ABSTRACT

A SENSOR FAULT DETECTION SIMULATION TOOL

By Jason Smith

As the demand for fault detection of sensors increases in the field of autonomous mobile robots, a tool is needed to easily allow sensor fault detection algorithms to be compared and analyzed. The focus of this thesis is on the development of such a tool. More specifically, this work presents a demonstration of the tool by comparing two sensor fault detection algorithms: the interacting multiple model (IMM) estimator and the simple fault detector (SFD). The IMM is a well known algorithm and is highly regarded in literature. The SFD is a novel algorithm.

The simulation tool was written in C# and was used to simulate a four-wheeled robot with four navigation sensors. The user interface allows the user to select a predefined path type for the robot to traverse, specify its length, and cause any sensor to fail during a simulation run.

A SENSOR FAULT DETECTION SIMULATION TOOL

A Thesis

Submitted to the Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

Department of Computer Science and Systems Analysis

by

Jason Smith

Miami University

Oxford, Ohio

2007

Co-Advisor_____
Dr. Jade Morton

Co-Advisor_____
Dr. Eric Bachman

Reader_____
Dr. Scott Campbell

Reader_____
Dr. Qihou Zhou

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to thank the Dayton Area Graduate Studies Institute (DAGSI) and the Air Force Research Laboratory for funding this project. I would also like to thank my thesis advisor Dr. Jade Morton for her support and help throughout my graduate work at Miami. I have worked closely with Dr. Scott Campbell from Miami's Computer Science and Systems Analysis Department during my last couple years on campus and have learned a great deal from him. Finally, I want to thank Dr. Eric Bachmann and Dr. Qihou Zhou for agreeing to serve on my committee and for their inputs and suggestions.

1 Introduction

1.1 Motivation

This thesis was born out of a fascination for autonomous mobile robots and their ability to perceive an environment. In particular, the main idea explored in this work is the reliability of a given robot's perception at the lowest level: its sensors. The demand for fault detection and identification of sensors in the field of autonomous mobile robotics is growing (Hashimoto 1321). As these systems become more sophisticated and widespread, assuring system reliability and safety become increasingly more important.

The field of sensor fault detection has long been studied in application domains other than mobile robots. For instance, Yu et al presented a study of sensor fault detection for sensors used in chemical processes, Kolokotsa et al presented a study of sensor fault detection used in energy management systems, Visinsky explored fault detection in the domain of stationary robots, like those used in manufacturing facilities, and Zhang et al provided a comparison of fault detection algorithms for flight control systems.

This work focused on the development of a real time sensor fault detection simulation tool. The tool provides an intuitive graphical user interface for simulating an autonomous mobile robot, sensors, and fault detection algorithms. The interface plots the outputs from the fault detectors and allows a user to cause any simulated sensor to fail at any time by clicking an appropriate button.

Two fault detection algorithms were chosen to demonstrate the usage and effectiveness of the simulation tool. The first was the well-known interacting multiple model (IMM) fault detection algorithm. In literature, this is a very popular algorithm because it is not computationally intensive and performs better than other well-known competing algorithms in detecting sudden state changes (Zhang 1294). It is also very popular in the problem domain of automatic target tracking (Mazor 103).

The second algorithm implemented was a simple heuristic, which we called the simple fault detection (SFD) algorithm. The SFD algorithm uses a less-complicated set

of equations than the IMM algorithm and eliminates the use of a Kalman filter, which is an integral part of the IMM. The SFD and IMM algorithms will be analyzed in more detail in subsequent chapters.

The other simulated entities used in the demonstration were an autonomous robot and four navigation sensors. The simulated robot could move along one of three different path shapes: a line, a circle, or a square. The length of each path, the path shape, and several other parameters were user-specified. The simulation details and results are presented in the following chapters.

1.2 Overview

Chapter 2 lays the foundation for the rest of this thesis. It begins with a discussion of sensor fusion and transitions into an exploration of previous work. Next, background information concerning sensor fault detection is presented before delving into an analysis of the IMM algorithm. Finally, the chapter concludes with an examination of the SFD algorithm.

The specifics of the sensor fault detection simulation are the subject of Chapter 3. First, the objectives of the simulation are presented, followed by the fault model and robot model. The characteristics of the simulated robot and sensors are discussed next before an in-depth explanation of the simulation software architecture. The chapter ends with a useful discussion of simulation features and details about using the simulation program.

The results of this thesis are presented in Chapter 4, starting with an explanation of each simulation test scenario. Afterward, the results from each test scenario are analyzed.

Chapter 5 summarizes the simulation results and draws a few conclusions. Finally, future work and ways in which the simulation could be extended are discussed.

2 Background

2.1 Sensor Fusion

This work is primarily concerned with sensor fault detection. However, a rudimentary explanation of sensor fusion should be presented first since fault detection is part of the fusion process.

The concept of sensor fusion deals with the integration of sensory information from more than one sensor. For instance, an autonomous robot might have several sensors for determining its location, such as wheel encoders, inertial navigation sensors, an electronic compass, a vision system, and a GPS receiver. Relying on information from only one of these sensors is dangerous for a couple of reasons. First, each sensor provides an incomplete view of the environment and has an associated error range or confidence level in its measurements. Second, one or more of the sensors may malfunction or become unreliable. Thus, a method is needed to fuse the data from all the sensors in order to create a more robust system.

There are several methods and algorithms for fusing sensor data. Some of the more popular approaches are Bayesian networks, the Dempster-Shafer theory, and Kalman filters. This work is concerned with a Kalman filter-based approach.

The Kalman filter is "a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown" (Welch 1).

2.2 Previous Work

Several researchers have proposed many different sensor fusion architectures and algorithms, which include methods of detecting and handling sensor degradation or failure. Some researchers have used other various methods for sensor fusion, such as Riemannian manifolds (Cain 106), linearly constrained least squares (Zhou 118), and

vector space (Rao 130). Other researchers have created novel algorithms and architectures (Kundur 83; Williams 2).

The approach pursued in this work was a Kalman filter-based method. For example, Roumeliotis et al applied the multiple model adaptive estimation (MMAE) technique to detect and identify sensor failures in a mobile robot (1383). That technique used a bank of Kalman filters. The idea behind their approach was to do fault detection by processing the residual signature of the Kalman filter and fault identification by having a particular filter respond to its matching failure (Roumeliotis 1384). Hashimoto et al extended that work by applying the interacting multiple model (IMM) method to detect and identify sensor failures in a dead reckoning mobile robot (1321). This thesis extended the work presented Hashimoto et al by creating a novel fault detection method and comparing its performance to the IMM method.

2.3 Multiple Models (MM)

One of the most effective approaches for a problem like sensor failure detection is based on the use of multiple models (MMs) (Zhang 1293). The MM method "runs a bank of filters in parallel, each based on a model matching to a particular mode (i.e., structure or behavior pattern) of the system. The overall state estimate is calculated by the probabilistically weighted sum of the outputs of all filters" (Zhang 1293). For instance, a robot may be equipped with two navigation sensors. Two sensors providing navigation data results in four possible system states or modes, as shown below in Table 2.1.

Mode	Fault Sensor
0	No failure. All sensors functioning properly.
1	Sensor 1
2	Sensor 2
3	Sensor 1 and sensor 2

Table 2.1: System mode table

For each system mode, a corresponding filter calculates a model-based estimate in each iteration. The overall system mode is then determined by the probabilistically weighted sum of all the filters. Figure 2.1 shows a block diagram of the process for this example.

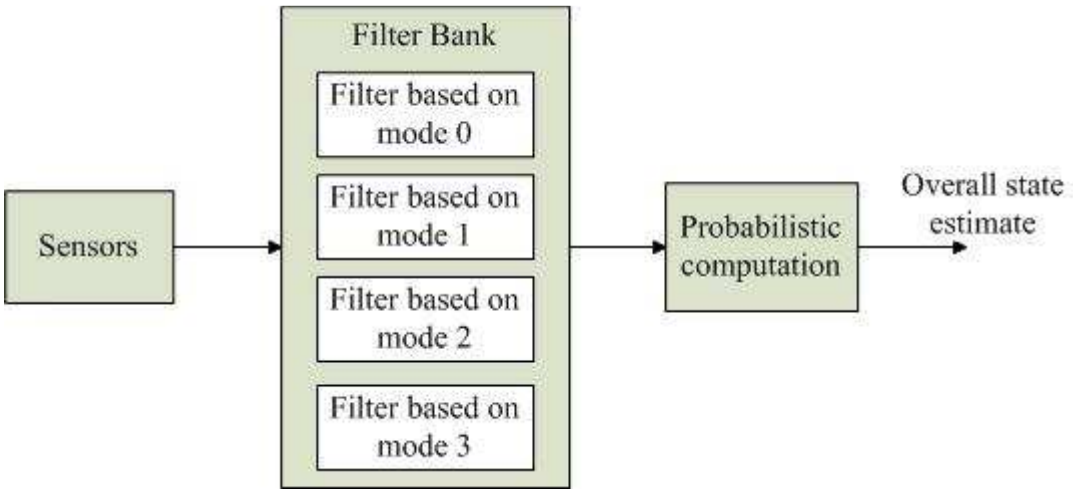

Figure 2.1: Multiple model block diagram

Several different MM algorithms for fault detection have been developed. Two of those appear often in sensor fault detection literature: multiple model adaptive estimation (MMAE) and the interacting multiple model (IMM) estimator.

The MMAE algorithm assumes that system modes do not jump, and the single-model-based filters are running in parallel without mutual interaction. Therefore, the MMAE algorithm does not function well under situations where the system modes experience sudden changes frequently, such as system failures (Hashimoto 1321). However, the IMM is not susceptible to that same limitation.

2.4 The Interacting Multiple Model (IMM) Algorithm

The IMM algorithm overcomes the weakness of the MMAE algorithm "by explicitly modeling the abrupt changes of the system by switching from one model to another in a probabilistic manner. Since structural changes (e.g., failures) of the system are explicitly considered and effectively handled, the IMM algorithm is much more promising for fault detection" (Zhang 1294).

The system mode transitions are defined in a probability transition matrix. A transition matrix for the robot with two sensors discussed in section 2.3 is shown below in Figure 2.2, where a = 0.997, b = 0.999, c = 1.0, and d = 0.001.

$$\begin{bmatrix} a & d & d & d \\ 0 & b & 0 & d \\ 0 & 0 & b & d \\ 0 & 0 & 0 & c \end{bmatrix}$$

Figure 2.2: Transition probability matrix

These numbers were used for illustrative purposes and were not calculated for this example.

The design of a transition probability matrix is dependent on sojourn time, which is the amount of time that has passed since a since a state changed. The diagonal entries in the matrix should be approximately equal to the mean sojourn time of each system mode as shown in Equation 2.1,

$$M_{jj} = max\left\{l_j, 1 - \frac{T}{\tau_j}\right\} \qquad \textbf{Equation 2.1}$$

where M_{jj} is the probability of transition from jth mode to itself, T is the sampling interval, τ_j is the expected sojourn time, and l_j is a designed lower limit for the jth mode transition probability (Zhang 1300). The matrix is also constrained as shown in Equation 2.2,

$$\sum_{j=0}^{x} M_{ij} = 1 \qquad \textbf{Equation 2.2}$$

where x is the total number of system modes.

The IMM also has the advantage of improved performance as compared to the MMAE for fault detection because the single-model-based filters interact with each other in a highly cost-effective fashion (Hashimoto 1321). Zhang et al describe the four major steps that occur in each cycle of the IMM (1297):

1. Model-conditional reinitialization (interacting or mixing of the estimates), in which the input to the filter matched to a certain mode is obtained by mixing the

estimates of all filters at the previous time under the assumption that this particular mode is in effect at the present time;

2. Model-conditional filtering, performed in parallel for each mode;
3. Mode probability update, based on the model-conditional likelihood functions;
4. Estimate combination, which yields the overall state estimate as the probabilistically weighted sum of the updated state estimates of all filters.

The implementation details of the IMM used in this work are presented in the following chapter. A block diagram of this algorithm is shown in Figure 2.3.

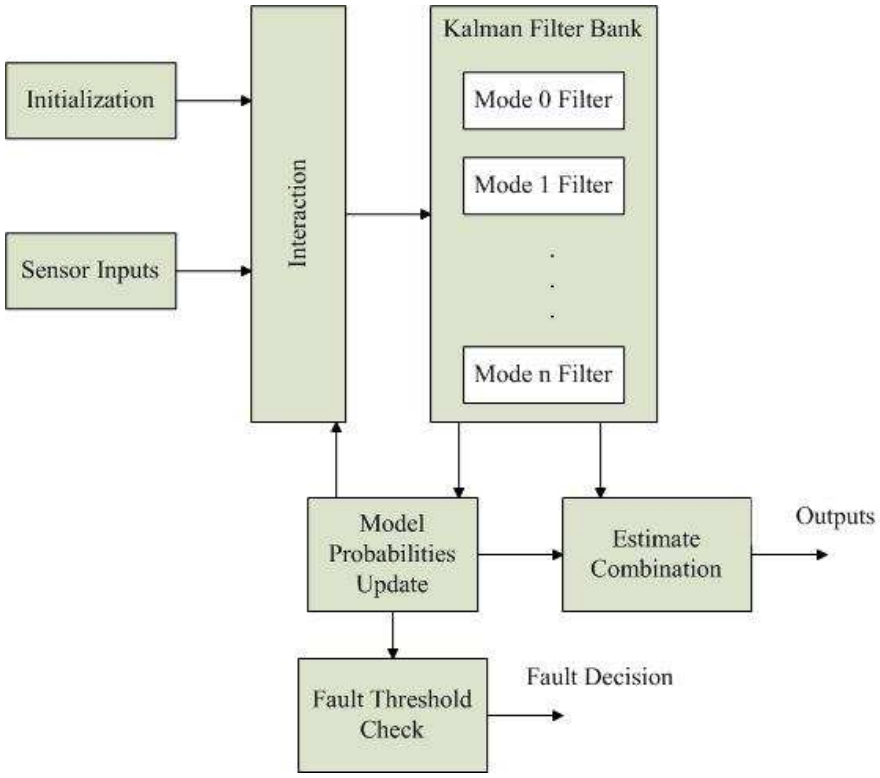

Figure 2.3: IMM block diagram

2.5 The Simple Fault Detection (SFD) Algorithm

The SFD algorithm uses the same explicitly defined system modes as the multiple model algorithms. In each cycle the SFD consists of three primary steps:

1. Prediction estimate, in which the predicted value and errors for each mode are calculated;

2. Mode probability update, which calculates the probabilities for each individual system mode;
3. Estimate combination, which generates the overall state estimate.

The implementation details of the SFD are presented in the following chapter. Figure 2.4 shows a block diagram of this algorithm.

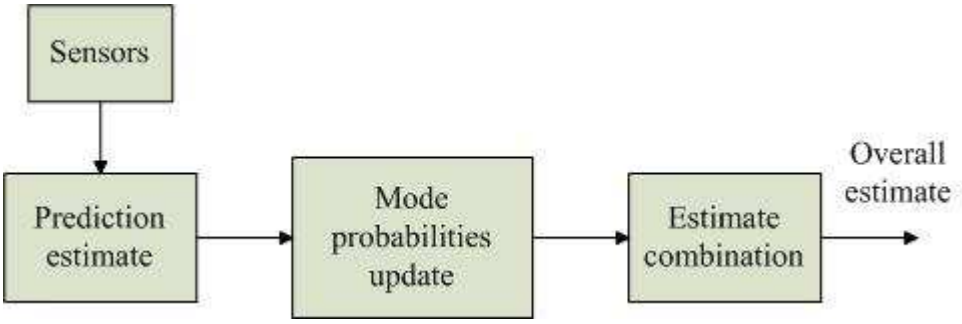

Figure 2.4: SFD block diagram

3 Simulation

3.1 Objective

The primary objective of this thesis is to compare the accuracy of the IMM against the accuracy of the SFD. The choice to create a simulation rather than conduct experiments on a real autonomous robot was made for several reasons.

First, a simulation allows us to focus solely on the fault detection and analysis. The simulation eliminates certain real-world variables, such as a faulty control system or a hardware related failure that could immobilize the robot. The simulation environment is completely controlled and bereft of unexpected failures that fall outside the scope of this work.

Second, simulations can be more easily repeated and repeated more frequently in a given period of time. Running a computer program is a quicker and easier way to generate test data.

Finally, while the author did have access to a mobile robot for a short period of time, it was beyond the scope of this thesis to create a mobile robot control system. Without the control system, the robot is not a viable test platform for any experiments involving autonomy.

3.2 Simulation Overview

The experiment in this thesis was realized by a simulation of a robot traversing three different paths of user-specified length: a straight line, a circle, and a square. The robot was equipped with four sensors: two wheel encoders, a digital compass, and a gyroscope. As the robot moved along a given path, the sensors generated values that were used by the IMM and SFD fault detectors to determine the velocity of the robot.

During each simulation run, the actual linear and angular velocities of the robot, along with the linear and angular velocities of each sensor and each fault detector, were plotted in a graph, like the one shown below in Figure 3.1. The content of the graph will be analyzed in the next chapter.

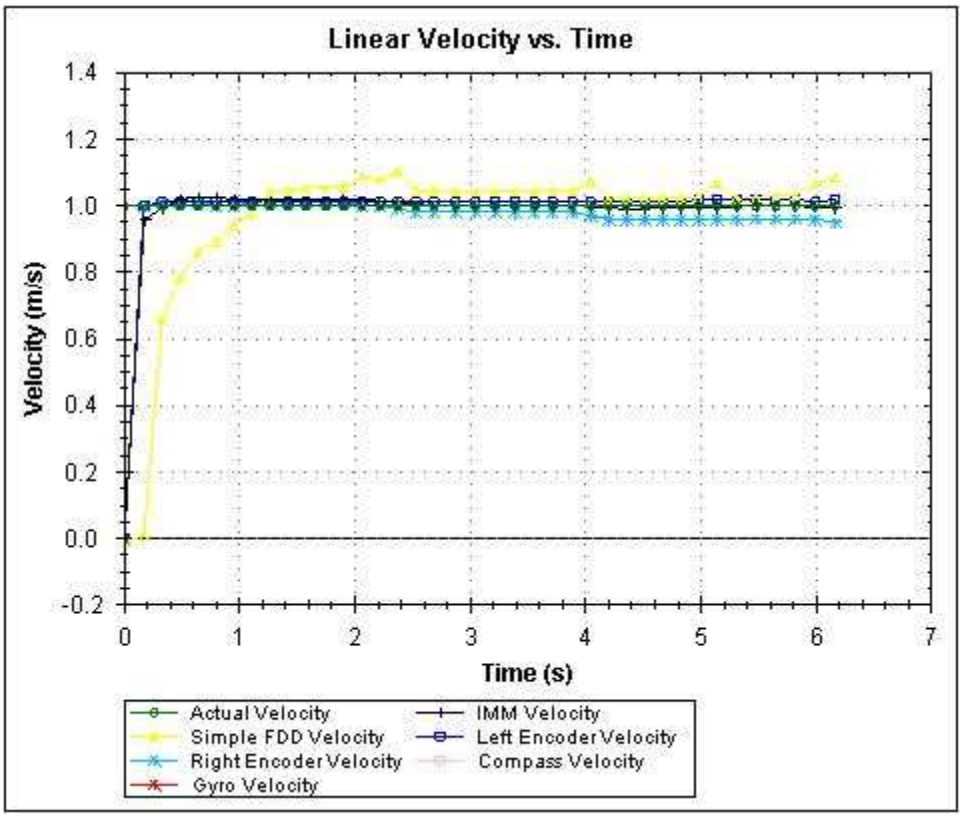

Figure 3.1: Simulation results example

Any of the sensors could be forced to fail by the end user during a simulation run. The failures were "hard" failures, in which the sensor could not recover and output zero. The specific sensor fault model used in this simulation is described in the following section.

3.3 Fault Model

The sensor fault model describes all the different types of hard failures that can occur and assigns a number to each. Since the robot was equipped with four sensors, it could have been in one of 16 possible states, or modes, at any given time. Table 3.1 lists each possible mode and the associated sensor fault(s).

Mode	Fault Sensor
0	No failure
1	Right wheel encoder
2	Left wheel encoder
3	Compass
4	Gyroscope
5	Right encoder, compass
6	Left encoder, compass
7	Right encoder, gyroscope
8	Left encoder, gyroscope
9	Compass, gyroscope
10	Right encoder, left encoder
11	Right encoder, compass, gyroscope
12	Left encoder, compass, gyroscope
13	Right encoder, left encoder, gyroscope
14	Right encoder, left encoder, compass
15	All sensors

Table 3.1: Possible system modes

3.4 Robot Model

The robot model was based on the velocity model used by Hashimoto et al, which assumed that the robot moved at a constant velocity, with a fault tolerant controller (1322). The velocity vector of the linear and angular velocities of the robot is shown below in Equation 3.1.

$$V = (v, \omega)^T \quad \textbf{Equation 3.1}$$

Equation 3.2 shows the rate kinematics of the robot, where t and t-1 are timestamps, and τ is the sampling period.

$$V(t) = V(t-1) + \tau\Delta V(t-1) \quad \textbf{Equation 3.2}$$

The sensor measurement vector is given in Equation 3.3, where z_L and z_R denote the velocities of the two drive wheels, calculated from the two wheel encoders, z_C denotes the compass output, and z_G denotes the gyroscope output.

$$z = (z_R, z_L, z_C, z_G) \quad \textbf{Equation 3.3}$$

The measurement model is shown in Equation 3.4,

$$z(t) = h_i\big(V(t)\big) + \Delta z(t) \quad \textbf{Equation 3.4}$$

where Δz is the noise vector with covariance matrix R, and h_i is the nonlinear kinematic function related to system mode i (Table 3.1). The nonlinear kinematic function can be calculated for each mode as shown below in Equation 3.5, where d is half the width of the robot.

$$h_0 = \begin{bmatrix} v + d\omega \\ v - d\omega \\ \omega \\ \omega \end{bmatrix}, h_1 = \begin{bmatrix} 0 \\ v - d\omega \\ \omega \\ \omega \end{bmatrix}, \ldots h_{15} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \textbf{Equation 3.5}$$

As described in section 3.3, when a sensor fails, it outputs zero. Thus, for each failure mode, the corresponding row of the kinematic function is zeroed out.

3.5 IMM Equations

Since this thesis extended the work of Hashimoto et al, the IMM equations are quite similar. This section presents each of the equations involved in each of the four steps of the IMM algorithm that were discussed in section 2.4.

3.5.1 Step 1: Model-Conditional Reinitialization

Each system mode's associated probability (μ_i), velocity (V_i), and velocity covariance (P_i) were interacted with each other and reinitialized as shown in the equations below,

$$\mu_i(t/t-1) = \sum_{j=0}^{15} M_{ji}\mu_j(t-1) \quad \textbf{Equation 3.6}$$

$$V_i(t/t-1) = \sum_{j=0}^{15} c_{ij} V_j(t-1) \quad \textbf{Equation 3.7}$$

$$P_i(t/t-1) = \sum_{j=0}^{15} c_{ij}[P_j(t-1) + \left(V_i(t/t-1) - V_j(t-1)\right)\left(V_i(t/t-1) - V_j(t-1)\right)^T] \quad \textbf{Equation 3.8}$$

Where M_{ji} is the transition probability matrix and c_{ij} used in Equation 3.7 and Equation 3.8 is calculated as shown below.

$$c_{ij} = M_{ji}\mu_j(t-1)/\mu_i(t/t-1) \quad \textbf{Equation 3.9}$$

3.5.2 Step 2: Model-Conditional Filtering

The bank of Kalman filters was used to calculate the state estimate of each of system mode and its associated covariance,

Prediction:
$$\left.\begin{aligned} V_i(t/t-1) &= V_i(t/t-1) \\ P_i(t/t-1) &= P_i(t/t-1) + \tau^2 Q(t-1) \end{aligned}\right\} \quad \textbf{Equation 3.10}$$

Update:
$$\left.\begin{aligned} V_i(t) &= V_i(t/t-1) + K_i(t)\tilde{z}_i(t/t-1) \\ P_i(t) &= P_i(t/t-1) - K_i(t)H_i(t)P_i(t/t-1) \\ K_i(t) &= P_i(t/t-1)H_i^T(t)S_i^{-1}(t/t-1) \end{aligned}\right\} \quad \textbf{Equation 3.11}$$

where H_i is a system parameter, $\tilde{z}_i$ is the measurement residual, and S_i is the associated covariance for each mode i. Each of these variables is calculated as follows.

$$H_i = \frac{\partial h_i}{\partial V} \quad \textbf{Equation 3.12}$$

$$\tilde{z}_i(t/t-1) = z(t) - h_i[V(t/t-1)] \quad \textbf{Equation 3.13}$$

$$S_i(t/t-1) = H_i(t)P_i(t/t-1)H_i^T(t) + R(t) \quad \textbf{Equation 3.14}$$

3.5.3 Step 3: Mode Probability Update

The probability for each mode was calculated using Equation 3.15,

$$\mu_i(t) = \frac{\mu_i(t/t-1)L_i(t)}{\sum_{j=0}^{15}\mu_j(t/t-1)L_j(t)} \qquad \textbf{Equation 3.15}$$

where L_i is the likelihood function for mode i. L_i is calculated as shown below.

$$L_i(t) = |2\pi S_i(t/t-1)|^{-1/2} \times exp\left[-\frac{1}{2}\tilde{z}_i^T(t/t-1)S_i^T(t/t-1)\tilde{z}_i(t/t-1)\right] \qquad \textbf{Equation 3.16}$$

3.5.4 Step 4: Estimate Combination

The estimate of the robot's velocity is the last calculation in the IMM cycle and is shown below in Equation 3.17.

$$V(t) = \sum_{j=0}^{15}\mu_j(t)V_j(t) \qquad \textbf{Equation 3.17}$$

3.5.5 Transition Probability Matrix

The transition probability matrix used in this simulation was the same matrix used by Hashimoto et al (1324). Figure 3.2 shows the matrix,

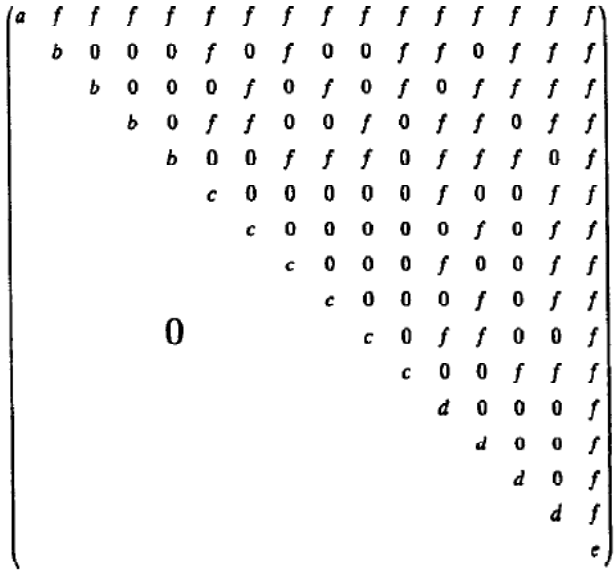

Figure 3.2: Transition probability matrix

where a = 0.985, b = 0.993, c = 0.997, d = 0.999, e = 1.0, and f = 0.001.

3.6 SFD Equations

This section presents each of the equations involved in each of the four steps of the IMM algorithm that were discussed in section 2.5.

3.6.1 Step 1: Prediction Estimate

The measured velocity vector (V_m) contains the average of the linear sensor velocities (v_{avg}) and average of the angular sensor velocities (ω_{avg}), as shown below in Equation 3.18.

$$V_m(t) = \begin{bmatrix} v_{avg} \\ \omega_{avg} \end{bmatrix} \quad \textbf{Equation 3.18}$$

The predicted velocity is calculated by adding the measured velocity of the previous cycle to the product of the predicted velocity of the previous cycle and the error vector (ε_i) of the previous cycle for each mode i, as shown below in Equation 3.19.

$$V_{p_i}(t) = V_m(t-1) + V_{p_i}(t-1) * \varepsilon_i(t-1) \quad \textbf{Equation 3.19}$$

The error vector and the total error (ε_{Ti}) for each mode is calculated using the following two equations.

$$\varepsilon_i(t) = \left| V_m(t) - V_{p_i}(t-1) \right| \quad \textbf{Equation 3.20}$$

$$\varepsilon_{T_i}(t) = \sqrt{(\varepsilon_i(t)[v_{avg}])^2 + (\varepsilon_i(t)[\omega_{avg}])^2} \quad \textbf{Equation 3.21}$$

3.6.2 Step 2: Mode Probability Update

Each mode probability (P_i) is calculated by dividing the inverse of the total error for that mode by the sum of the inverses as shown in Equation 3.22.

$$P_i(t) = \frac{\frac{1}{\varepsilon_{T_i}(t)}}{\sum_{j=0}^{15} \frac{1}{\varepsilon_{T_j}(t)}} \quad \textbf{Equation 3.22}$$

3.6.3 Step 3: Estimate Combination

The overall SFD velocity estimate equation is very similar to the overall IMM velocity estimate and is shown below in Equation 3.23

$$V(t) = \sum_{i=0}^{15} V_{p_i}(t)\, P_i(t)$$ **Equation 3.23**

3.7 Robot and Sensor Characteristics

The simulation was composed of several major components that were briefly mentioned in the previous section. The following subsections present the characteristics of each major component.

3.7.1 Robot Characteristics

The robot had four wheels and a square base one meter wide by one meter long as shown below in Figure 3.3. The two front wheels were motorized and provided locomotion and differential steering.

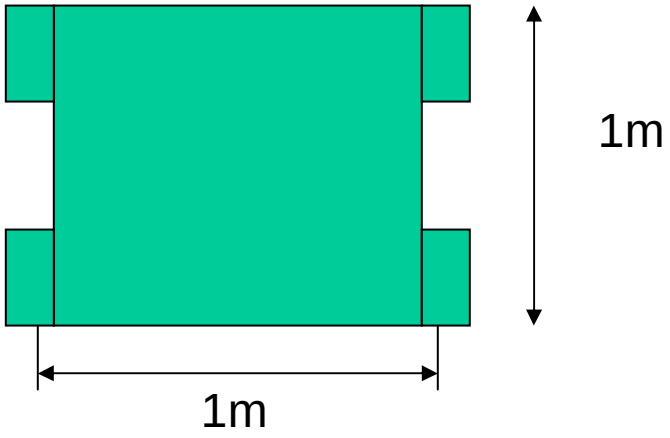

Figure 3.3: Robot dimensions

The robot traveled with a target linear velocity, v, of 1 meter / second on each path. On a circular path, the target angular velocity, ω, was calculated from v. First, the circumference of the circular path was calculated in meters as shown in Equation 3.24, where r is the user specified radius.

$$c = 2\pi r$$ **Equation 3.24**

Next, the amount of time required in seconds for the robot to completely traverse the path at the specified linear velocity was calculated.

$$t = \frac{c}{v}$$ **Equation 3.25**

Finally, the angular velocity in radians / second was calculated.

$$\omega = \frac{2\pi}{t}$$ **Equation 3.26**

3.7.2 Wheel Encoders Characteristics

The two wheel encoders measured the linear speed of the robot's two drive wheels. They had an update rate of 10 Hertz and an error rate of ± 0.01 meters / second.

3.7.3 Digital Compass Characteristics

The compass was modeled after a Honeywell HMR3100 digital compass. It had an update rate of 20 Hertz, a 0.5 degree resolution, and a heading accuracy of ± 5 degrees RMS. The compass was used to compute the robot's angular velocity.

3.7.4 Gyroscope Characteristics

The gyroscope was modeled after a Watson Industries DMS-E604 gyroscope. It had an update rate of 45 Hertz, a 0.1 degree resolution, and a heading accuracy of ± 0.05 degrees/second. The gyroscope was used to compute the robot's angular velocity.

3.8 Software

The simulation was written in the C# programming language for version 2.0 of Microsoft's .NET Framework. C# was chosen for a few different reasons. First, we wanted to create a graphical user interface, and .NET facilitates professional-quality GUI development quite well. Second, the author was already familiar with the language and platform since he is employed as a Microsoft .NET developer

The simulation was divided into three main assemblies[1]: MatrixLibrary.dll, Simulation.dll, and SimulationGUI.exe. The component-level architecture for application is shown below in Figure 3.4, and a brief explanation of each assembly is provided in the following subsections.

[1] "Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly." Microsoft Developer Network <http://msdn2.microsoft.com/en-us/library/hk5f40ct(vs.80).aspx>.

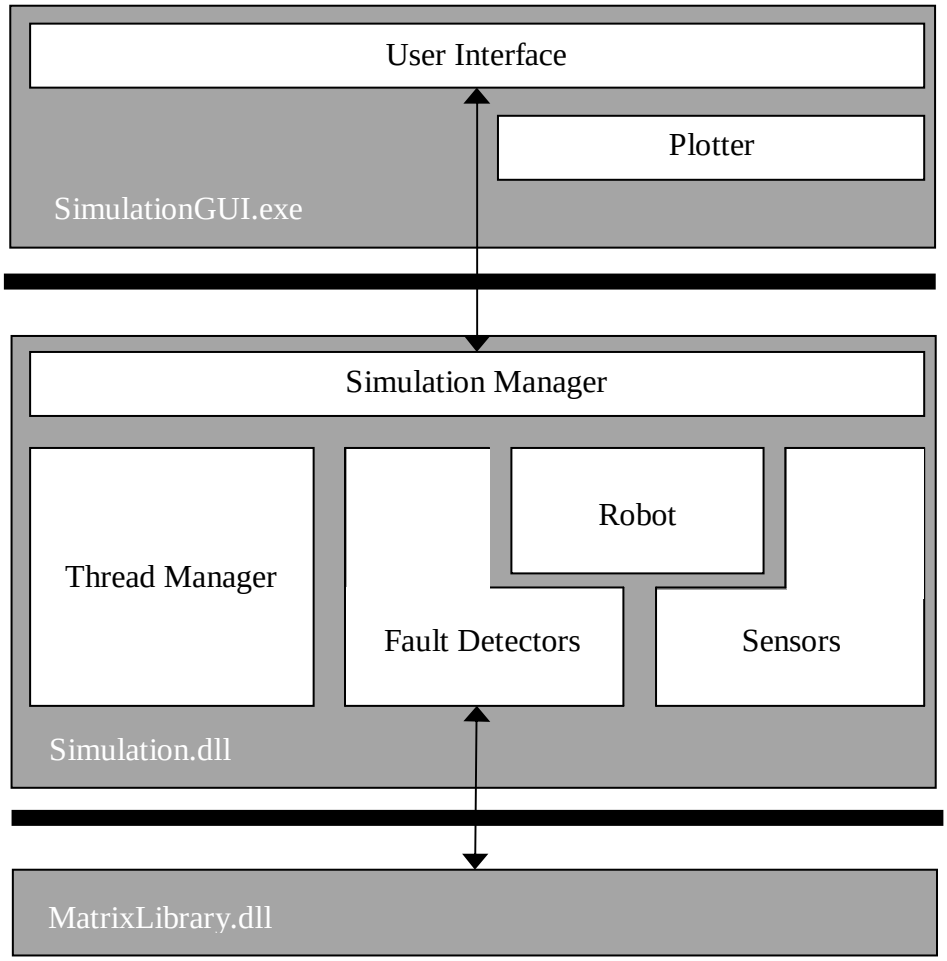

Figure 3.4: Component architecture

3.8.1 MatrixLibrary.dll

The MatrixLibrary assembly contains several classes used for performing mathematical operations on matrices, such as calculating determinants and inverses. It was created and freely distributed by The MathWorks, Inc. and the National Institute of Standards and Technology under the name DotNetMatrix. However, the author added a class, MatrixOperations, to perform several operations like reading a matrix from a text file, raising a matrix to a specified power, and formatting a matrix as a string to be printed to log files.

3.8.2 SimulationGUI.exe

The SimulationGUI assembly is the main executable of the application, and contains the classes used to construct the user interface, which is shown below in Figure 3.5.

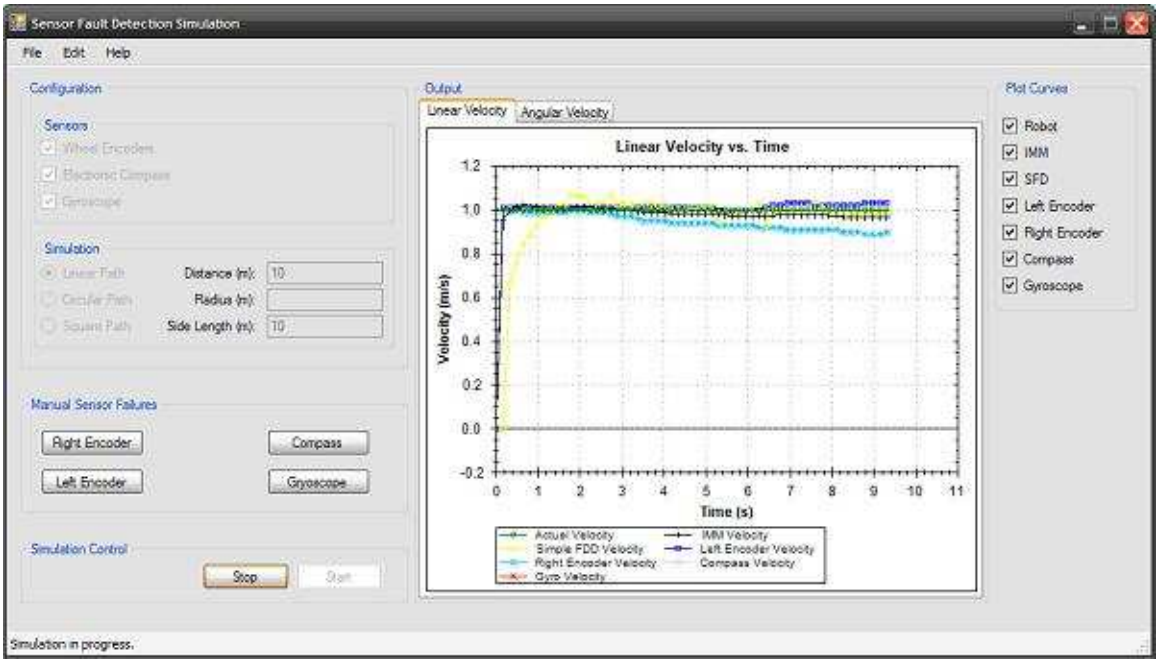

Figure 3.5: The simulation user interface during a test run

This assembly uses the freely available ZedGraph.dll[2] for constructing parts of the graph. The author created a Plotter class to display different curves on the plot (see the "Plot Curves" heading on the right side of Figure x), display plot points, display plot lines, change the sizes of the points and curves, and change the colors of the plot area and the surrounding area by selecting the appropriate options in the "Edit" menu.

An application configuration file is associated with the SimulationGUI assembly. Application configuration files in .NET share the same name as the executable file with a ".config" file extension. In this case, the file is named "SimulationGUI.exe.config". The configuration file is an XML file containing instrumentation settings, application settings that can be modified, such as the size of the application in pixels, the refresh period of the plot, and the state of the application among other things, and the reference paths to the

[2] <http://zedgraph.org>

afore mentioned associated assemblies. More information about the configuration file is presented in a later subsection.

3.8.3 Simulation.dll

The Simulation assembly contains the core simulation classes in three different namespaces[3]: Simulation, Simulation.Sensors, and Simulation.FaultDetectors. Each namespace contains several classes, but only a few important classes from each namespace are discussed below.

3.8.3.1 The Simulation Namespace

The Simulation namespace contains 10 classes, including ThreadManager, SimulationManager, and Robot. The ThreadManager is a static class used to create and manage threads as well as to dispose of them when a simulation run is completed, the stop button is clicked, or an error condition occurs. Each simulated entity (the robot, each sensor, and each fault detector) runs in its own thread.

The SimulationManager is responsible for starting, stopping, and passing information from the simulation to the user interface and vice-versa. When the start button on the application is clicked, the SimulationManager starts the robot, the fault detectors, and the sensors. While the simulation is running, the user interface calls the SimulationManager to get the latest simulated velocity values to plot for each curve every 150 milliseconds. The plot update rate can be changed by editing the PlotPeriod element in the application configuration file (SimulationGUI.exe.config). Once the robot completes the user-specified path, it raises an event notifying subscribers that the simulation run has completed.

The Robot class encapsulates the information presented in section Robot Characteristics. The robot never deviates from its user-specified path, nor does its linear velocity diverge from 1 m/s. This allows the experiment to focus on the simulated sensor

[3] "A namespace is a logical grouping of the names—identifiers—used within an application. Each name within a namespace is unique. A namespace contains only the name of a type, but not the type itself. A developer creates namespaces in order to organize classes into functional units. A namespace of names is analogous to a folder of files." C# Online.Net <http://en.csharp-online.net/Glossary:Definition_-_Namespace>

outputs and the outputs from the fault detectors since the actual robot velocity is known at all times.

3.8.3.2 The Simulation.Sensors Namespace

The Simulation.Sensors namespace contains nine classes, including the WheelEncoder, Compass, and Gyroscope sensor classes. Each of these classes encapsulates the information presented in section 3.7.

3.8.3.3 The Simulation.FaultDetectors Namespace

The Simulation.FaultDetectors namespace contains five classes, including the Imm and Sfd fault detector classes. The fault detectors use the values generated by the sensors in their calculations of the robot's velocity.

3.9 Program Usage and Features

Before using the program to run a simulation, it must first be configured. The top left side of the program window contains the configuration section as shown in Figure 3.6.

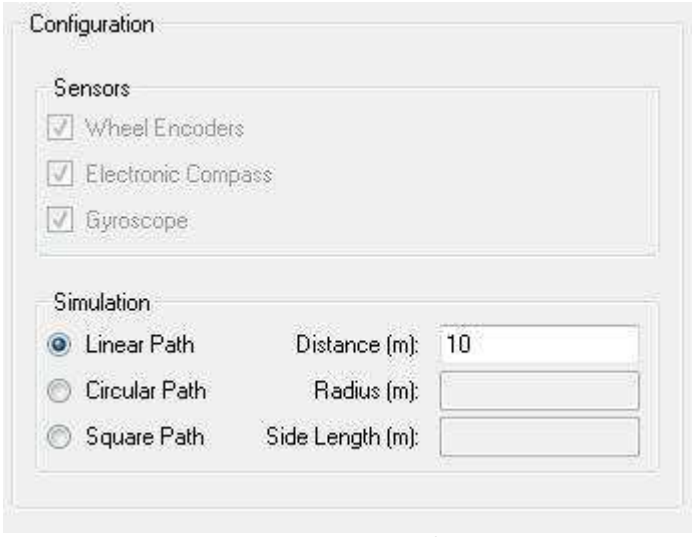

Figure 3.6: Simulation configuration

The configuration is intuitive and easy to use. In the 'Simulation' grouping, the user can select the type of path and the length of the path the robot should traverse. The 'Sensors' section was designed to allow the user to choose which sensors should be included on the

robot during a simulation run. However, this feature was a late addition to the program and was only partially implemented due to time constraints. It is currently disabled (non-selectable to the end user) and, instead, acts as a visual indicator of the sensors that are equipped to the robot.

Directly below the 'Configuration' group is the 'Manual Sensor Failure' group and 'Simulation Control' group as shown below in Figure 3.7.

Figure 3.7: Sensor fault and simulation controls

The purpose of the 'Simulation Control' group is to simply start and stop the simulation by clicking the appropriate button. The 'Manual Sensor Failures' section contains four buttons corresponding to each of the four sensors on the robot. During a simulation run, each of the buttons is enabled allowing the user to manually cause a sensor to fail.

The middle of the application contains the 'Output' group, which uses a tab control to display a linear velocity plot on the first tab and an angular velocity plot on the second tab. These two plots graphically display the appropriate velocities of the robot, each sensor, and each fault detector. Figure 3.8 shows a screenshot.

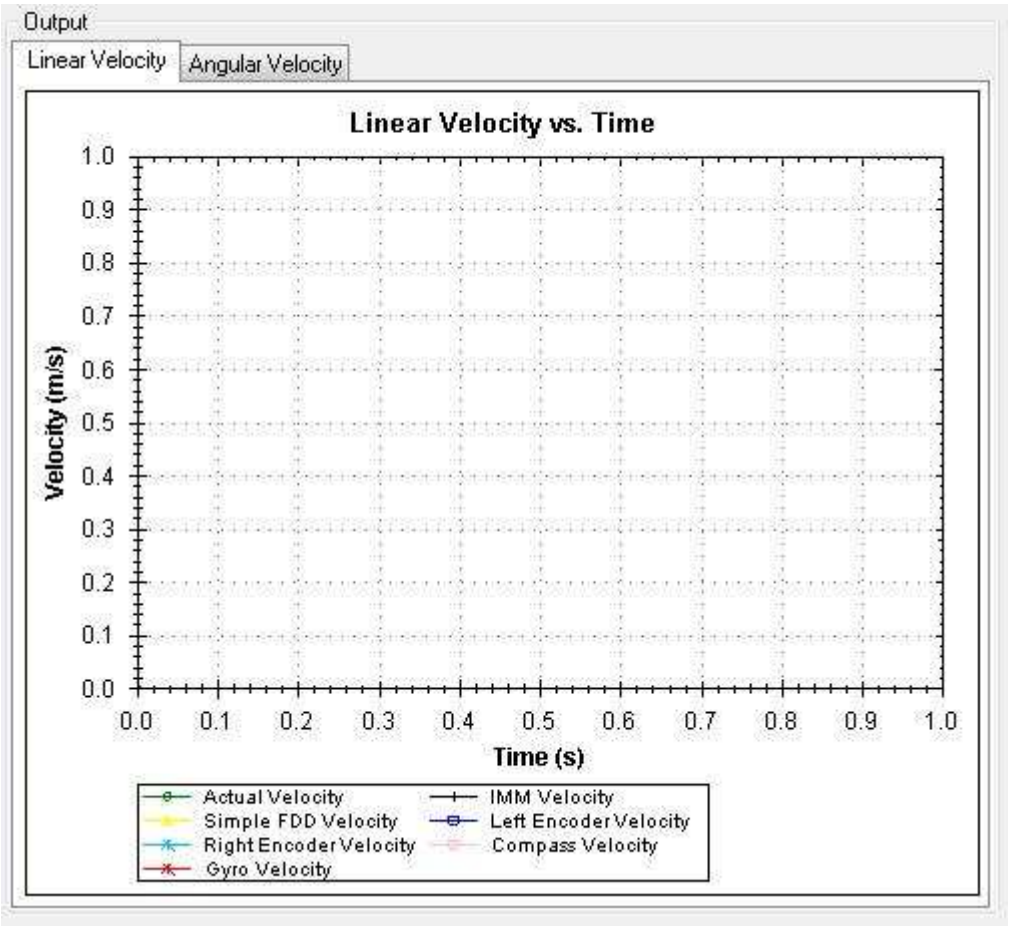

Figure 3.8: Simulation output

The far right side of the program window contains the 'Plot Curves' group, as shown in Figure 3.9.

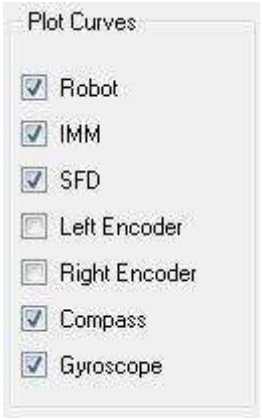

Figure 3.9: Plot curves

This grouping allows the user to choose which velocity curves to display on the plot by simply checking the appropriate checkbox.

The file menu allows the user to do common application tasks, like printing, print preview, and page setup. The program generates a simple report to send to the printer that contains the date, the sensors, the simulation type, and both plots on a single page. The file menu also allows the two plots generated by a simulation run to be saved as a portable network graphics (PNG) file.

The edit menu allows the user to control the visual aspects of the plots, such as whether to show the plot points, the lines connecting the points, or both. The user can also choose the size of the lines and plot points, as well as the colors of the graph and whether to display a grid.

Other plot options can be selected via a context menu on the plot itself. The context menu can be displayed by right-clicking anywhere on the plot. Some of the context menu items include zooming, un-zooming, displaying the point values on a mouse over, printing the plot, and saving the plot.

The simulation application also uses an external XML configuration file that was alluded to earlier, SimulationGUI.exe.config. The configuration file is used to store application settings and user preferences, like the colors of the plots, whether plot points were displayed or lines were displayed, which simulation type and length was selected, which plot curves were checked, and several other items. The configuration file can be manually modified but probably shouldn't be necessary under most circumstances. The one item that a user may want to modify is the plot period, which is specified in milliseconds. The default value is 150 milliseconds, and it probably should not need to be changed.

The last feature of the program is logging. Each simulated component has an associated logger object that outputs values to text files in the 'logs' directory of the application install path. Each log is appropriately named and contains the output values from each component.

The install path of the application contains three subdirectories, 'bin', 'logs', and 'resource'. The logs directory has already been mentioned. The bin directory contains the assemblies (DLLs) that were previously described in this chapter. The resource directory contains the transition probability matrix shown in Figure 3.2 as a text file to be used by the IMM.

4 Results

4.1 Test Scenarios

This chapter provides an analysis of the results generated by the simulation application. Several test scenarios were identified under which the simulation was run in order to appropriately compare and contrast the performance of the IMM and SFD fault detectors.

For each robot path type, the simulation was first run without any sensor failures to produce a baseline. Afterward, different combinations of sensor faults were incorporated into the simulation runs. Table 4.1: Simulation test scenarios depicts the test scenarios for each robot path type.

Robot Path Type	**Sensor Faults Tested**
Straight Line (10 meters long)	• No faults • Right encoder • Both wheel encoders • Compass
Square (5 meters x 5 meters)	• No faults • Right encoder • Compass
Circle (3 meter radius)	• No faults • Right encoder • Compass • Compass and gyroscope

Table 4.1: Simulation test scenarios

These test scenarios are not exhaustive. There are 16 possible system modes, which mean there are at least 16 possible test scenarios for each robot path type. However, only a few different scenarios produce unique results. As such, the tests listed

above reflect only distinctive and interesting results. For example, the combination of a compass fault and a gyro fault in a linear path test run was neither unique nor interesting. When the robot was moving in a straight line, the angular velocity was zero. Thus, if either the compass or gyroscope failed, the resultant angular and linear velocities remained virtually unchanged.

4.2 Hypotheses

A few hypotheses were developed before generating any test data with the simulation program. Table 4.2 lists each hypothesis, and the following subsections discuss each one in turn.

1	The SFD will respond to any sensor faults.
2	The SFD has a shorter fault response time.
3	The IMM is more accurate than the SFD.

Table 4.2: Simulation hypotheses

4.2.1 Hypothesis 1: The SFD will respond to any sensor faults

The SFD should respond to any sensor fault during the simulation. However, the SFD's response will not be as accurate as the IMM's response, but it is not expected to be.

4.2.2 Hypothesis 2: The SFD has a shorter fault response time

The SFD has fewer computational steps per cycle than the IMM. In fact, it eliminates the use of any type of Kalman filtering. When a sensor fails, the SFD should have a faster response time than the IMM.

4.2.3 Hypothesis 3: The IMM is more accurate than the SFD

Based on the literature, the IMM has an excellent track record in terms of performance and accuracy. On the other hand, the SFD lacks some of the IMM's sophistication and uses simpler methods to detect faults that are not as robust.

4.3 Results

The results section of this chapter is divided into three subsections, one for each robot path type, as depicted above in Table 4.1. For each test listed in that table, two output plots are shown: linear velocity versus time and angular velocity versus time. An analysis of each plot follows.

4.3.1 Linear Path Simulation Results

All of the linear path simulation test runs shared a common path length of 10 meters. Since the robot was traveling at a linear velocity of 1 m/s, each plot ends near the 10 second mark. Figure 4.1 shows the baseline linear velocity plot when no sensor failures occur.

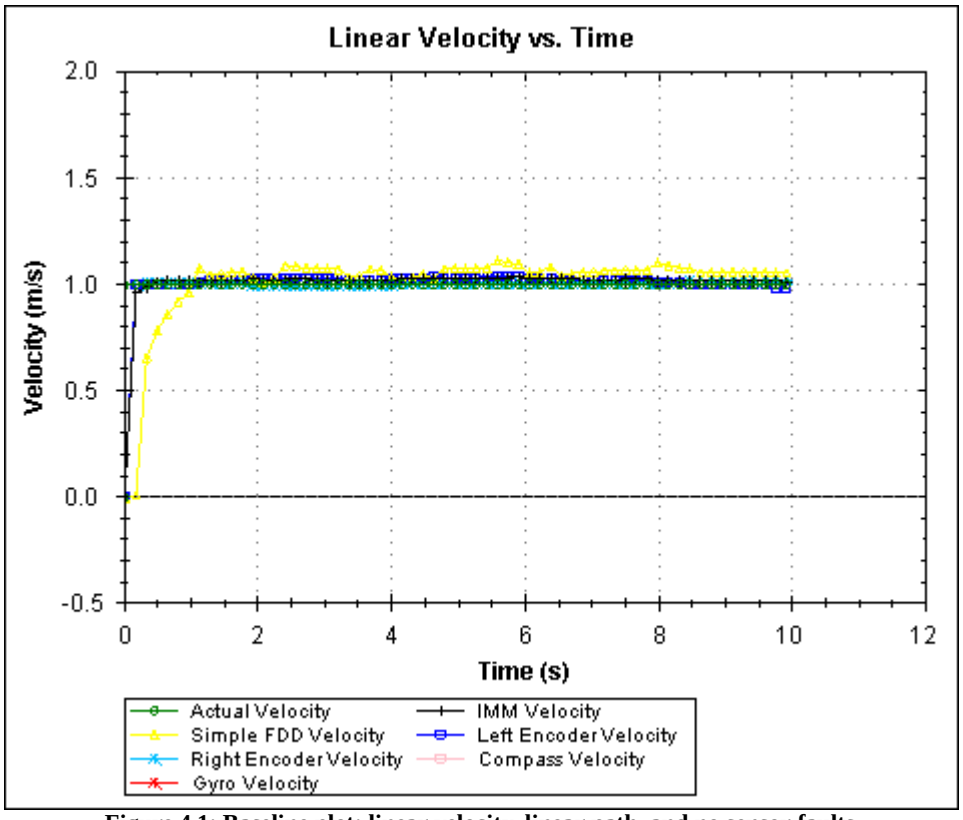

Figure 4.1: Baseline plot: linear velocity, linear path, and no sensor faults

There are no surprises in Figure 4.1 except for the slow initialization of the SFD plot when compared to the IMM. The second hypothesis from the previous section stated that the SFD should respond to a sensor fault quicker than the IMM. While there are no

sensor faults in this plot, the slow initialization can be easily explained. The IMM makes an initial assumption of the robot's velocity while the SFD does not.

The corresponding baseline angular velocity plot is shown below in Figure 4.2, with both fault detectors correctly calculating an angular velocity very close to zero rad/s. This was the expected value since the robot was moving along a linear path.

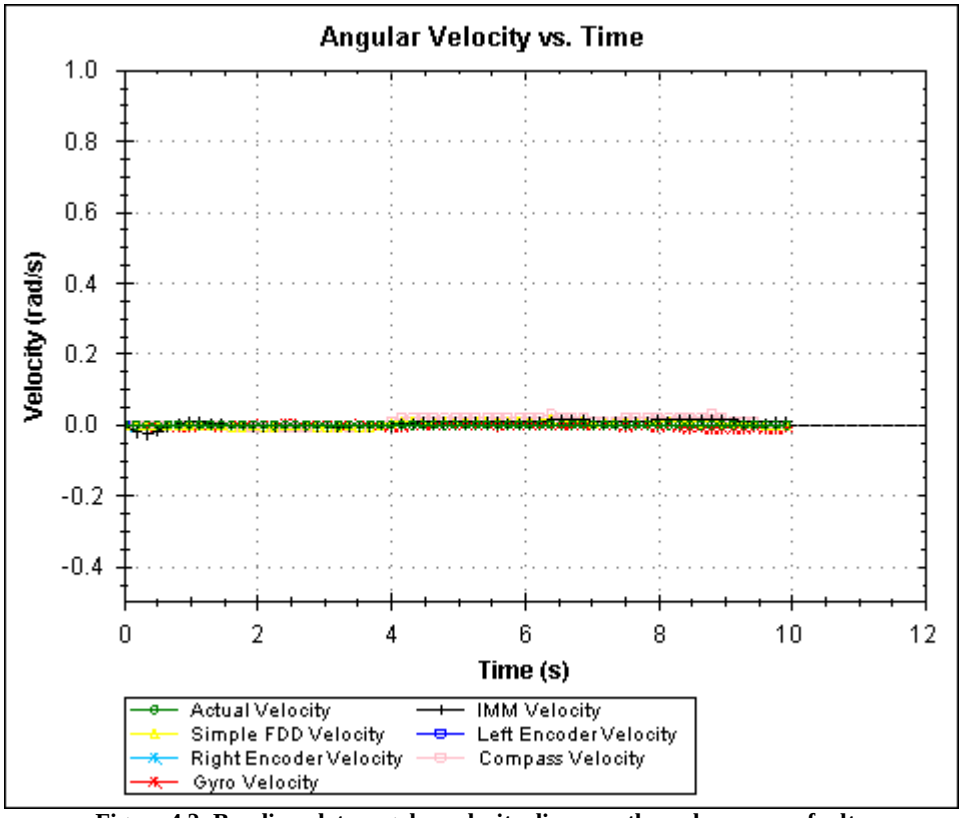

Figure 4.2: Baseline plot: angular velocity, linear path, and no sensor faults

The next set of plots show the results for a right wheel encoder failure at roughly 4 seconds into the simulation run. The linear velocity plot, shown in Figure 4.3, clearly shows that the SFD reacted to the sensor fault instantaneously. This plot also indicates that the SFD is very sensitive to failures, as it spiked to zero and then very quickly recovered to half the actual velocity, which is what we expected. The SFD is a simple heuristic that produces an average of the sensor outputs.

The IMM, conversely, handled the sensor failure appropriately and continued to produce the correct linear velocities. This was expected because only one sensor, the left

wheel encoder, was available for calculating the linear velocity. This graph does not display the actual velocity plot so that the graph is less convoluted and easier to read. The actual velocity is known to be 1 m/s, as mentioned in the previous chapter.

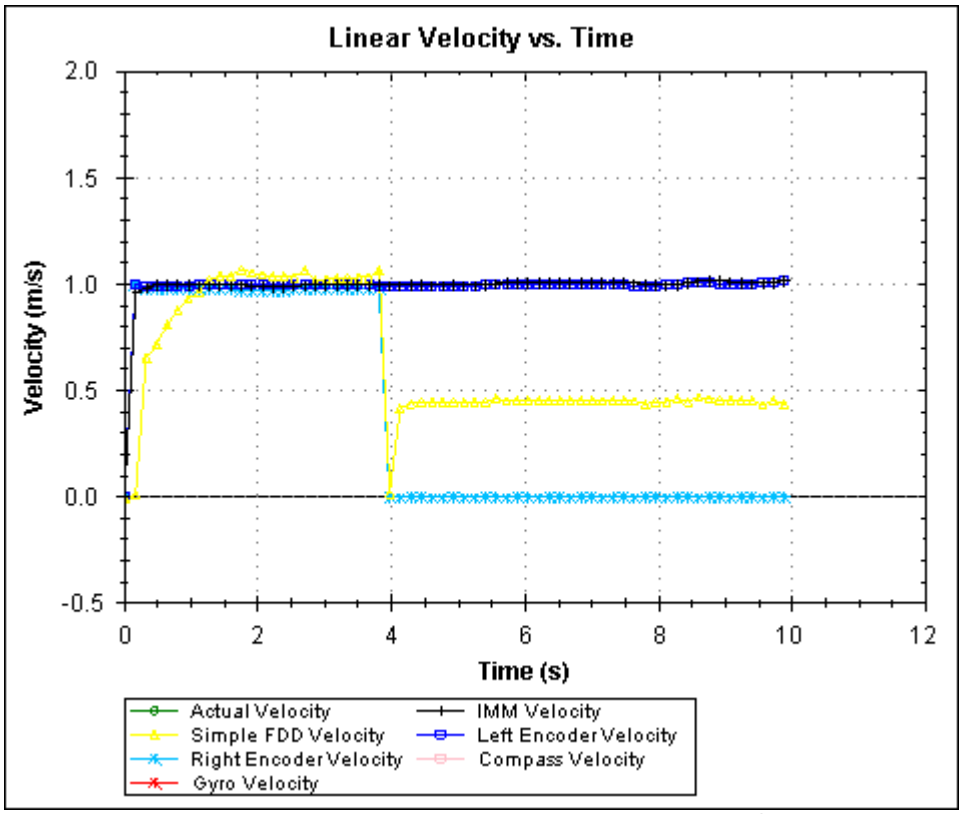

Figure 4.3: Linear velocity, linear path, and right encoder fault

The angular velocity plot, shown in Figure 4.4, is very similar to the baseline plot in Figure 4.2, as it should be. The only difference between the two is the velocity calculated from the compass output. The compass had an accuracy of ±5 degrees rms, while the gyroscope had an accuracy of ±0.05 degrees/second, as explained earlier in section 3.7 Robot and Sensor Characteristics. This fact accounts for the discrepancy between the compass and gyroscope.

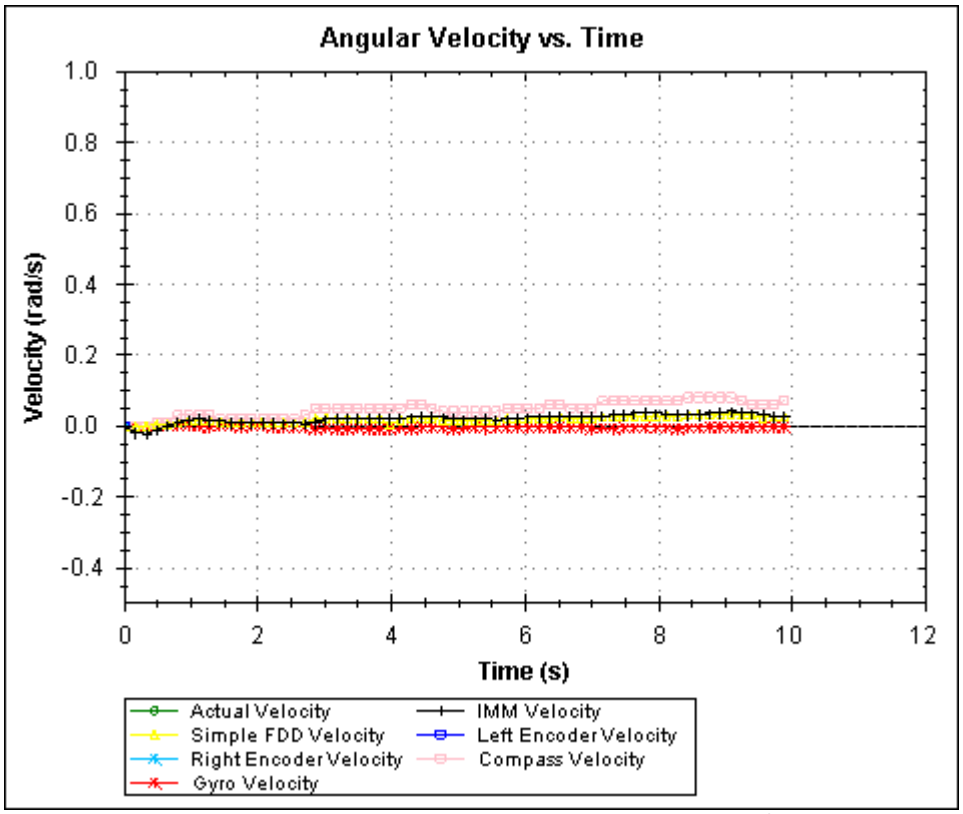

Figure 4.4: Angular velocity, linear path, and right encoder fault

Figure 4.5 shows two wheel encoder faults, one at roughly 1 second, and another at roughly 2 seconds. After the second encoder fault, the IMM dropped to zero since there were no other sensors available for calculating linear velocity. The SFD also dropped to zero, though the robot's actual velocity remained unchanged at 1 m/s.

The IMM and SFD both responded to the sensor faults at nearly the same instant. There has been no evidence presented at this point to validate the second hypothesis that the SFD will react to a sensor fault quicker than the IMM. However, the third hypothesis, which stated that the IMM would produce more accurate results than the SFD, was validated in Figure 4.5.

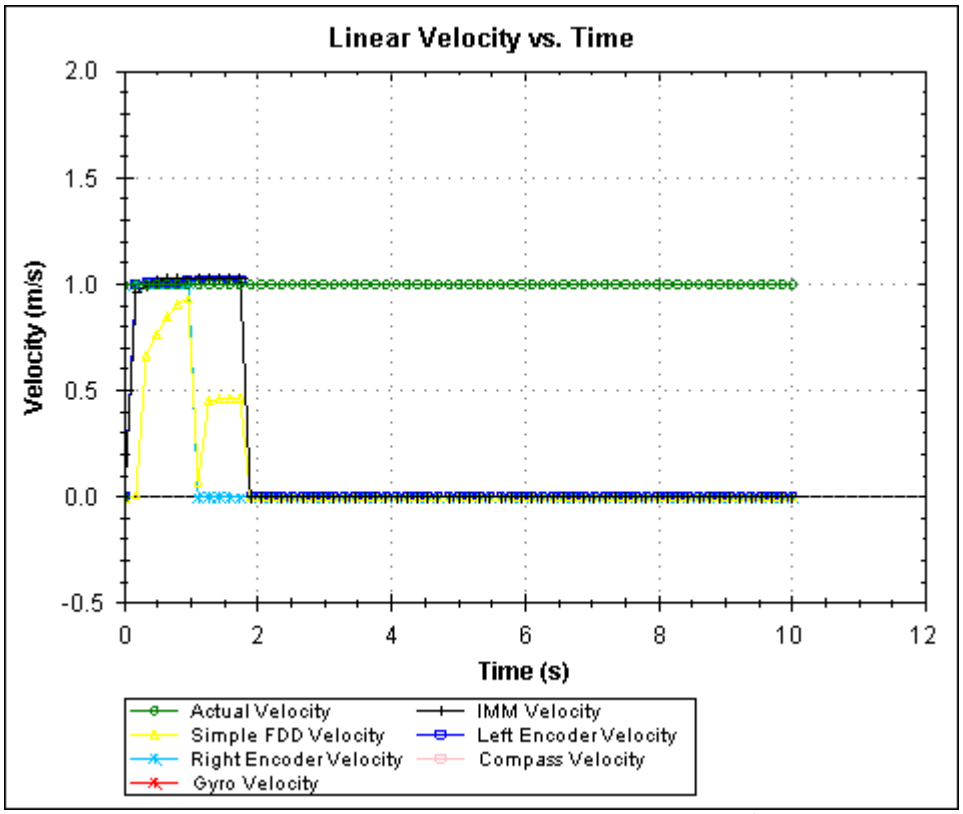

Figure 4.5: Linear velocity, linear path, and both encoder faults

Figure 4.6 shows the corresponding angular velocity plot of both wheel encoder failures. As expected, the IMM and SFD both produced angular velocities near zero.

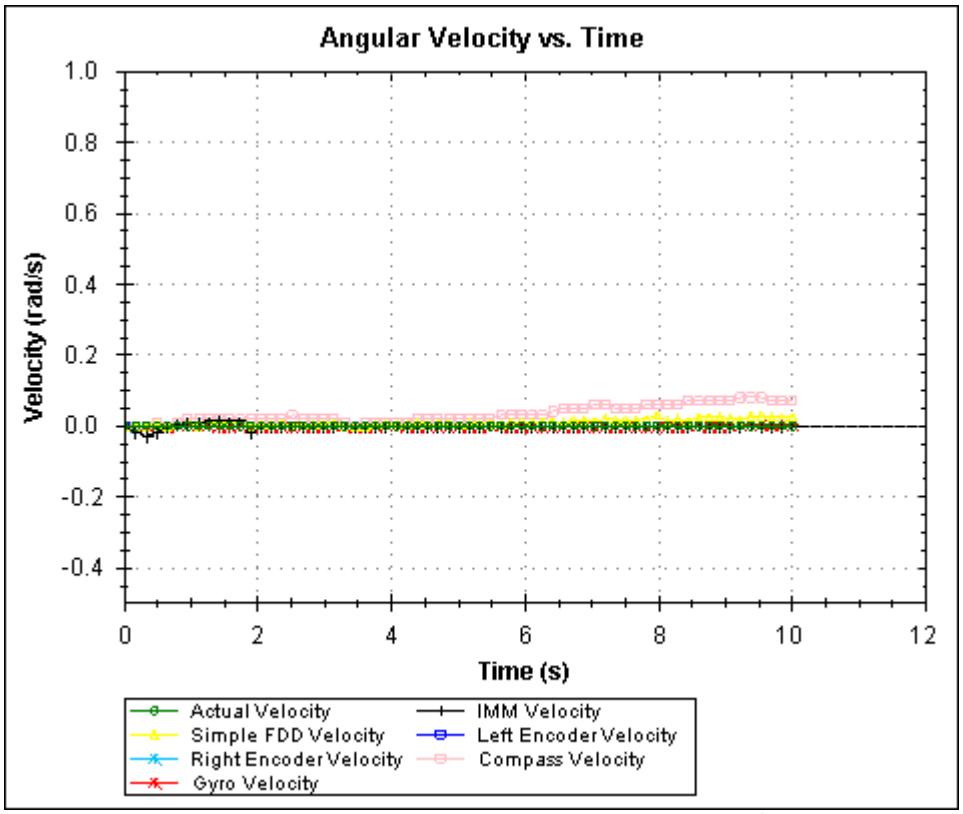

Figure 4.6: Angular velocity, linear path, and both encoder faults

The next two plots show the effects of a compass failure when the robot was moving in a linear path. The compass fault occurred at 5 seconds; however neither Figure 4.7 nor Figure 4.8 shows any change in the IMM and SFD velocities. In fact, both figures are nearly identical to their respective baseline plots. The results are not surprising since the compass was used to calculate angular velocity and the robot was moving in a linear path.

The important point to notice in this scenario is that an angular velocity sensor fault may not be evident as long as the robot maintains a linear path. This should not be a problem for the IMM because once the robot deviates from the linear path, the sensor failure would become evident, and it would be handled accordingly. On the other hand, it may be problematic to determine precisely when the actual failure occurred.

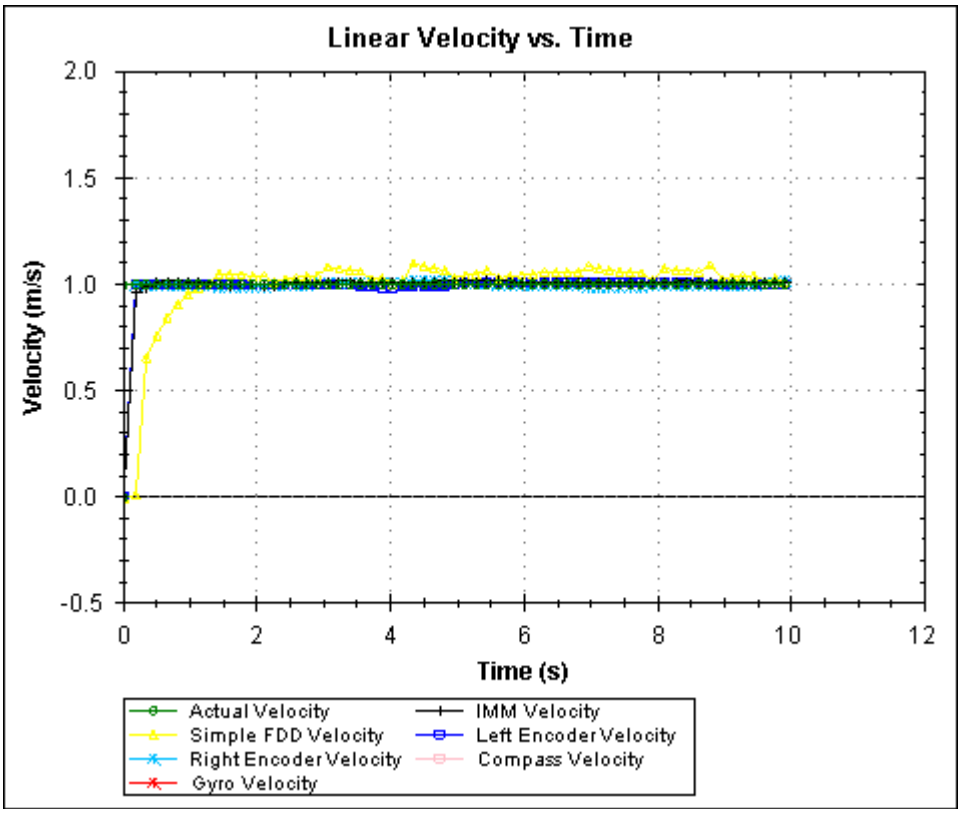

Figure 4.7: Linear velocity, linear path, and compass fault

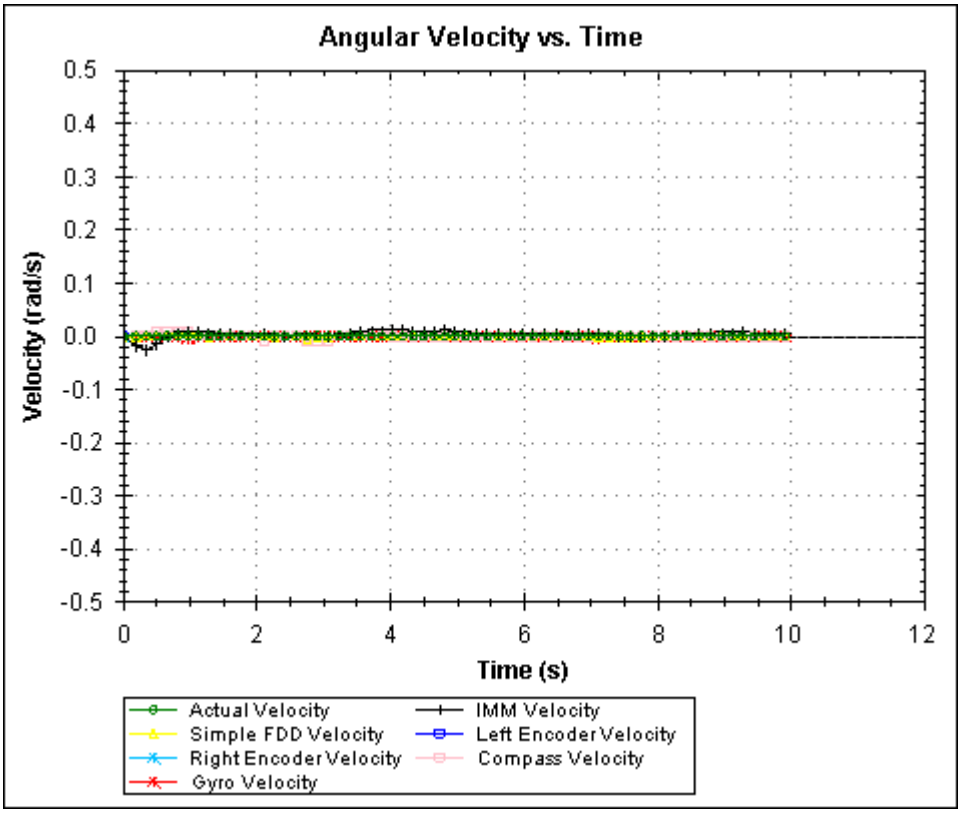

Figure 4.8: Angular velocity, linear path, and compass fault

4.3.2 Square Path Simulation Results

The square path results were generated by the robot traversing a square with a height and width of five meters in the counter-clockwise direction, as shown below in Figure 4.9.

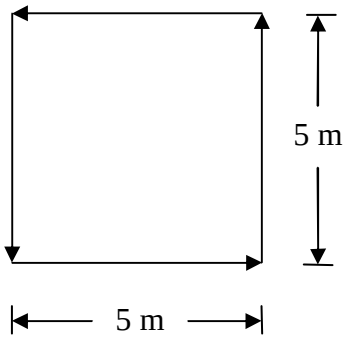

Figure 4.9: Square path dimensions and robot travel direction

The robot moved along each side of the square with a linear velocity of 1.0 m/s. When it reached the end of a side, it stopped for 1.5 seconds, turned 90 degrees at 1.047 radians/second, stopped for another 1.5 seconds, and then proceeded to move linearly for another five meters. Figure 4.10 and Figure 4.11 show the baseline plots of only the robot's actual linear and angular velocities in order to help visually demonstrate this.

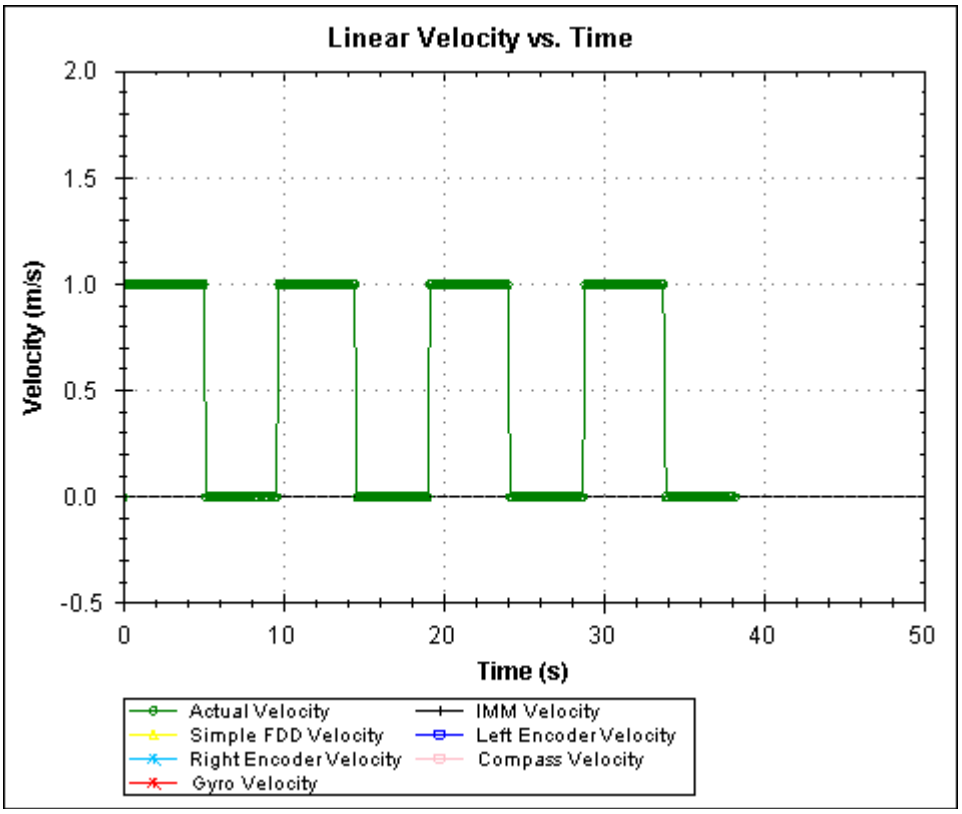

Figure 4.10: Baseline plot: Square path, linear velocity, and robot only

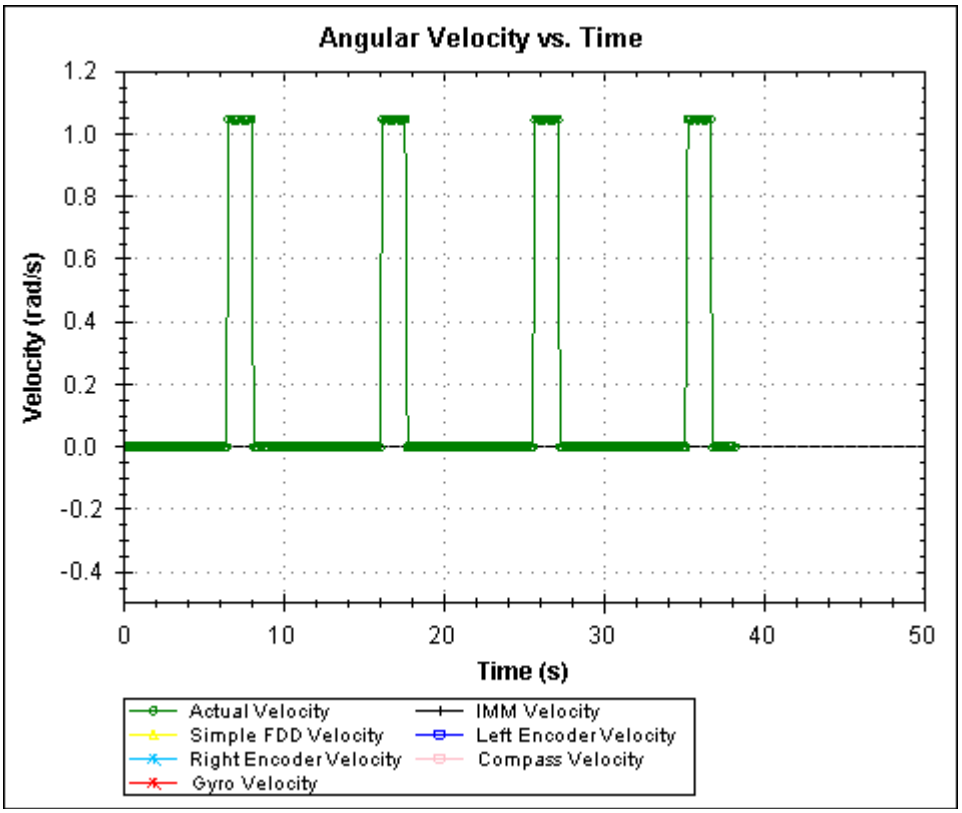

Figure 4.11: Baseline plot: Square path, angular velocity, and robot only

Figure 4.12 shows the corresponding sensor and fault detector linear velocity baseline plot, while Figure 4.13 shows the corresponding angular velocity baseline plot. There are a few interesting items to note in each.

At roughly 6.5 seconds on the linear velocity plot, shown in Figure 4.12, the wheel encoders indicated wheel velocities of the same magnitude but in different directions for approximately 1.5 seconds. Since the robot used differential steering, it made all of its turns by causing the two drive wheels to spin in opposite directions at the same speed.

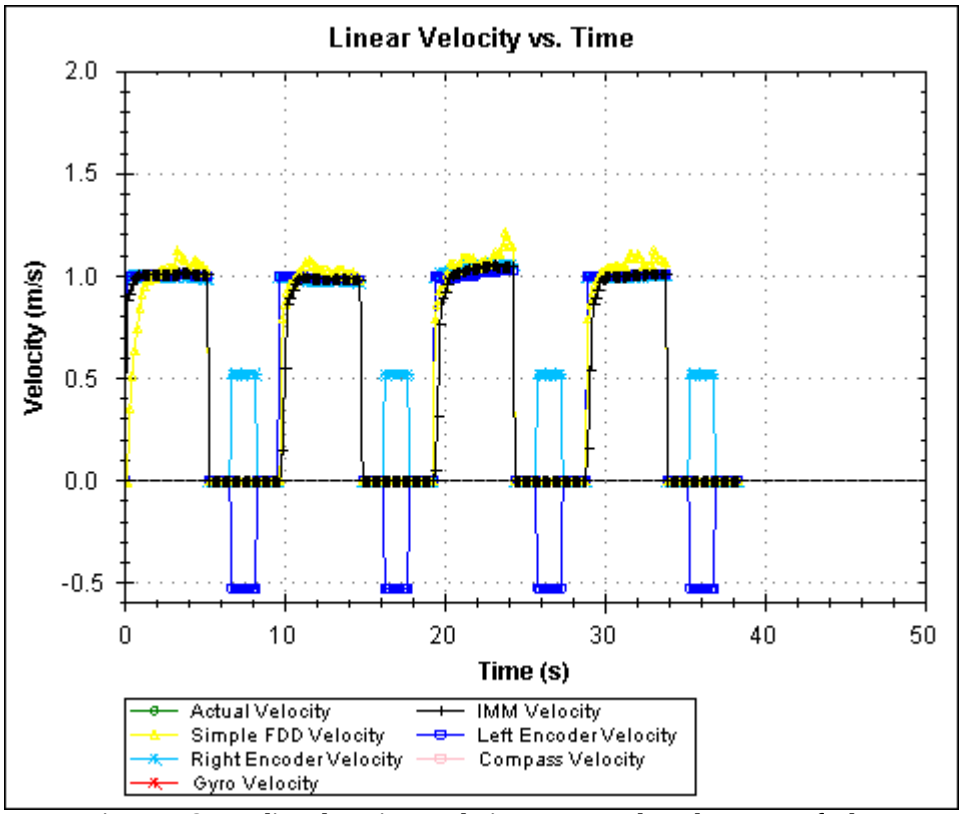

Figure 4.12: Baseline plot: Linear velocity, square path, and no sensor faults

One other thing to note in Figure 4.12 is the inaccuracy of the SFD. This plot further supports the third hypothesis that the IMM will provide more accurate estimation results than the SFD.

The sensor and fault detector angular velocity baseline plot is shown in Figure 4.13. This plot, as expected, is nearly identical to the robot's baseline plot in Figure 4.11.

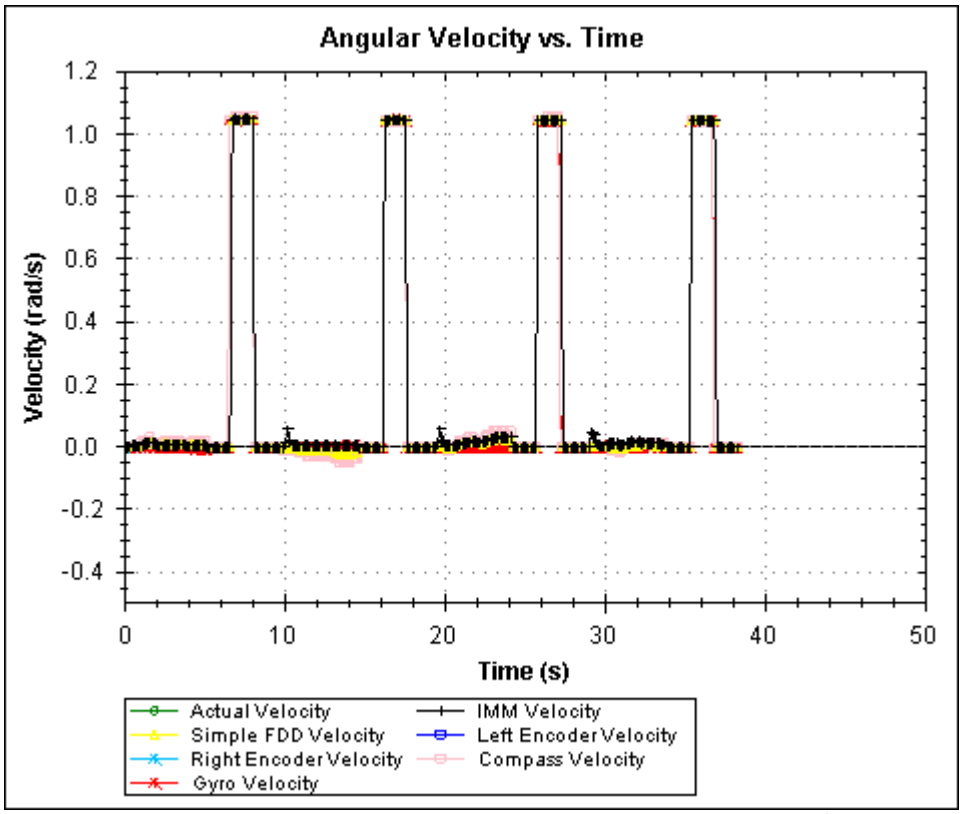

Figure 4.13: Baseline plot: Angular velocity, square path, and no sensor faults

Figure 4.14 shows a right wheel encoder failure at 12.5 seconds. The SFD immediately dropped to zero and then jumped halfway back up to the average. The IMM correctly discarded the faulty sensor and correctly calculated the robot's linear velocity.

During the robot's second turn at 16 seconds, and every turn thereafter, the IMM incorrectly calculated the robot's linear velocity. This was not surprising because after the right wheel encoder failed, the left encoder was the only sensor available for calculating linear velocity.

Figure 4.15 shows the corresponding angular velocity plot, which is not very interesting. It is nearly identical to the baseline plot, as expected.

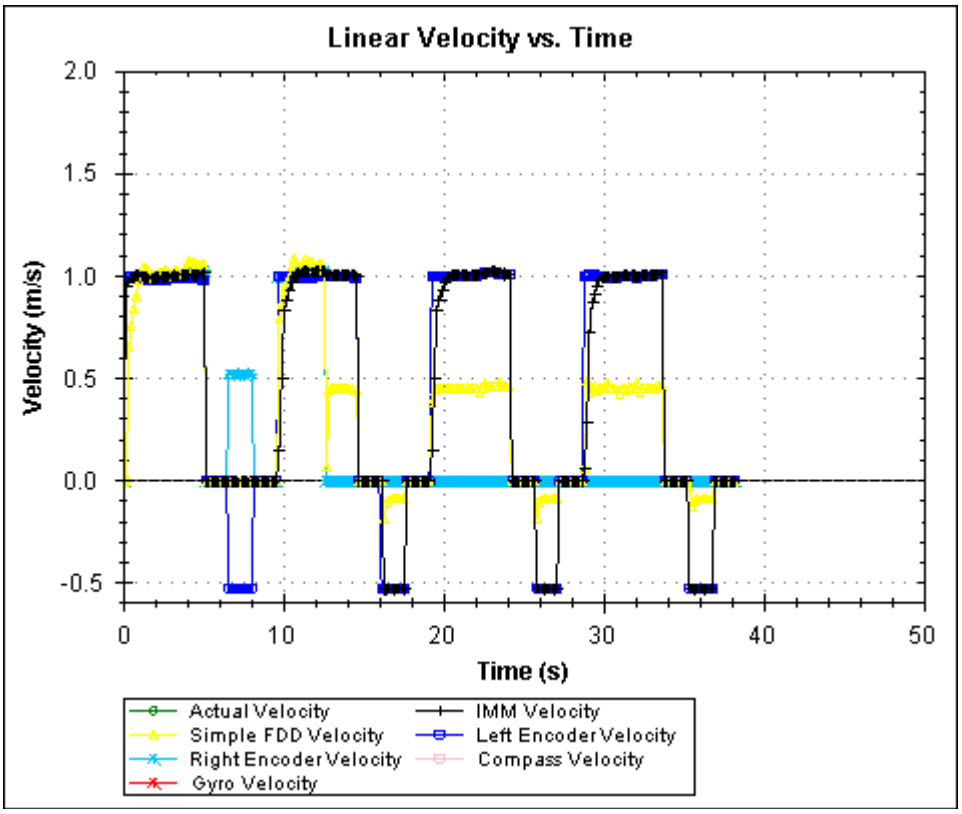

Figure 4.14: Linear velocity, square path, and right encoder fault

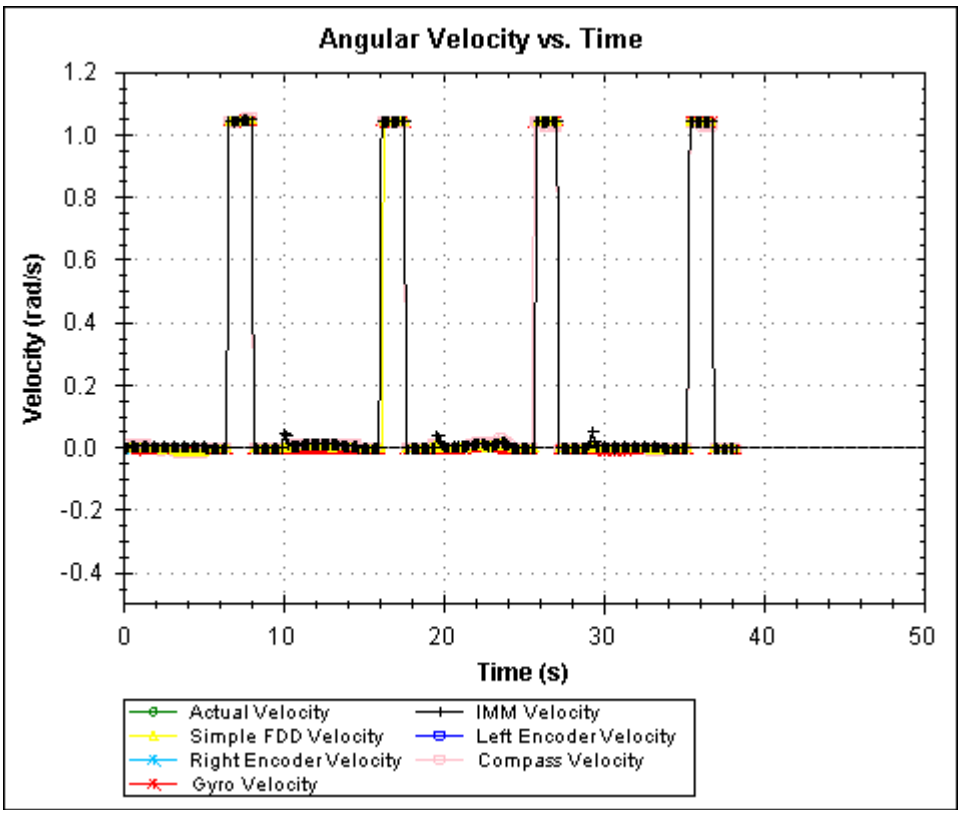

Figure 4.15: Angular velocity, square path, and right encoder fault

The final square path test scenario was a compass failure. The linear velocity plot produced by this scenario was identical to the baseline plot and was not included in this document. However, the angular velocity plot is shown below in Figure 4.16.

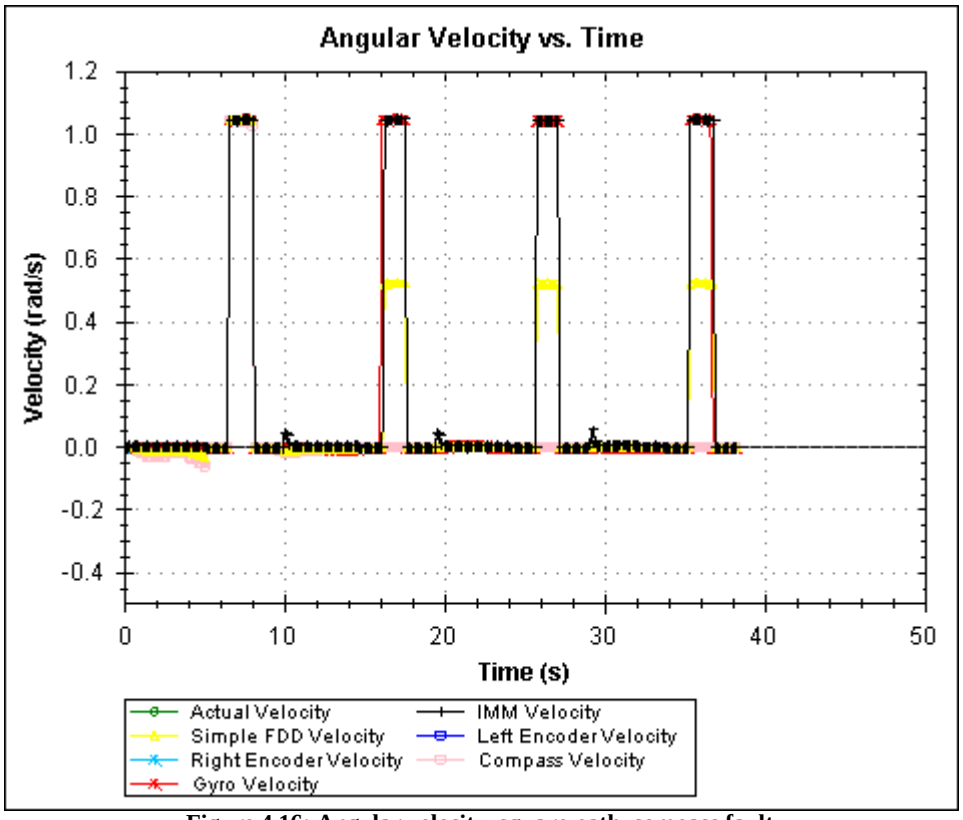

Figure 4.16: Angular velocity, square path, compass fault

The compass fault occurred at 12 seconds, but the angular velocity at that time was already zero. However, the following robot turn, at approximately 16 seconds, shows the compass velocity at zero. The IMM correctly calculated the robot's angular velocity and the SFD calculated the average.

4.3.3 Circular Path Simulation Results

The circular path results were generated by the robot traversing a circle with a radius of three meters in the counter-clockwise direction, as shown below in Figure 4.17.

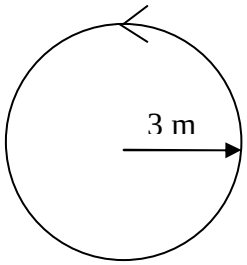

Figure 4.17: Circular path dimensions and robot travel direction

The robot traversed the circle with a linear velocity of 1.0 m/s. The angular velocity, calculated using Equation 3.24, Equation 3.25, and Equation 3.26 was $\frac{1}{3}$ rad/s. Figure 4.18 and Figure 4.20 show the linear velocity and angular velocity baseline plots, respectively.

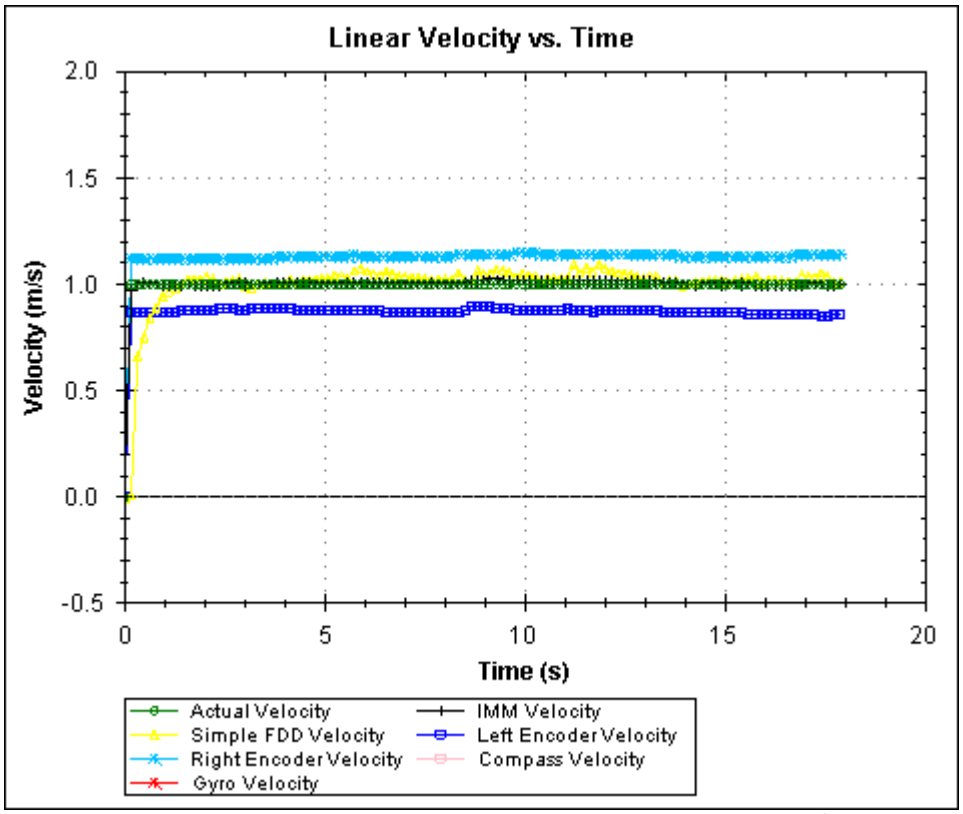

Figure 4.18: Baseline plot: Linear velocity, circular path, and no sensor faults

The right wheel velocity was slightly larger than the left wheel velocity in Figure 4.18 because the robot used differential steering to move in a circle. Thus, there were three angular velocities: one for the left drive wheel, one for the right drive wheel, and

one for the robot's center point. Each angular velocity was calculated using the equations mentioned above, but each had a different radius. Figure 4.19 shows a diagram to illustrate the different radii, where RE and LE are the right and left wheel encoders, respectively.

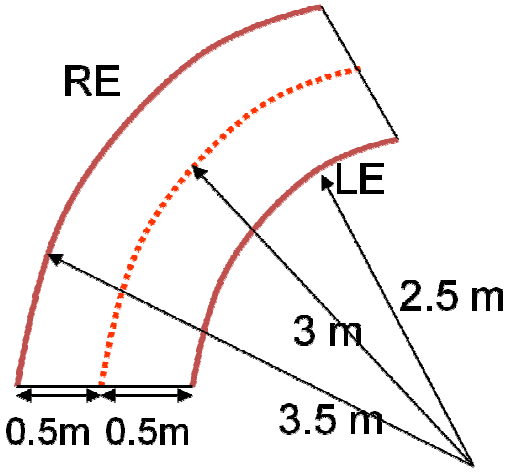

Figure 4.19: Circular path, angular velocity calculation

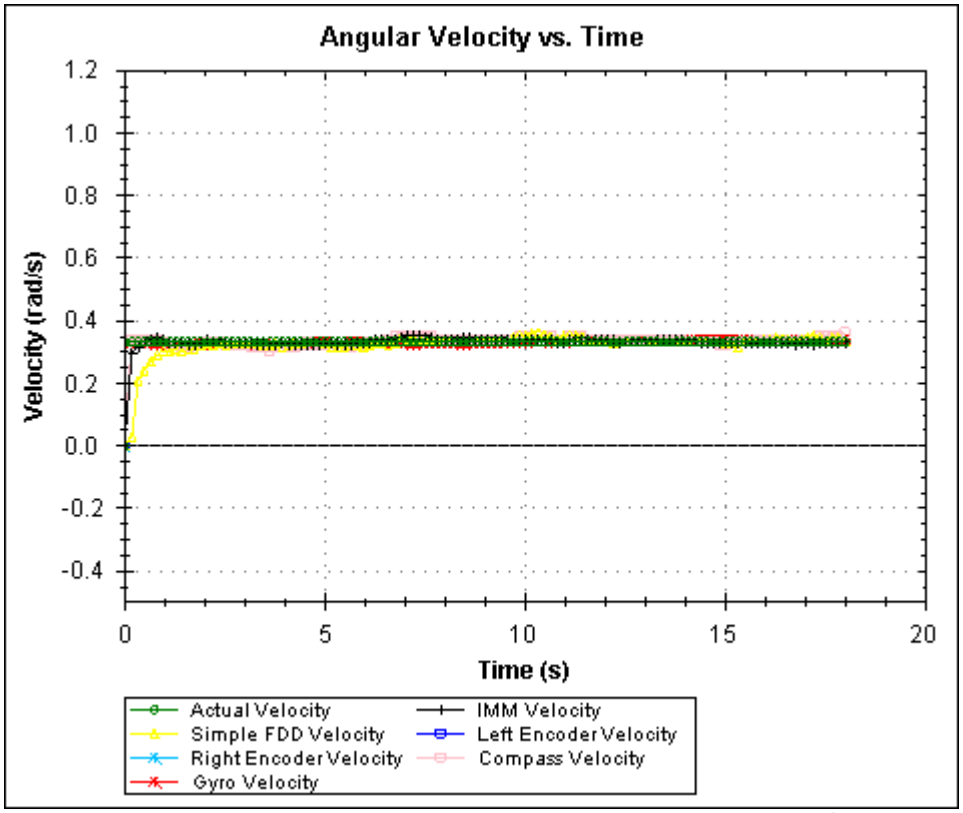

Figure 4.20: Baseline plot: Angular velocity, circular path, and no sensor faults

Figure 4.21 shows a linear velocity plot with a right wheel encoder failure at 7.5 seconds. The IMM responded by following the left wheel encoder since it was the only remaining sensor used to calculate linear velocity. The SFD, on the other hand, dropped to zero and then immediately jumped up to the average linear velocity.

This linear velocity plot did provide evidence to support the second hypothesis that the SFD will react to sensor faults quicker than the IMM. Though the SFD responded slightly faster, the IMM, again, produced more accurate results.

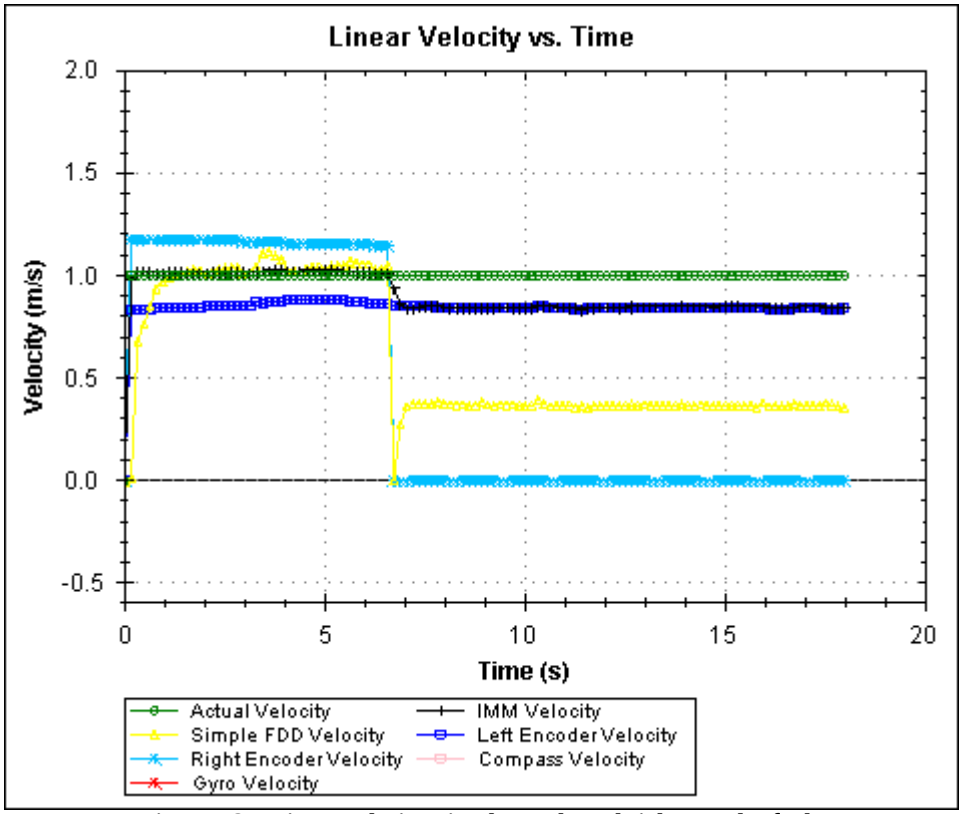

Figure 4.21: Linear velocity, circular path, and right encoder fault

Figure 4.22 shows the corresponding angular velocity plot of the wheel encoder fault. It is very similar to the baseline plot, except for the SFD anomaly when the encoder failed. However, the SFD quickly recovered from that sharp spike.

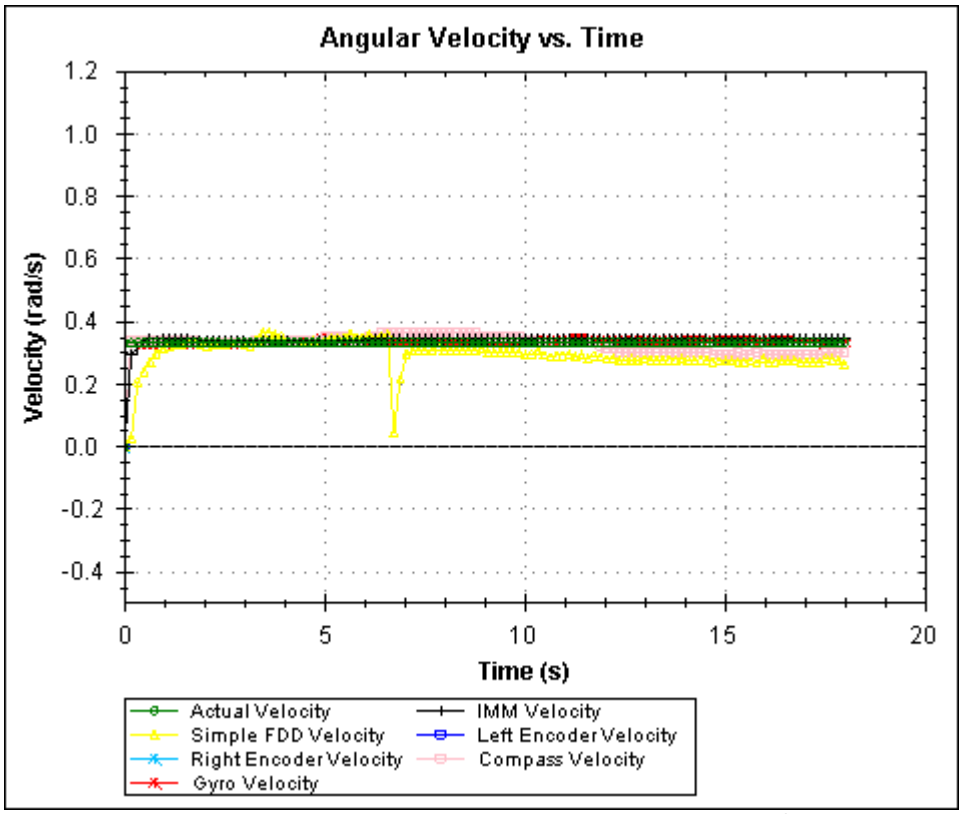

Figure 4.22: Angular velocity, circular path, and right encoder fault

Figure 4.23 shows the linear velocity plot of a compass failure at approximately 5.5 seconds. Since the compass was used only to calculate angular velocity, its failure had no effect on either fault detector's linear velocities. The plot is identical to the baseline plot.

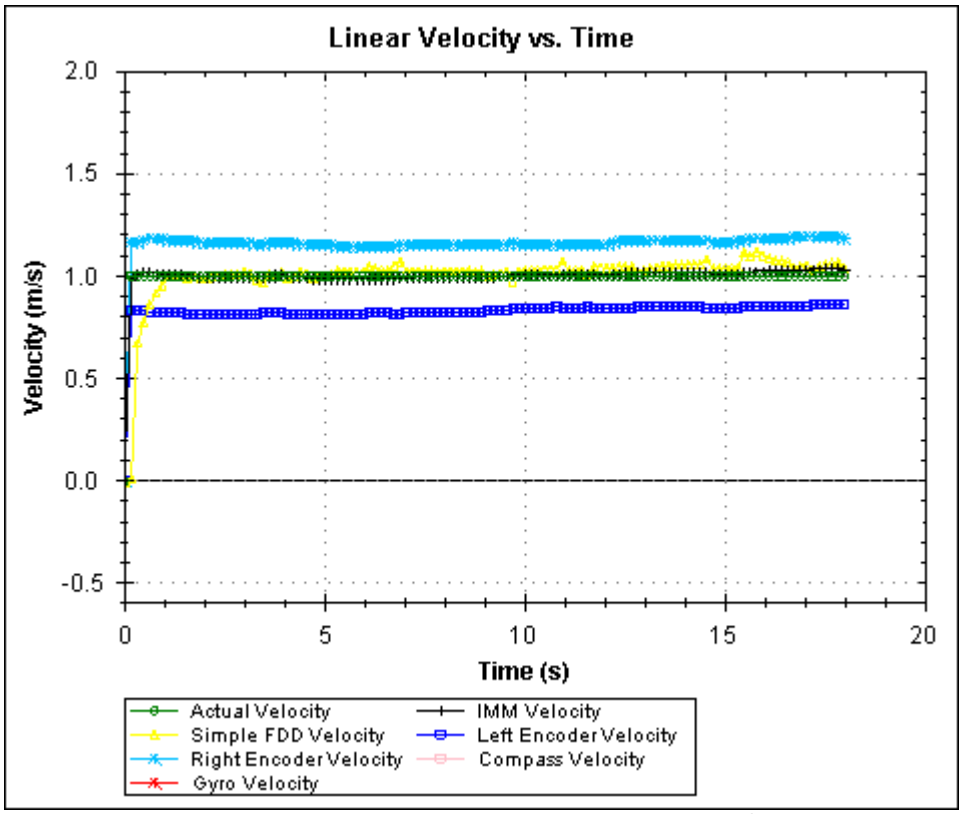

Figure 4.23: Linear velocity, circular path, and compass fault

Figure 4.24 shows the angular velocity plot of the compass failure. The IMM correctly calculated the robot's angular velocities, while the SFD dropped to the average value. Given the previous test plots, these results were not surprising. In fact, they were expected.

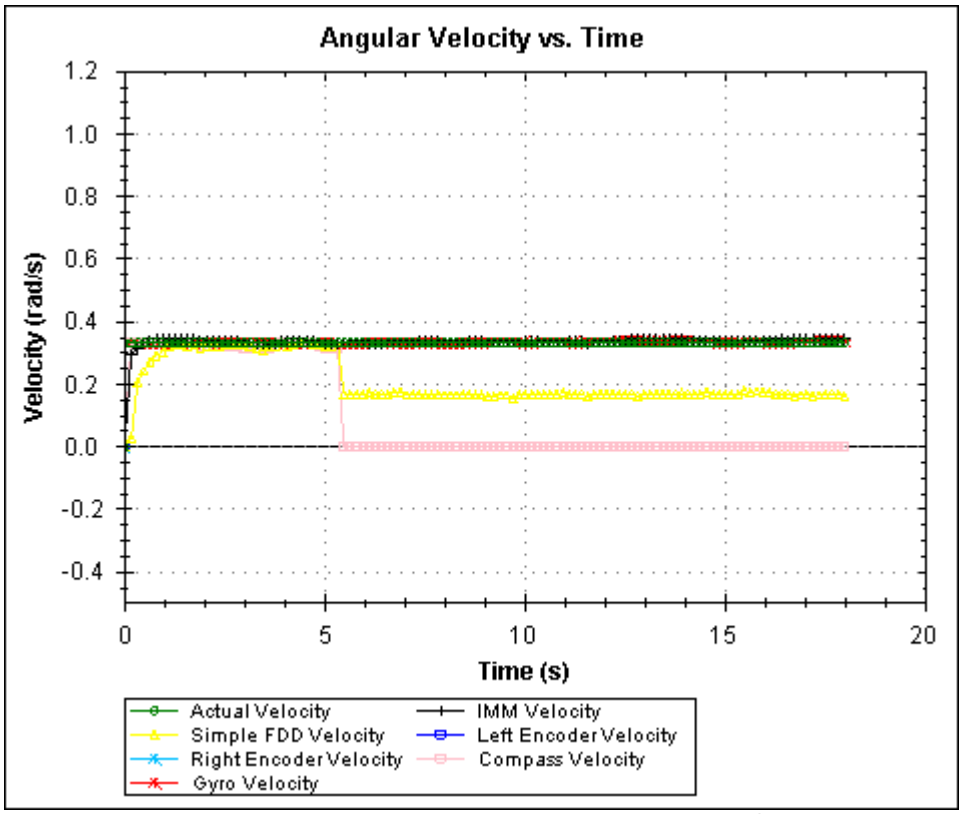

Figure 4.24: Angular velocity, circular path, and compass fault

The final test scenario for the circular path was to force the compass and the gyroscope to fail. Figure 4.25 and Figure 4.26 show the corresponding linear and angular velocity plots. The linear velocity plot was nearly identical to the baseline plot since neither failed sensor was used to calculate linear velocity. The angular velocity plot was entirely expected, though.

First, the compass failed at 10 seconds. At that point, the plot resembled the angular velocity plot shown in Figure 4.24. Almost two seconds later, the gyroscope failed, causing the IMM and SFD to correctly drop to zero.

The following chapter analyzes the results presented in this chapter. Following that discussion are the concluding remarks of this work.

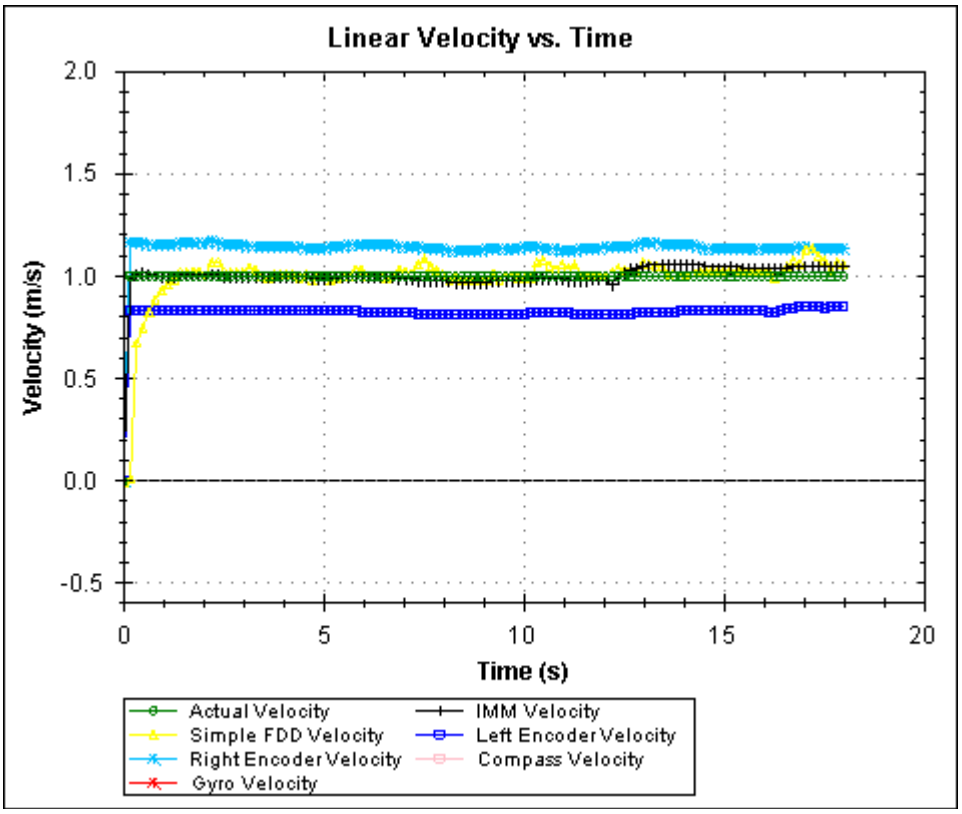

Figure 4.25: Linear velocity, circular path, compass fault, and gyro fault

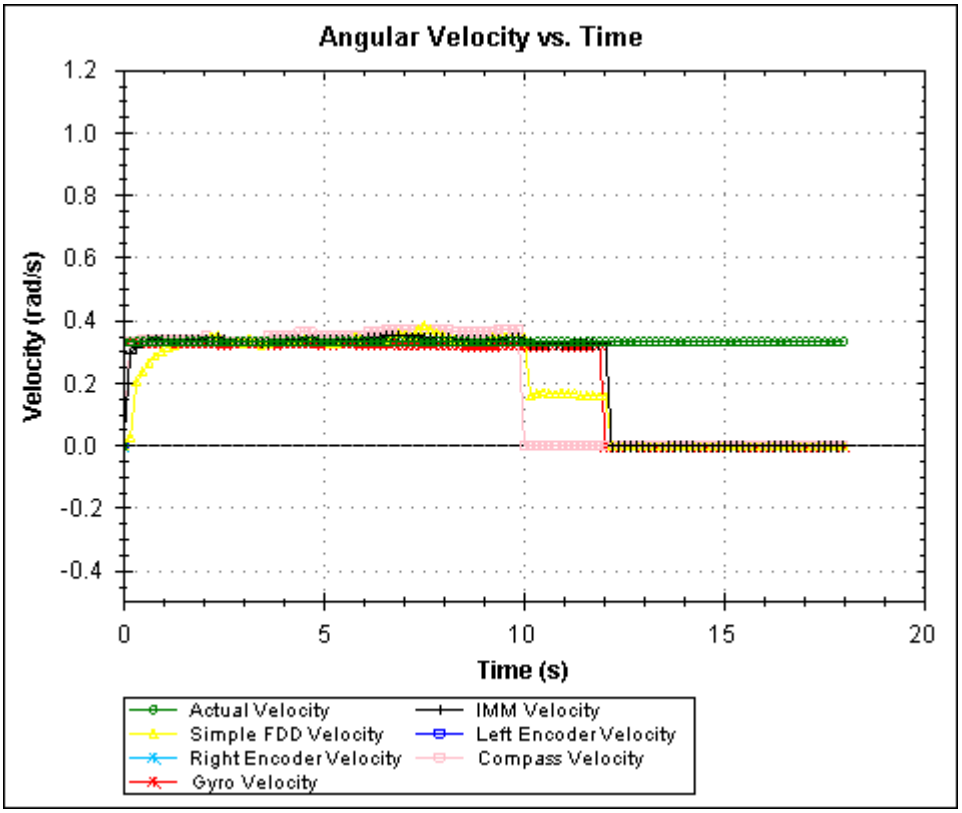

Figure 4.26: Angular velocity, circular path, compass fault, and gyro fault

5 Conclusion

Overall, the simulation application provided an intuitive, easy-to-use platform for comparing and analyzing sensor fault detection algorithms. The results provided in the previous chapter demonstrated the utility of the tool, by graphically displaying the outputs of the robot, the sensors, and the fault detectors.

This work could easily be extended to include more sensors, different sensors, and other fault detection algorithms. In fact, this simulation tool was designed with future expansion in mind.

Currently, the simulation dynamically loads any class in the Simulation.FaultDetectors namespace that implements the IFaultDetector interface. In the future, that namespace could be moved into its own assembly so that other people could easily write new fault detector classes without potentially affecting the main simulation logic.

Adding additional sensors could be accomplished in a similar manner by implementing the existing ISensor interface. The Simulation.Sensors namespace could be moved into a separate assembly so that each new sensor class could be added without potentially affecting the main simulation logic. However, the user interface would need to be modified slightly so that different combinations of sensors could be selected for simulation. That portion of the user interface would also need to construct itself dynamically, as well.

Bibliography

Bar-Shalom, Yaakov, Huimin Chen. "IMM Estimator with Out-of-Sequence Measurements." IEEE Transactions on Aerospace and Electronic Systems 41.1, January 2005: 90-98

Cain, Michael P. "Fusion of Data From Spatially Separated Sensors Using Riemannian Manifolds." Proceedings of SPIE 3067 Orlando, Florida, April 1997: 106-117

Demetriou, Michael. "Robust Adaptive Techniques for Sensor Fault Detection and Diagnosis." Proceedings of the IEEE Conference on Decision & Control 1 Tampa, Florida, December 1998: 1143-1148

Dima, Cristian S., Nicolas Vandapel, and Martial Hebert. "Sensor and Classifier Fusion for Outdoor Obstacle Detection: an Application of Data Fusion to Autonomous Off-Road Navigation." Proceedings of the Applied Imagery Pattern Recognition Workshop October 2003: 255-262

Goel, Puneet, Goksel Dedeoglu, Stergios I. Roumeliotis, Gaurav S. Sukhatme. "Fault Detection and Identification in a Mobile Robot Using Multiple Model Estimation and Neural Network." Proceedings of the IEEE International Conference on Robotics & Automation 3 San Francisco, CA, April 2000: 2302-2309

Hashimoto, Masafumi, Hiroyuki Kawashima, Takashi Nakagami, and Fuminori Oba. "Sensor Fault Detection and Identification in Dead-Reckoning System of Mobile Robot: Interacting Multiple Model Approach." Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems Maui, Hawaii, Nov. 2001: 1321-1326

Heckerman, David. “A Tutorial on Learning With Bayesian Networks.” Microsoft Research Technical Report November 1996. Microsoft. 30 June 2007. <ftp://ftp.research.microsoft.com/pub/tr/tr-95-06.pdf>

Jensen, Finn V. Bayesian Networks and Decision Graphs. Springer, 2001

Johnston, Leigh A., Vikram Krishnamurthy. “An Improvement to the Interacting Multiple Model (IMM) Algorithm.” IEEE Transactions on Signal Processing 49.12, December 2001: 2909-2923

Kirubarajan, T., Y. Bar-Shalom. “Kalman Filter Versus IMM Estimator: When Do We Need the Latter?” IEEE Transactions on Aerospace and Electronic Systems 39.4, October 2003: 1452-1457

Kolokotsa, Dionissia, Anastasios Pouliezos, George Stavrakakis. “Sensor Fault Detection in Building Energy Management Systems.” Proceedings of the International Conference on Technology and Automation October 2005. Technological University of Crete. 10 June 2006 <http://pouliezos.dpem.tuc.gr/pdf/icta_05_53107.PDF>

Kundur, Deepa, Dimitrios Hatzinakos, and Henry Leung. “A Novel Approach to Multispectral Blind Image Fusion.” Proceedings of SPIE 3067 Orlando, Florida, April 1997: 83-93

Maybeck, Peter S. Stochastic Models, Estimation, and Control, Volume 1. Academic Press, 1979

Mazor, E., A. Averbuch, Y. Bar-Shalom, J. Dayan. “Interacting Multiple Model Methods in Target Tracking: A Survey.” IEEE Transactions on Aerospace and Electronic Systems 34.1, January 1998: 103-123

Patwari, Neal, Alfred O. Hero, Josh Ash, Randolph L. Moses, Spyros Kyperountas, Neiyer S. Correal. "It Takes a Network: Cooperative Geolocation of Wireless Sensors." Ohio State University, January 2005

Rao, Nageswara S. V. "Fusion Rule Estimation Using Vector Space Methods." Proceedings of SPIE 3067 Orlando, Florida, April 1997: 130-135

Roumeliotis, Stergios I., Gaurav S. Sukhatme, and George A. Bekey. "Sensor Fault Detection and Identification in a Mobile Robot." Proceedings of the IEEE/RSJ Intl. Conference on Intelligent Robots and Systems 3 Victoria, B.C., Canada, October 1998: 1383-1388

Tebo, Albert. "Sensor Fusion Employs a Variety of Architecture, Algorithms, and Applications." OE Reports 164, August 1997

Vasquez, Juan R., Peter S. Maybeck. "Enhanced Motion and Sizing of Bank in Moving-Bank MMAE." Proceedings of the American Control Conference 40.3 June 1999: 770-779

Visinsky, Monica L.. Fault Detection and Fault Tolerance Methods for Robotics. MS thesis. Rice University, 1991

Welch, Greg, Gary Bishop. "An Introduction to the Kalman Filter." University of North Carolina at Chapel Hill, April 2004.

Williams, Arnold, Peter Pachowicz, and Larry Ronk. "A Novel Architecture for Expert Assisted Decision Level Fusion." Proceedings of SPIE 3067 Orlando, Florida, April 1997: 2-13

Yu, D.L., J.B. Gomm, D. Williams. "Sensor fault diagnosis in a chemical process via RBF neural networks." Control Engineering Practice 7.1 January 1999: 49-55

Zhang, Youmin, X. Rong Li. “Detection and Diagnosis of Sensor and Actuator Failures Using IMM Estimator.” IEEE Transactions on Aerospace and Electronic Systems 34.4, October 1998: 1293-1313

Zhou, Yifeng, Henry Leung. “A Linearly Constrained Least Squares Approach for Multisensor Data Fusion.” Proceedings of SPIE 3067 Orlando, Florida, April 1997: 118-129

Appendix

Class Diagrams

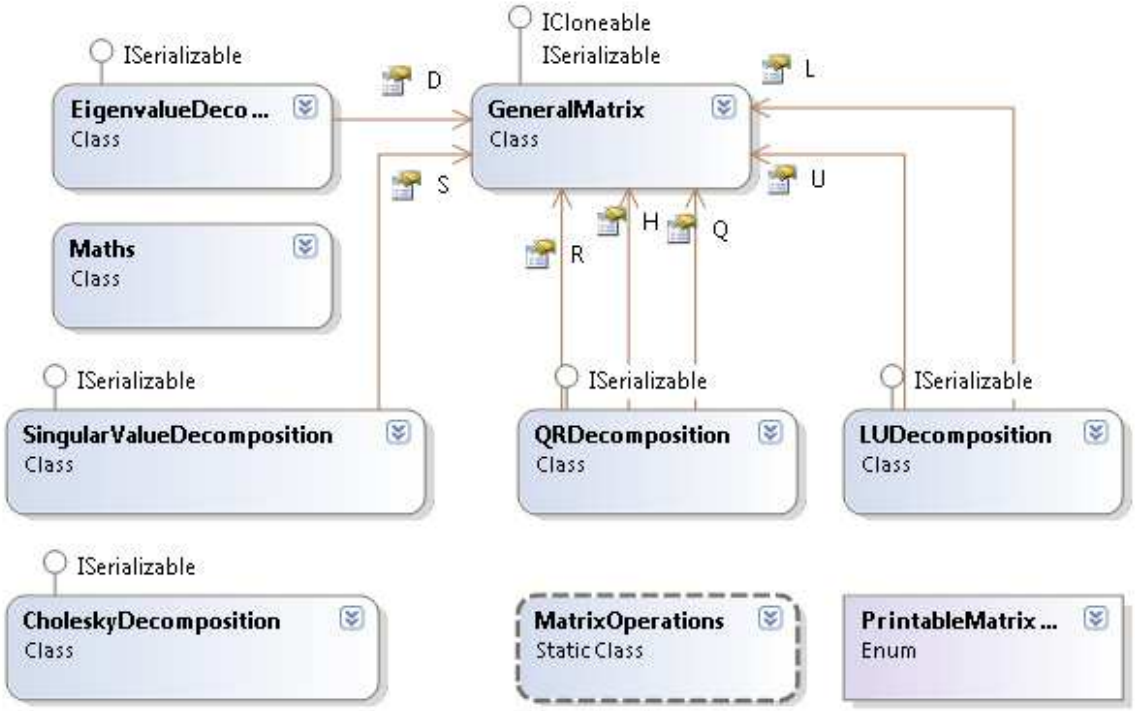

Figure 0.1: MatrixLibrary.dll class diagram

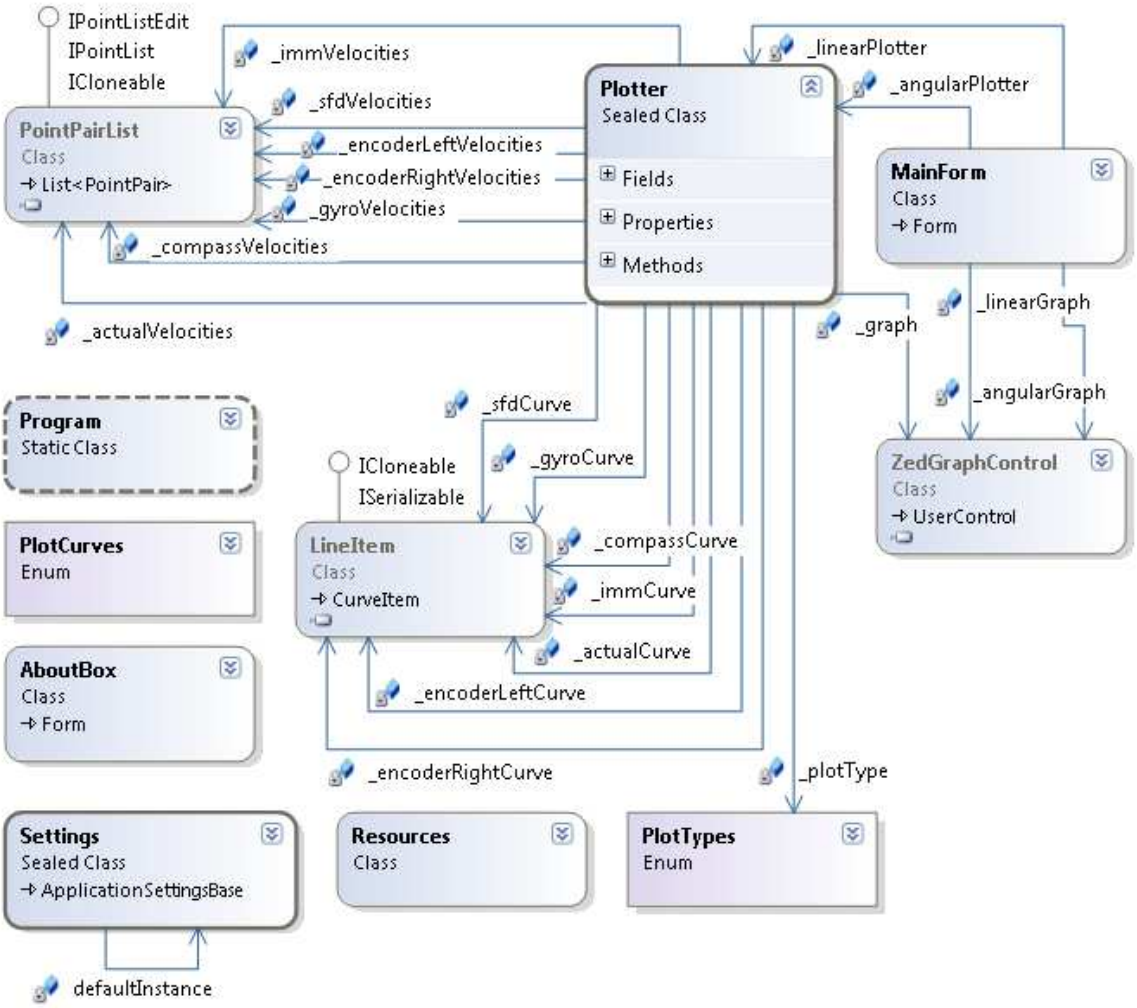

Figure 0.2: SimulationGUI.exe class diagram

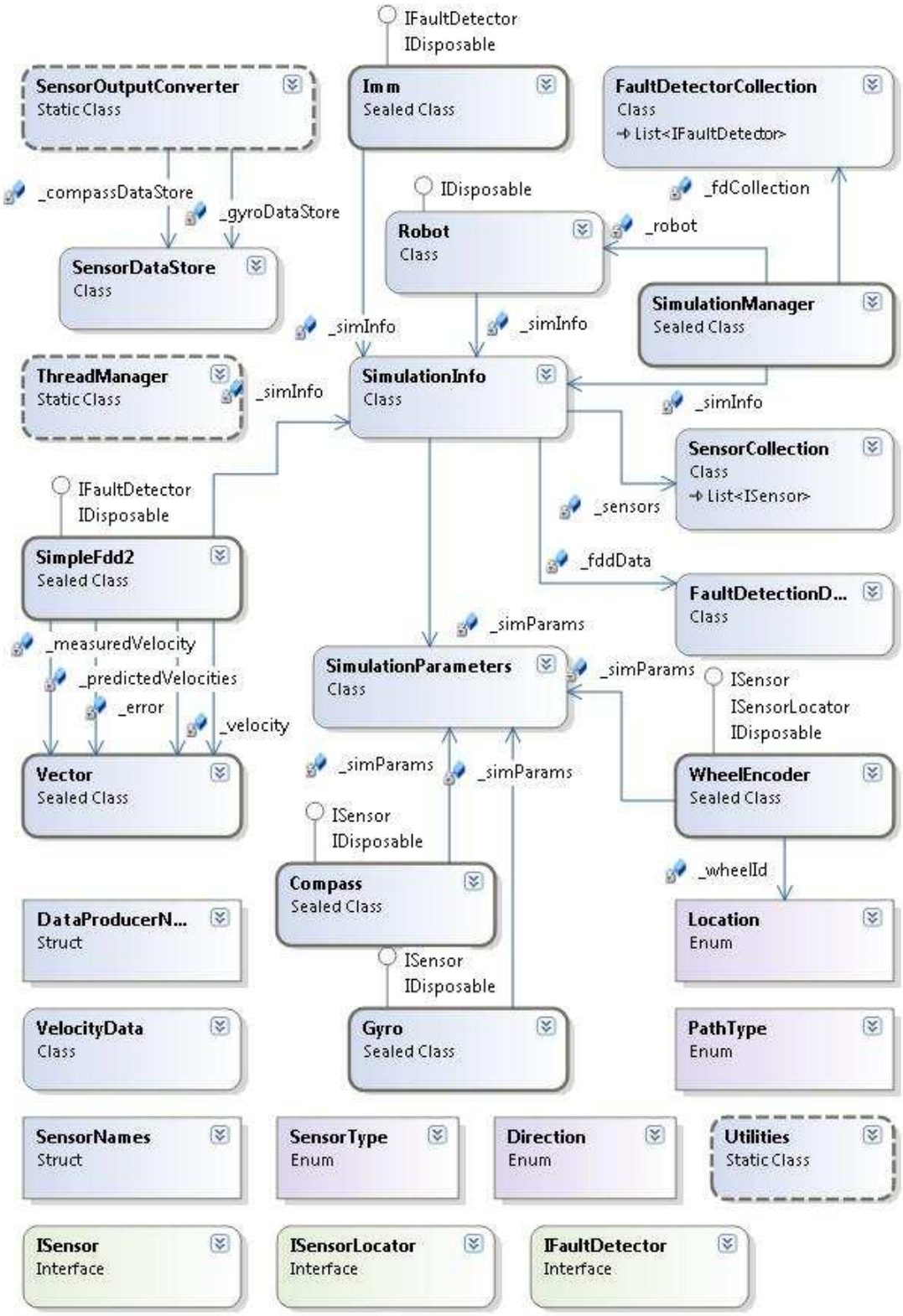

Figure 0.3: Simulation.dll class diagram