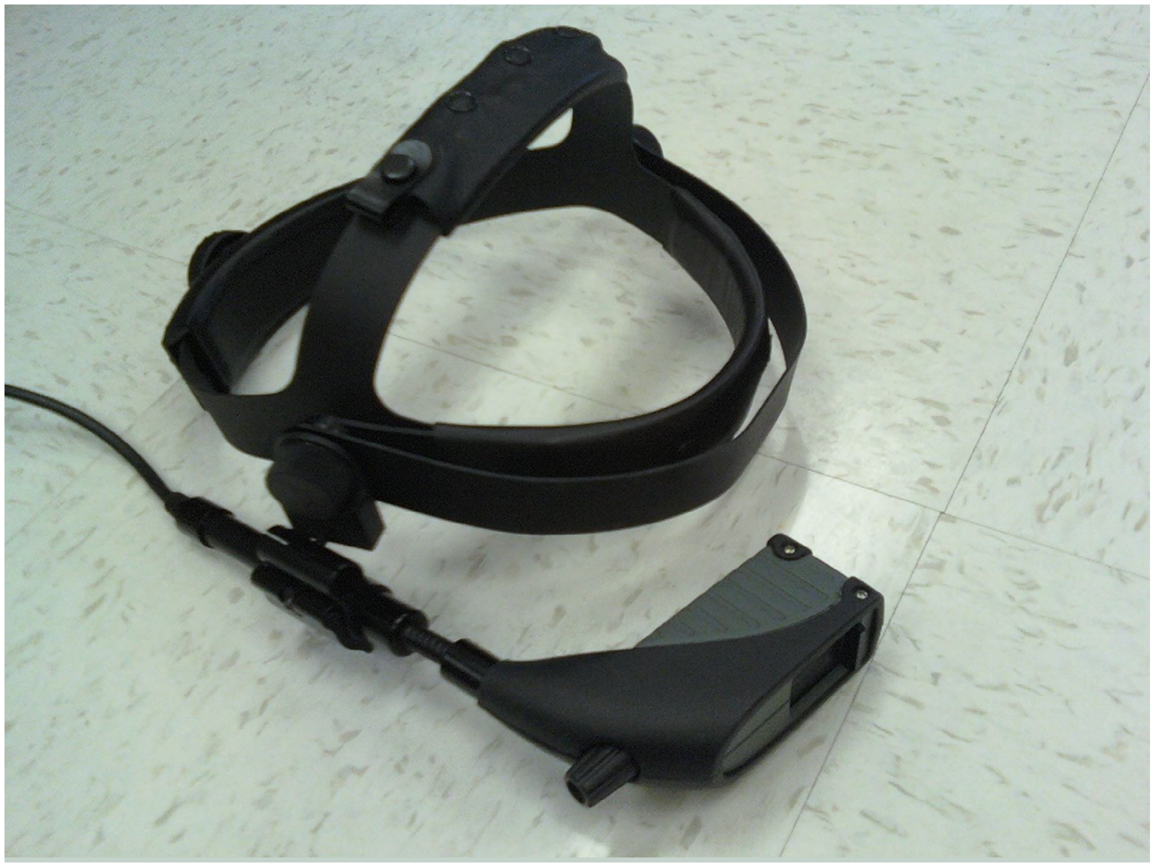
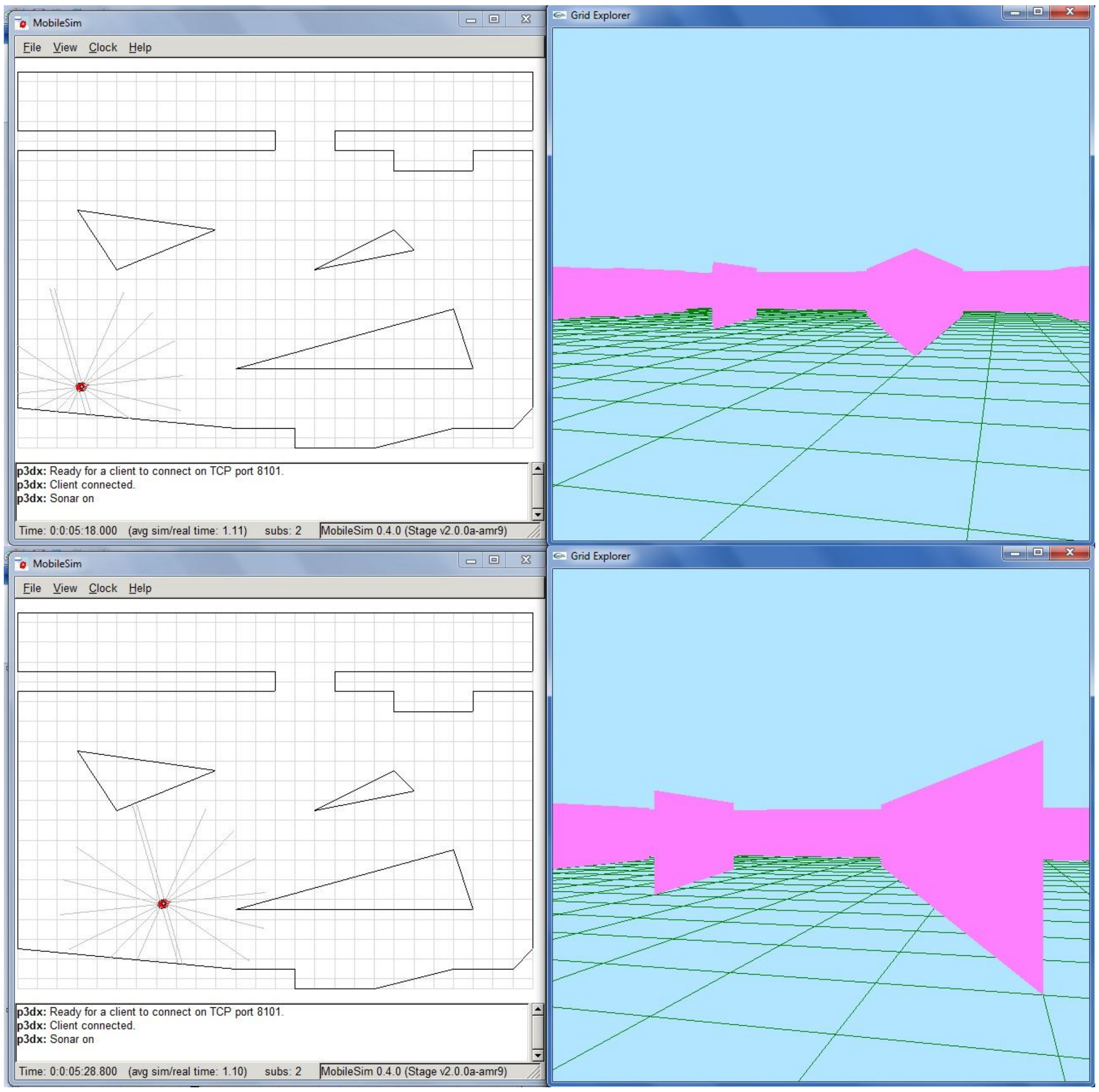


Fig. 23 – LCD EyePiece

DISCUSSION OF RESULTS

This project did have some significant results. First, the framework for the manual override mode is operational in the mobile agent's control algorithms and supports both joystick and sensor glove inputs. Also, there is communication to the LCD eyepiece worn by the user to provide visual feedback of the agent's environment. The visual feedback is updated in real time and does display an accurate and animated “vision” of the agent's sensors(Fig. 24).



p3dx: Ready for a client to connect on TCP port 8101.
p3dx: Client connected.
p3dx: Sonar on

```
Time: 0:0:05:18.000   (avg sim/real time: 1.11)   subs: 2   MobileSim 0.4.0 (Stage v2.0.0a-amr9)
```

p3dx: Ready for a client to connect on TCP port 8101.
p3dx: Client connected.
p3dx: Sonar on

```
Time: 0:0:05:28.800   (avg sim/real time: 1.10)   subs: 2   MobileSim 0.4.0 (Stage v2.0.0a-amr9)
```

Fig. 24 – Real Time Updating (delta = 10 seconds)

The sensor glove can interface into the human mounted computer, and the computer can continually read the status of the accelerometer and potentiometers from the dataglove. The data from the potentiometers is stable and feature recognition algorithms relying on the potentiometer data are reliable. However, the accelerometer information contains large spikes due to the user's imperfect gestures when providing motions for the feature recognition algorithm to recognize. The output from the accelerometer mounted on the dataglove will require significant filtering from a statistical filter before it can be made useful. The mode switching algorithm that relies upon a clenched or unclenched fist to change the mode of the agent has been written and can be implemented, as well as the parts of the algorithm that change the degree to which the claw attached to the robotic arm is open or closed. However, manipulation of the position of the robotic arm via motion measured by the accelerometer is impossible due to high noise content inherent to the accelerometer signal. The algorithm for manipulating the servomotors mounted on the robot arm is complete and can be implemented into the control programs to be loaded into the microcontroller located on the mobile agent. The specifications for reliable position of the LCD headpiece require the sensor fusion of an accelerometer, gyroscope, and digital compass. Because stabilization of these devices requires a control scheme too time consuming for the scope of this project, a problem with a narrower scope was addressed. Instead of designing a Kalman filter to provide accurate position results from the three sensors, a Wiener filter was developed to reduce gyroscopic drift. When tested in MATLAB with simulated nonlinear gyroscopic drift, the 11-tap filter was able to reduce the effect of the drift to a -12 order of magnitude. Finally, preliminary statistics have been gathered for the design of a filter to compensate for the problems of the dataglove mounted accelerometer that was previously mentioned.

CONCLUSIONS

A mobile agent is able to operate in either autonomous or manual override mode and the data from its sensors concerning its environment is animated in OpenGL and updated to the LCD eyepiece worn by the user in real time to provide visual feedback. Information can be read into a human mounted computer from a dataglove to provide reliable finger position information measured by potentiometers. Data from the accelerometer mounted on the sensor glove can be read into the human mounted computer, and preliminary statistics to design a filter to cancel the noise and motion spikes from the accelerometer have been gathered. An algorithm is complete to switch between the modes of the mobile agent based on potentiometer readings, and also to calculate the degree to which the robotic arm to be mounted on the mobile agent should be open or closed. The range of each of the servomotors mounted on the robotic arm have been measured and an algorithm has been written to implement the desired position control commands. A Wiener filter has been designed that can reduce a simulated gyroscopic sensor reading with drift to an order of magnitude of -12.

REFERECES

[1] Dimitris G. Manolakis, Vinay K. Ingle, Stephen M. Kogon. Statistical and adaptive signal processing: spectral estimation, signal modeling,vadaptive filtering.
[2] Nourbakhsh, Illah R., and Roland Siegwart. Introduction to Autonomous Mobile Robots (Intelligent Robotics and Autonomous Agents.) London: The MIT Press, 2004.
[3] Tipton, Scott, and Nick Halabi. Multi Robot Navigation and Mapping for Combat Environment: Functional Description and System Block Diagram. Bradley University, 2009