



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**REAL-TIME IMPLEMENTATION OF AN ASYNCHRONOUS
VISION-BASED TARGET TRACKING SYSTEM IN AN
UNMANNED AERIAL VEHICLE**

by

Michael A. Schenk

June 2007

Thesis Advisor:
Second Reader:

Isaac I. Kaminer
Vladimir N. Dobrokhodov

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 2007	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Real-Time Implementation of an Asynchronous Vision-Based Target Tracking System for an Unmanned Aerial Vehicle.			5. FUNDING NUMBERS	
6. AUTHOR(S) Michael A. Schenk				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) Currently, small unmanned aerial vehicles developed by NPS have been able to locate and track stationary and moving targets on the ground. New methods of continuous target tracking are always being developed to improve speed and accuracy, ultimately aiding the user of the system. This thesis describes one such method, utilizing an open loop filter as well as an external correction source: Perspective View Nascent Technologies (PVNT). While the PVNT correction can theoretically improve the accuracy from 20-30 meters to 1-2 meters, it does have a disadvantage in that the target position updates are delayed anywhere from 1-10 seconds. In order to account for the delay, an asynchronous filter is used to update the target position data given the external position correction from PVNT. Two cases have been tested including the general filter and one that utilizes a road model in the calculations. While an earlier thesis developed the basic simulation for the system, this thesis discusses improvements and corrections to the simulation model as well as the necessary steps for real-time implementation.				
14. SUBJECT TERMS Unmanned Aerial Vehicle, Asynchronous Filter, Perspective View Nascent Technologies, Vision-Based Target Tracking			15. NUMBER OF PAGES 157	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**REAL-TIME IMPLEMENTATION OF AN ASYNCHRONOUS
VISION-BASED TARGET TRACKING SYSTEM IN AN
UNMANNED AERIAL VEHICLE**

Michael A. Schenk
Ensign, United States Navy
B.S., United States Naval Academy, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
June 2007**

Author: Michael A. Schenk

Approved by: Isaac I. Kaminer
Thesis Advisor

Vladimir N. Dobrokhodov
Second Reader

Anthony J. Healey
Chairman, Department of Mechanical and Astronautical
Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Currently, small unmanned aerial vehicles developed by NPS have been able to locate and track stationary and moving targets on the ground. New methods of continuous target tracking are always being developed to improve speed and accuracy, ultimately aiding the user of the system. This thesis describes one such method, utilizing an open loop filter as well as an external correction source: Perspective View Nascent Technologies (PVNT). While the PVNT correction can theoretically improve the accuracy from 20-30 meters to 1-2 meters, it does have a disadvantage in that the target position updates are delayed anywhere from 1-10 seconds. In order to account for the delay, an asynchronous filter is used to update the target position data given the external position correction from PVNT. Two cases have been tested including the general filter and one that utilizes a road model in the calculations. While an earlier thesis developed the basic simulation for the system, this thesis discusses improvements and corrections to the simulation model as well as the necessary steps for real-time implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	OVERVIEW	1
B.	BACKGROUND	1
1.	Perspective View Nascent Technologies (PVNT)	1
2.	Asynchronous Constant Gain Kalman Filter	2
C.	THESIS DESCRIPTION	3
II.	SYSTEM STRUCTURE	5
A.	SEQUENCE OF OPERATION.....	5
1.	Linear Filter with PVNT Update	5
B.	GENERAL AND ROAD FOLLOWING FILTER DIFFERENCES.....	6
C.	OBJECTIVES	7
III.	MODELING.....	9
A.	NON REAL-TIME MODELING.....	9
1.	General Filter	9
2.	Road Following Filter with Separate Model File Integration.....	9
a.	<i>True Target Motion with Road Following Characteristics ...</i>	<i>10</i>
b.	<i>PVNT Model.....</i>	<i>12</i>
c.	<i>Optimization Function.....</i>	<i>12</i>
d.	<i>Asynchronous Filter</i>	<i>13</i>
e.	<i>Open-Loop Filter.....</i>	<i>15</i>
3.	Road Following Filter with Numerical Euler Integration	15
a.	<i>Integration Equations.....</i>	<i>16</i>
B.	REAL-TIME MODELING.....	17
1.	Problems	17
a.	<i>Data Storage.....</i>	<i>17</i>
b.	<i>The S-Function</i>	<i>18</i>
c.	<i>Arrays vs. Buffers.....</i>	<i>18</i>
2.	General Filter	19
a.	<i>The True Target Model and PVNT Update Generator</i>	<i>20</i>
b.	<i>The Open-Loop Filter</i>	<i>22</i>
c.	<i>Overall Real-Time Design and Function with S-Function Block.....</i>	<i>23</i>
3.	Road Following Filter	24
a.	<i>The True Target Model and PVNT Update Generator</i>	<i>24</i>
b.	<i>The Open Loop Filter</i>	<i>25</i>
c.	<i>Overall Real-Time Design and Function with S-Function Block.....</i>	<i>26</i>
IV.	SIMULATION AND RESULTS	29
A.	SIMULATION	29
1.	Road Models	29

a.	<i>Third Order Road Model</i>	29
b.	<i>Circular Road Model</i>	30
2.	Simulation Parameters	32
a.	<i>Simulation Time</i>	32
b.	<i>Sample Time</i>	32
c.	<i>Asynchronous Kalman Filter Gains</i>	32
d.	<i>Reference Frame</i>	33
e.	<i>PVNT Parameters</i>	33
B.	RESULTS	34
1.	Non Real-Time Models	34
a.	<i>Road Following Model with Separate Simulink Model Integration</i>	35
b.	<i>Road Following Model with Numerical Forward Euler Integration</i>	37
2.	Real-Time Models	41
a.	<i>General Filter</i>	41
b.	<i>Road Following Filter</i>	62
c.	<i>Additional Road Models and Worst Case Scenarios</i>	85
V.	CONCLUSION AND RECOMMENDATIONS	95
A.	CONCLUSIONS	95
B.	RECOMMENDATIONS	95
	APPENDIX	97
A.	GENERAL FILTER	97
1.	Manual	97
2.	Code	105
B.	ROAD FOLLOWING FILTER S-FUNCTION	117
1.	Manual	117
2.	Code	127
	LIST OF REFERENCES	139
	INITIAL DISTRIBUTION LIST	141

LIST OF FIGURES

Figure 1.	Linear filtering with PVNT update	5
Figure 2.	PVNT processing and Kalman filtering with respect to time	6
Figure 3.	Target Estimation Comparison for Open Loop and Road Following Filters.....	7
Figure 4.	Road following asynchronous filter.....	10
Figure 5.	True target position and velocity generation with road following.....	10
Figure 6.	Simulated Road Profile [After Ref. 6]	11
Figure 7.	PVNT update generator	12
Figure 8.	Asynchronous filter – Subsystem 1	13
Figure 9.	Asynchronous filter – Subsystem 2	13
Figure 10.	Separate Simulink model containing asynchronous filter	14
Figure 11.	Single integration open-loop filter	15
Figure 12.	Asynchronous filter model with state variables.....	16
Figure 13.	True target model and PVNT update generator for general filter.....	20
Figure 14.	PVNT update signal subsystem	21
Figure 15.	General filter real-time model.....	23
Figure 16.	Open loop filter for the road following model.....	25
Figure 17.	Road following filter real-time model	26
Figure 18.	Third order road equation in Simulink subsystem for real-time simulation	30
Figure 19.	Circular road equation in Simulink subsystem for real-time simulation	31
Figure 20.	Circular road model	32
Figure 21.	Simulated pseudo-random PVNT update delay.....	33
Figure 22.	Comparison of actual vs. estimated target position – Simulink integration	35
Figure 23.	Comparison of actual vs. estimated target velocity – Simulink integration	36
Figure 24.	Comparison of actual vs. estimated ρ value – Simulink integration.....	37
Figure 25.	Comparison of actual vs. estimated target position – Euler integration	38
Figure 26.	Comparison of actual vs. estimated target velocity – Euler integration	39
Figure 27.	Comparison of actual vs. estimated ρ velocity – Euler integration	40
Figure 28.	General filter position comparison plot – Third order road model – Ideal conditions.....	41
Figure 29.	General filter position error vs. time – Third order road model	42
Figure 30.	General filter velocity comparison plot – Third order road model – Ideal conditions.....	43
Figure 31.	General filter velocity error vs. time – Third order road model	43
Figure 32.	General filter position comparison plot – Circular road model – Ideal conditions.....	44
Figure 33.	General filter position error vs. time – Circular road model.....	45
Figure 34.	General filter velocity comparison plot – Circular road model – Ideal conditions.....	46
Figure 35.	General filter velocity error vs. time – Circular road model.....	47
Figure 36.	General filter position comparison plot – Third order road model – 5 second PVNT delay	48

Figure 37.	General filter velocity comparison plot – Third order road model – 5 second PVNT delay	49
Figure 38.	General filter position comparison plot – Third order road model – 10 second PVNT delay	49
Figure 39.	General filter velocity comparison plot – Third order road model – 10 second PVNT delay	50
Figure 40.	General filter position comparison plot – Circular road model – 5 second PVNT delay	51
Figure 41.	General filter position error plot – Circular road model – 5 second PVNT delay	52
Figure 42.	General filter velocity comparison plot – circular road model – 5 second PVNT delay	53
Figure 43.	General filter position error plot – Circular road model – 10 second PVNT delay	54
Figure 44.	General filter velocity comparison plot – Circular road model – 10 second PVNT delay	54
Figure 45.	General filter position comparison plot – Third order road model – ± 5 m PVNT noise.....	55
Figure 46.	General filter velocity comparison plot – Third order road model – ± 5 m PVNT noise.....	56
Figure 47.	General filter position comparison plot – Third order road model – ± 10 m PVNT noise.....	57
Figure 48.	General filter velocity comparison plot – Third order road model – ± 10 m PVNT noise.....	58
Figure 49.	General filter position comparison plot – Circular road model – ± 5 m PVNT noise.....	59
Figure 50.	General filter position error plot – Circular road model – ± 5 meter PVNT noise	59
Figure 51.	General filter velocity comparison plot – Circular road model – ± 5 meter PVNT noise.....	60
Figure 52.	General filter position error plot – Circular road model – ± 10 meter PVNT noise.....	61
Figure 53.	General filter velocity comparison plot – Circular road model – ± 10 meter PVNT noise.....	62
Figure 54.	Road following filter position comparison plot – Third order road model – Ideal conditions.....	63
Figure 55.	Road following filter position error plot – Third order road model – Ideal conditions.....	64
Figure 56.	Road following filter velocity comparison plot – Third order road model – Ideal conditions.....	64
Figure 57.	Road following filter velocity error plot – Third order road model – Ideal conditions.....	65
Figure 58.	Road following filter ρ comparison plot – Third order road model – Ideal conditions.....	66

Figure 59.	Road following filter position comparison plot – Circular road model – Ideal conditions.....	66
Figure 60.	Road following filter position error plot – Circular road model – Ideal conditions.....	67
Figure 61.	Road following filter velocity comparison plot – Circular road model – Ideal conditions.....	68
Figure 62.	Road following filter velocity error plot – Circular road model – Ideal conditions.....	69
Figure 63.	Road following filter ρ comparison plot – Circular road model – Ideal conditions.....	69
Figure 64.	Road following filter position comparison plot – Third order road model – 5 second PVNT delay	70
Figure 65.	Road following filter velocity comparison plot – Third order road model – 5 second PVNT delay	71
Figure 66.	Road following filter ρ comparison plot – Third order road model – 5 second PVNT delay	71
Figure 67.	Road following filter position comparison plot – Third order road model – 10 second PVNT delay	72
Figure 68.	Road following filter velocity comparison plot – Third order road model – 10 second PVNT delay	73
Figure 69.	Road following filter ρ comparison plot – Third order road model – 10 second PVNT delay	73
Figure 70.	Road following filter position error plot – Circular road model – 5 second PVNT delay	74
Figure 71.	Road following filter velocity comparison plot – Circular road model – 5 second PVNT delay	75
Figure 72.	Road following filter position error plot – Circular road model – 10 second PVNT delay	76
Figure 73.	Road following filter velocity comparison plot – Circular road model – 10 second PVNT delay	77
Figure 74.	Road following filter position comparison plot – Third order road model – ± 5 m PVNT noise.....	78
Figure 75.	Road following filter velocity comparison plot – Third order road model – ± 5 m PVNT noise.....	78
Figure 76.	Road following filter ρ comparison plot – Circular road model – ± 5 m PVNT noise.....	79
Figure 77.	Road following filter position comparison plot – Third order road model – ± 10 m PVNT noise.....	80
Figure 78.	Road following filter velocity comparison plot – Third order road model – ± 10 m PVNT noise.....	80
Figure 79.	Road following filter ρ comparison plot – Circular road model – ± 10 m PVNT noise.....	81
Figure 80.	Road following filter position error plot – Circular road model – ± 5 m PVNT noise.....	82

Figure 81.	Road following filter velocity comparison plot – Circular road model – ± 5 m PVNT noise.....	83
Figure 82.	Road following filter position error plot – Circular road model – ± 10 m PVNT noise.....	84
Figure 83.	Road following filter velocity comparison plot – Circular road model – ± 10 m PVNT noise.....	85
Figure 84.	Road model comparison	86
Figure 85.	Filter position estimation comparison – Model 1	87
Figure 86.	Filter estimation error comparison – Model 1	88
Figure 87.	Filter position estimation comparison – Model 2	88
Figure 88.	Filter estimation error comparison – Model 2	89
Figure 89.	Filter position estimation comparison – Model 3	90
Figure 90.	Filter estimation error comparison – Model 3	90
Figure 91.	Filter position estimation comparison – Model 4	91
Figure 92.	Filter estimation error comparison – Model 4	92
Figure 93.	Position comparison plot – Worst case scenario.....	93
Figure 94.	Filter estimation error comparison – Worst case scenario.....	93

ACKNOWLEDGMENTS

The author would like to thank Professor Isaac I. Kaminer for his efforts in explaining the core concepts behind the vision-based target tracking system as well as aiding in the familiarization process with previous work on the subject matter. The author would also like to thank Dr. Eng. Ioannis Kitsios for the countless meetings spent helping to develop the real-time models and orientation to the ways of the S-function.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

The goal of the work in this thesis is to contribute to continuous improvements that are being made in the area of vision-based target tracking and motion estimation. Changes to the current system allow the entire process to be faster, more accurate, and more user-friendly. Improvements to the technology can be simulated using computer programs such as MATLAB and Simulink, implemented, and then field tested during scenarios run by NPS during the Tactical Network Topology (TNT) sessions. Ultimately, the systems developed here can make their way to military use in surveillance and reconnaissance missions. Concepts examined in this thesis include the development and implementation non real-time and real-time target motion estimation systems as well as asynchronous target tracking filters with and without road following capabilities.

B. BACKGROUND

Several important tools are discussed that play prominent roles in the developed systems in this thesis. The PVNT position update system is described first, followed by the background on the asynchronous constant gain Kalman filter.

1. Perspective View Nascent Technologies (PVNT)

One of the problems with incorporating detailed terrain maps into real-time systems is the large amount of required data storage and equally large amount of necessary computing power needed to deal with the loading and retrieval of map sections. Developed by Dr. Wolfgang Baer, the PVNT system offers a low-cost alternative available on a personal computer. The PVNT system begins with terrain data collected by the National Imagery and Mapping Agency (NIMA) and contains tools that allow updates to be included from local measurement devices and other sensors [1]. The inclusion of this new data results in a more accurate terrain mapping system than can be more efficiently updated to reflect terrain variations rather than creating entirely new maps.

Another major advantage PVNT has over other scene-visualization programs is that the terrain data are stored using raster formats (pixels) instead of using a polygon database [1]. This makes implementation of the system using PVNT combined with remote sensor arrays in real-time much more practical.

Tests conducted at Camp Roberts, CA, depict how PVNT works hand-in-hand with a vision-based target tracking system. Initially, the target is acquired and the gimbal-mounted camera passes data to image processing software. Then, open-loop, non-linear filters are able to estimate the target position and resulting velocity. After around 20 seconds of tracking, the accuracy for this portion of the system is within about 10-20 meters. The PVNT software then compares data coming in from GPS, camera angle values, and the images from the UAV camera to the terrain database for the area. Since the accuracy of the database has roughly a one meter resolution, the accuracy of the position update from PVNT can be ten times more accurate than the non-linear filter estimation. However, because of the multiple data inputs and necessary image comparison between the camera and terrain database, the required processing time results in a delay up to ten seconds before the position update is delivered to the system [6].

2. Asynchronous Constant Gain Kalman Filter

It is in target motion estimation that the Kalman filter can be employed. One of the reasons that the Kalman filter works well with target tracking applications is its ability to compare and integrate data from multiple sensors (such as a position update with estimated target velocity and estimated position) to give the most accurate result. However, standard Kalman filters are hindered by the fact that they must have evenly-spaced data inputs and updates for maximum effectiveness. The filter runs into accuracy problems when data arrives at different sampling rates or delayed times.

The asynchronous constant gain Kalman filter was developed because of the need for an accurate estimation tool in a system with delayed data inputs. It is the preferred filter to ensure better system robustness and overall result accuracy because the asynchronous version of the filter is able to accept out of synchronization data entries from sensors. Thus, the asynchronous constant gain Kalman filter is a better match for

this target tracking system since the data from the PVNT update is delayed anywhere from one to ten seconds before being entered into the filter [6].

C. THESIS DESCRIPTION

Chapter I presents a general overview of the work of the thesis with respect to UAV target tracking capabilities as well as background for two of the main tools utilized in the thesis: the PVNT update system and asynchronous constant gain Kalman filter. The next chapter will outline, step-by-step, the process that takes place during target motion estimation with and without position updates. The chapter also discusses the general and road following filters; the two different styles of filters that are employed in the real- and non real-time models. Chapter III will briefly review the current non real-time general filter model for target tracking presented in an earlier thesis and then develop a road following version of the model for non real-time simulation in Simulink. Additionally, the chapter will discuss the steps needed to convert the non real-time models into real-time models along with the actual modeling of the real-time general and road following filters in Simulink. Chapter IV will go over the system parameters and actual simulation of the non real-time and real-time models. While the non real-time road following model will be tested using a single road model to ensure proper function, the real-time general and road following models will be tested with numerous simulated roads and varying input errors. All of the necessary results will follow in the chapter along with explanations for the response of the system to different scenarios. Finally, Chapter V will present the conclusions from the thesis results and recommendations for future work in the field of study.

THIS PAGE INTENTIONALLY LEFT BLANK

II. SYSTEM STRUCTURE

This chapter provides flow charts and diagrams describing the processes that take place in the filter operations with and without PVNT updates. The final section also explains the objectives for the thesis.

A. SEQUENCE OF OPERATION

The first task is to organize the order of the processes within the target tracking system. Then, the improved system with the general linear filter must be altered to incorporate the PVNT position updates in parallel with the normal operation.

1. Linear Filter with PVNT Update

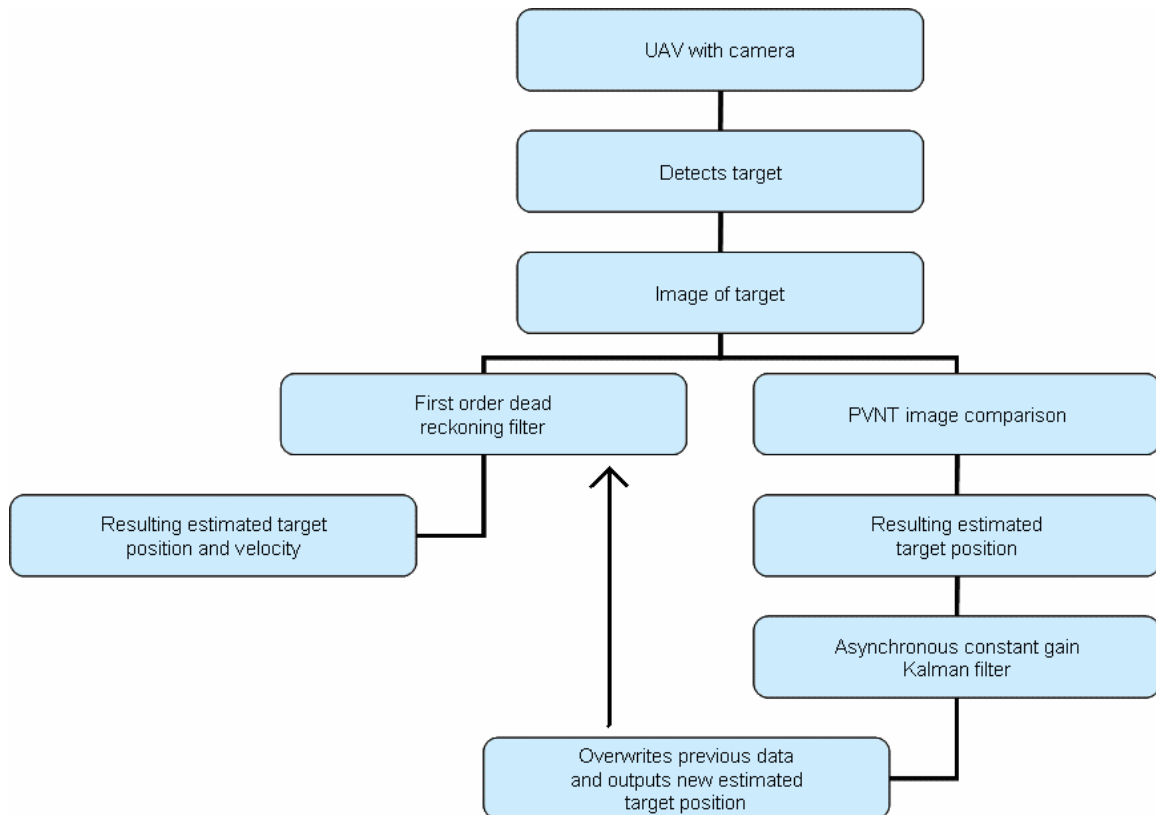


Figure 1. Linear filtering with PVNT update

Figure 1 shows the asynchronous filtering system with the addition of the PVNT position update. After the necessary image processing by the PVNT software, the new estimated target position is fed into the asynchronous constant gain Kalman filter. The asynchronous filter then performs the required calculations, rewriting over the data previously stored during the delay, and outputs a new estimated target position to the original non-linear filter. Figure 2 below shows how the PVNT processing and asynchronous constant gain Kalman filter relates to the time interval for the system.

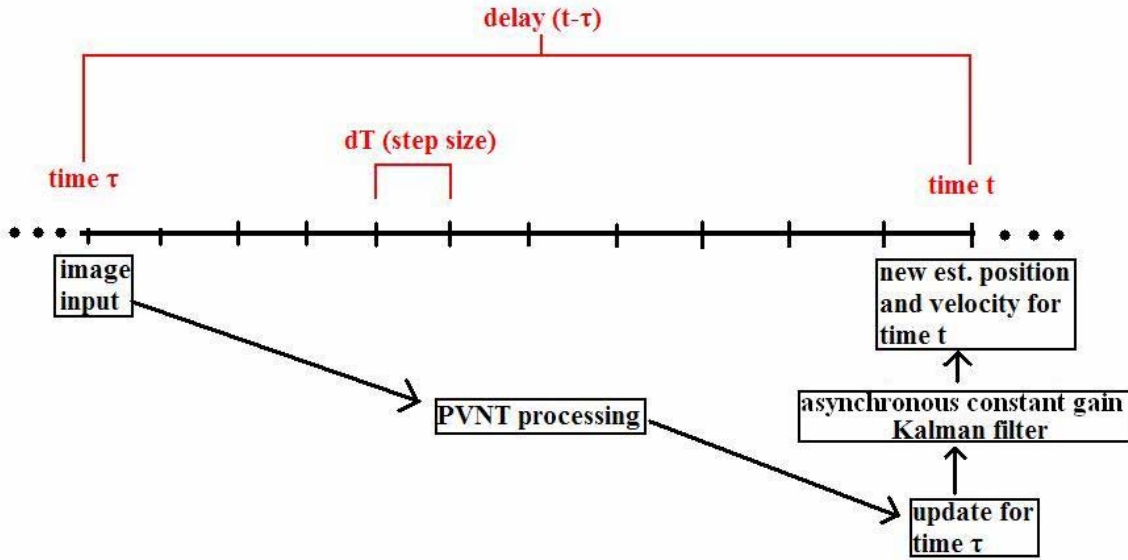


Figure 2. PVNT processing and Kalman filtering with respect to time

B. GENERAL AND ROAD FOLLOWING FILTER DIFFERENCES

The two different style filters tested in this thesis are the general and road following filters. Both filters receive PVNT updates and perform target motion estimation along road models during the simulations. The difference, however, is that the road following filter uses the road model equations in the target motion estimation calculations while the general filter does not. This allows the x, y, z coordinates used by the general filter model to be simplified in the road following model by a road parameter, ρ . This concept is discussed in greater detail in Chapter III.

The figure below depicts the effects of the road following parameter by comparing the position estimates of an open loop (OL) filter and an asynchronous road following filter (AF) with PVNT along a sample road profile. The filter estimates are identical during straight portions of the road, but the road following filter provides much better position estimates during areas of greater curvature. The position error is due to velocity estimation error, but the estimate still lies along the road profile.

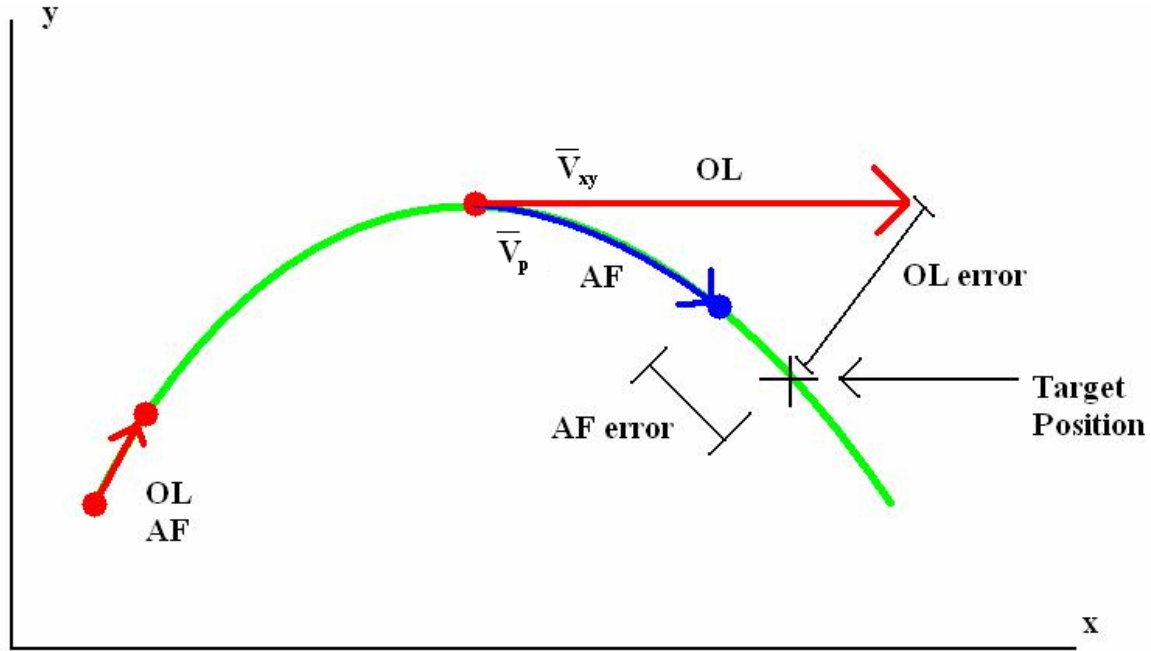


Figure 3. Target Estimation Comparison for Open Loop and Road Following Filters

C. OBJECTIVES

While the basic non real-time filter design is discussed in the master's thesis in Reference 6, numerous improvements and corrections were needed to make the non real-time road following filter perform correctly. The generation and storage of the PVNT update will be changed and the asynchronous constant gain Kalman filter will be modeled in Simulink. Additionally, the non real-time general and road following filter models will be converted into systems capable of real-time implementation. The two real-time filter systems will also be extensively simulated and the results analyzed to determine conditions of peak and poor performance.

THIS PAGE INTENTIONALLY LEFT BLANK

III. MODELING

This chapter details the modeling process for the non real-time and real-time systems. Included in the sections are models for the general and road following filters for each style system as well as development of the road equations. The process of converting the non real-time models into models that can be implemented in real-time systems is also cited with major focus placed on the S-function and its capabilities.

A. NON REAL-TIME MODELING

Before models can be implemented in a real-time system, non real-time models had to be produced. These non real-time models serve as a starting point for the development of the real-time filter.

1. General Filter

The general asynchronous filter and MATLAB code is found in Reference 6 and models the general target tracking filter incorporating a delayed PVNT update. Many of the components for the road following version of the filter are similar and will be discussed in the next section.

2. Road Following Filter with Separate Model File Integration

The main addition to the road following asynchronous filter is the parameter ρ , which defines the road along which the target is moving. Since the target plane is assumed to be two dimensional, ρ relates to the x and y coordinates of the target while z relates to local altitude. The road following asynchronous filter was corrected from Reference 6 due to errors in the MATLAB code concerning data storage and retrieval. Portions of the Simulink model were also adjusted to correctly generate and pass on the PVNT position update to later tasks in the system. The updated road following filter system is shown in Figure 4.

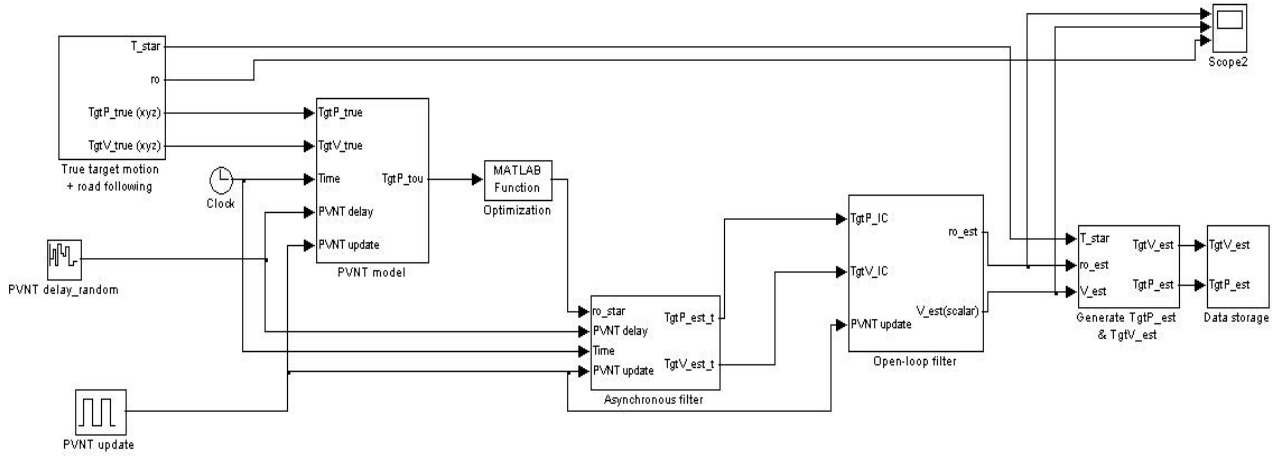


Figure 4. Road following asynchronous filter

Next, the Simulink diagram is broken down with a description of the function of each component in the overall system.

a. True Target Motion with Road Following Characteristics

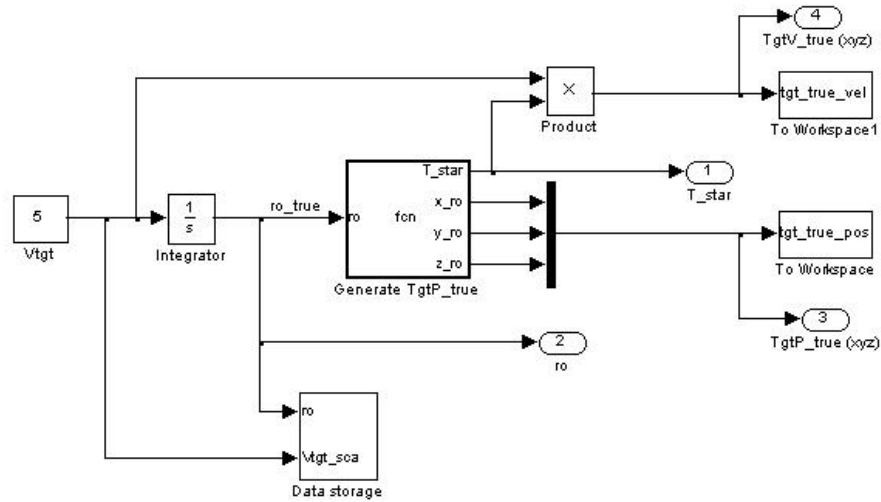


Figure 5. True target position and velocity generation with road following

The non real-time system begins with the target of constant velocity and integrating to calculate ρ . The new ρ value is then inputted into a MATLAB function block that calculates x , y , and z target position based on a predetermined equation for the road. For the preliminary simulation tests, the equation for the road based on the parameter ρ was taken from Reference 6:

$$\begin{aligned} x &= \rho \\ y &= 0.0000192\rho^3 - 0.025\rho^2 + 9.74\rho \\ z &= 0 \end{aligned} \quad (1) \quad [\text{Ref. 6}]$$

The above system of equations produces a road profile that is depicted in the next figure.

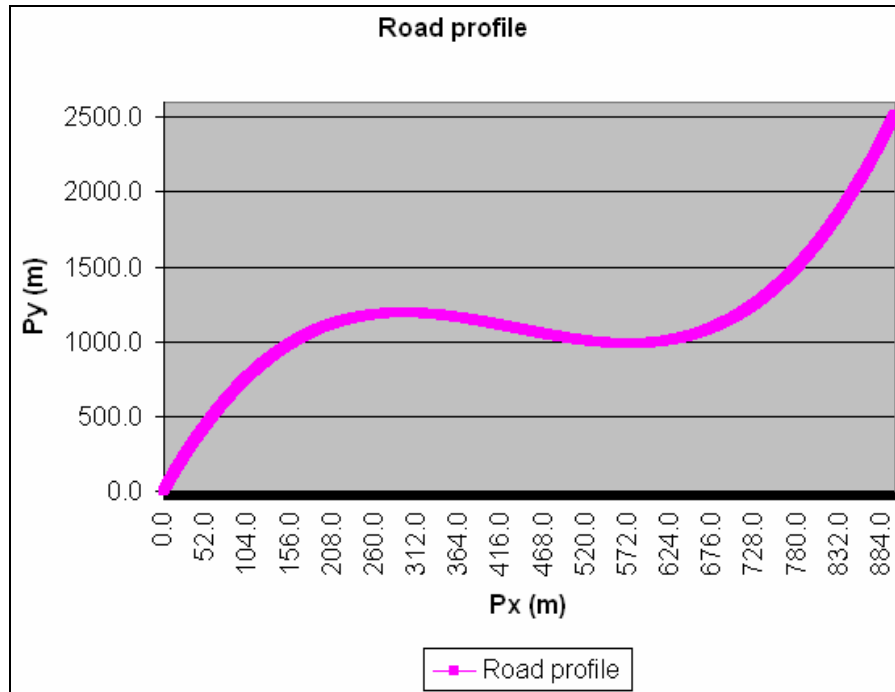


Figure 6. Simulated Road Profile [After Ref. 6]

This target data along with the ρ values are stored for later use as the simulated PVNT update. For the Simulink model, it is more practical to assume that the PVNT estimate, with one to two meter accuracy, can be simulated by taking an actual target position from the true target model at time τ instead of trying to run PVNT software.

b. PVNT Model

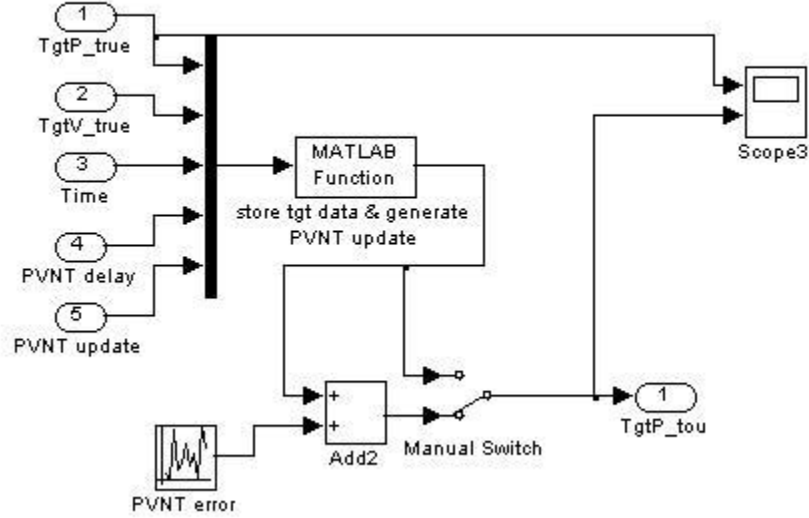


Figure 7. PVNT update generator

Figure 7 depicts the PVNT update block and its components for the non real-time road following model. The block receives and takes in the true target position and velocity from the target model along with the system time and a random PVNT update delay. The fifth input is an oscillating step signal that indicates when the PVNT update is active (signaling an update is ready to be sent to the asynchronous constant gain Kalman filter). Additionally, a PVNT input error can be included in the system to simulate the expected one to two meter accuracy of the device.

c. Optimization Function

The optimization function block is used to determine the parameter ρ , and was not changed from the earlier thesis. The method of optimization that is used in the non real-time road following filter simply finds ρ that minimizes the distance from the inputted PVNT position update to the road. Once the minimum distance over a maximum range is found, the corresponding ρ value for the x and y road coordinates is passed on to the asynchronous filter.

d. Asynchronous Filter

The asynchronous filter portion of the diagram has multiple triggered subsystems.

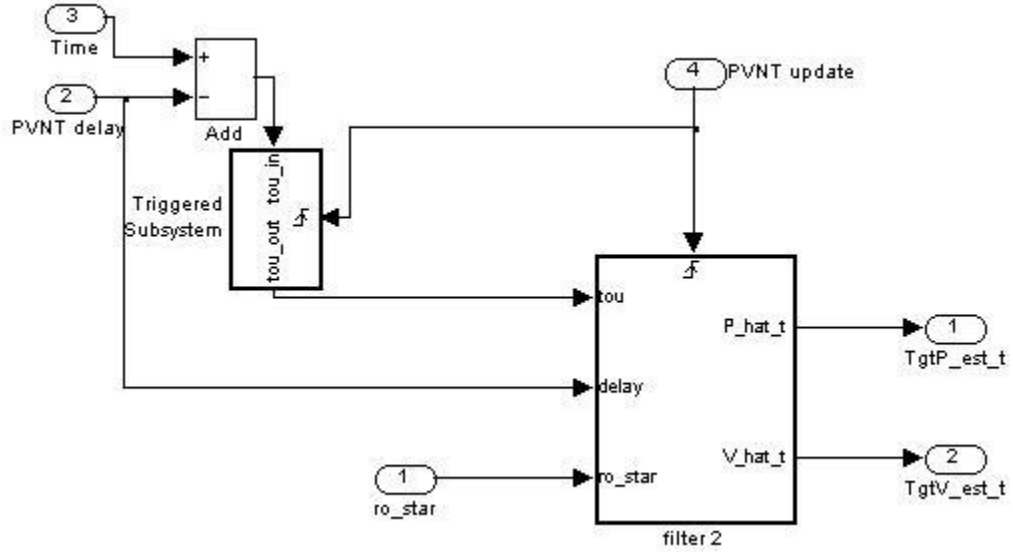


Figure 8. Asynchronous filter – Subsystem 1

The first subsystem calculates time τ once given the current system time t and the PVNT delay time. The time τ is then passed on to the second subsystem along with the PVNT delay and the PVNT update ρ^* .

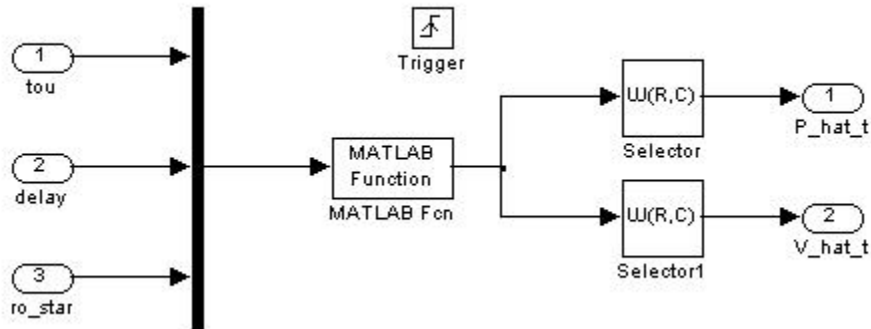


Figure 9. Asynchronous filter – Subsystem 2

Figure 9 shows the data being passed through a mux block and into a MATLAB function block, which outputs the estimated position ($\hat{p}$) and velocity ($\hat{v}$) data for the target at time t . It is important to note that this subsystem only runs when the PVNT update is present. The MATLAB function block refers to a function written in MATLAB code that actually performs the asynchronous double integration. The filter is actually contained in a separate Simulink model file and called by the MATLAB function.

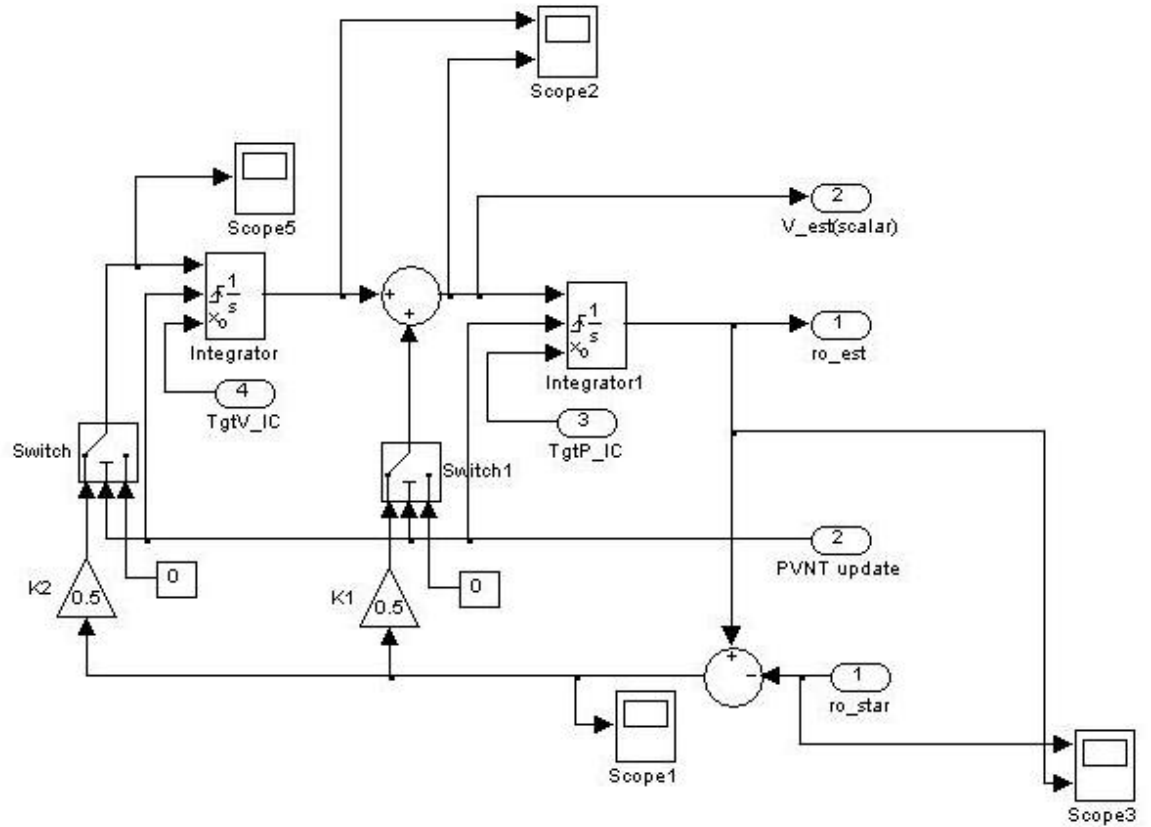


Figure 10. Separate Simulink model containing asynchronous filter

The filter operates independently of the time of the overall system, allowing it to be asynchronous. The MATLAB code stores the new estimated position and velocity data as the asynchronous filter integrates from time τ to t and outputs the last position and velocity values to initialize the open-loop filter. While the asynchronous filter for the target tracking system without road following capabilities must integrate

variables for velocity (in the x , y , and z directions) and position (x , y , z), the asynchronous filter for the road following system only integrates the scalar velocity and road parameter, ρ . Since the road equation is known before tracking begins, the ρ value is easily converted into x , y , and z coordinates.

e. Open-Loop Filter

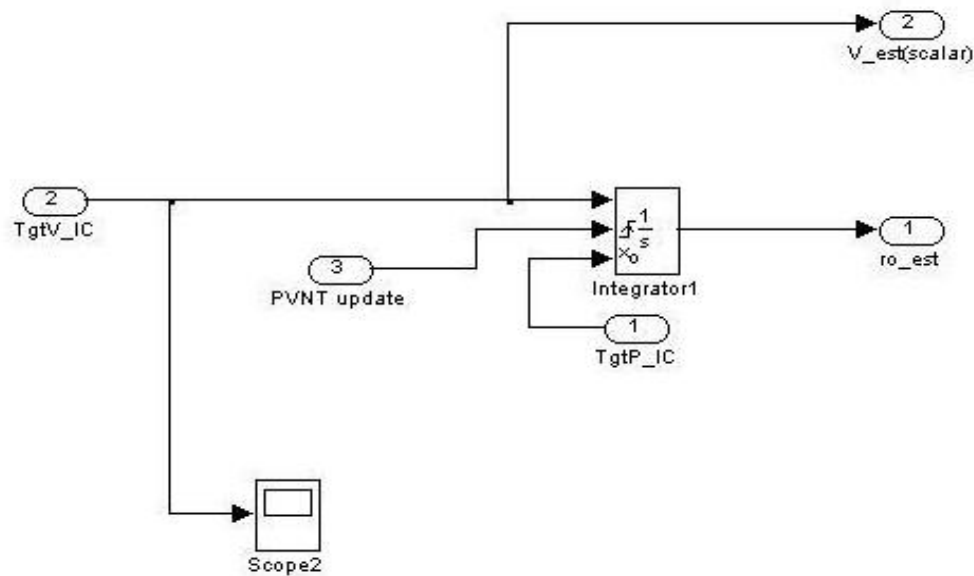


Figure 11. Single integration open-loop filter

The final step in the simulation is the open-loop filter subsystem. This block essentially performs a dead-reckoning position calculation based on the inputted estimated velocity and contains a single integrator that operates during the periods of time with no PVNT update.

3. Road Following Filter with Numerical Euler Integration

The main difference between the road following filter developed in Section 2 above and the one described in Section 3 of this chapter is the conversion of the Simulink model file containing the asynchronous filter in Figure 10 to a set of numerical equations. These equations are used for forward Euler integration, allowing the system to quickly determine position estimates from time τ to time t (see Figure 2). It is necessary to

implement the integration model in MATLAB code to decrease computation time during simulation. Additionally, a real-time system cannot operate properly using multiple Simulink models.

a. Integration Equations

In order to implement double integrator into MATLAB code, the diagram must be represented numerically. By taking the asynchronous filter model from Figure 10, removing the scope blocks, and adding state variables before and after the integrator blocks, a set of equations can be developed.

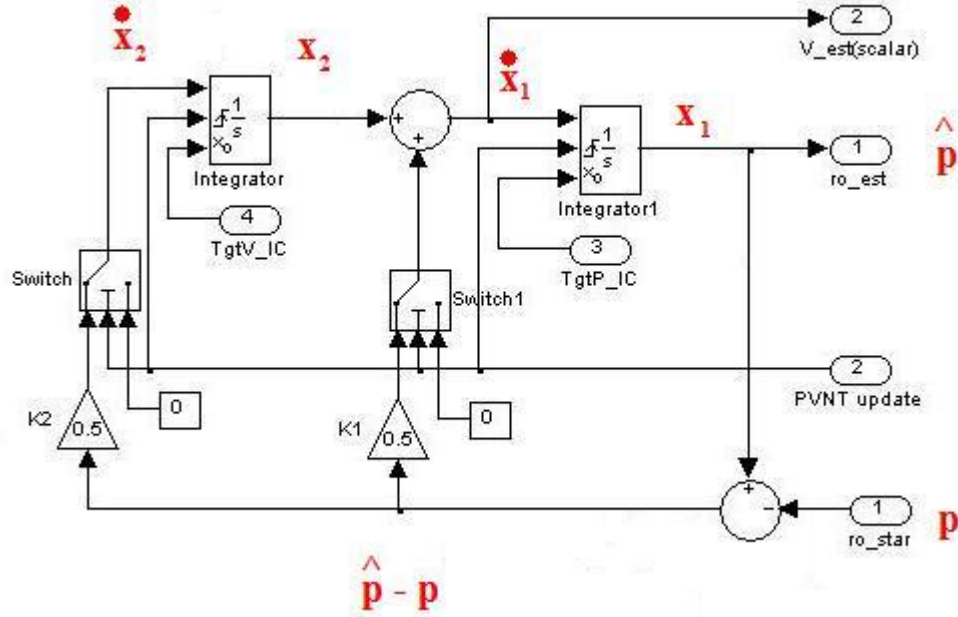


Figure 12. Asynchronous filter model with state variables

Using the state variables shown in Figure 12, the state equations are as follows:

$$\begin{aligned} \dot{x}_2 &= K_2 \left(\hat{\rho} - \rho \right) & \text{and} & & x_2(k+1) &= x_2(k) + \dot{x}_2 dt \\ \dot{x}_1 &= x_2 + K_1 \left(\hat{\rho} - \rho \right) & & & x_1(k+1) &= x_1(k) + \dot{x}_1 dt \end{aligned} \quad (2)$$

These state equations for Euler integration can easily be implemented in MATLAB code utilizing the data already stored in arrays and loops within the script.

B. REAL-TIME MODELING

The non real-time models can be used as a starting point for developing the general and road following models that can be implemented in real-time. Problems encountered by shifting to a real-time model and their solutions are first discussed followed by the actual design of the general and road following real-time models.

1. Problems

Many of the problems encountered during the conversion from a non real-time to real-time model dealt with synchronized data storage and retrieval. The computation speed of different methods of modeling the system is also analyzed and discussed.

a. Data Storage

One of the problems that arose when modifying the MATLAB code to allow the system to run in real-time was the method of data storage. The simulation of the system that had been created in Simulink simply wrote all of the data to arrays in the function code and by using “to workspace” blocks in the simulation model. There is no problem with this method when the system only runs for 180 seconds, as in the tests for the non real-time road following model in Chapter IV. A system that is actually implemented in hardware, however, may run from just a few minutes up to several hours. Hours of run time can result in massive amounts of data from the programs being executed. Additionally, it is not practical to increase the step size of the program to reduce the amount of data collected as accuracy will suffer as a result of the decreased number of inputs.

The solution, in the case of this system, is to only hold the minimum amount of data required before releasing it. The next decision to be made is how much data actually needs to be stored. The system being studied has a delay associated with the PVNT processing time, assumed to be anywhere from one to ten seconds. Once the

PVNT position update is calculated for time τ (equal to the current time, t , minus the PVNT delay), it is compared to the estimated target position at time τ . This is the first portion of the asynchronous filter implemented through the Euler integration embedded in the MATLAB function. Therefore, the minimum amount of data that can be stored without affecting system function is the maximum PVNT delay divided by the step size or,

$$\frac{delay_{max}}{dT} = \frac{t - \tau}{dT} \quad (3)$$

From this equation, the code can be adjusted to allocate only enough data to account for the maximum expected PVNT delay. The resulting program will help avoid data overflow and storing huge amounts of data over prolonged run times.

b. The S-Function

The next problem is how to simulate the system with the necessary speed for real-time implementation. The solution to the dilemma is found in the S-function block in Simulink. The S-function block is linked to an S-Function file containing C code (in this case) that can carry out all the necessary tasks of a system with the speed needed for real-time simulation. Therefore, every MATLAB function shown in the earlier sections of this chapter such as the asynchronous filter, optimization, and data storage functions needed to be implemented in C code. After the code compiles without error, it is converted into a MATLAB .mex file, which allows it to be used by the Simulink model. The S-function can receive inputs, send out outputs, and make function calls, combining the relative simplicity of a Simulink model with the speed and capabilities of C code.

c. Arrays vs. Buffers

A problem encountered with the conversion process from MATLAB code to C code involved the fact that the size of an array in MATLAB does not have to be preset, but an array in C code does. The maximum size of an array in C code is set by using an integer to define the number of data storage spaces. Unfortunately, this means

that global variable or a parameter cannot be used to initialize the maximum array size. Therefore, the method of data storage in C code would have to be different from the methods used in MATLAB code. Since the delays of the PVNT update vary anywhere from one to ten seconds and the code had to be robust enough to handle a larger delay if the user required it, an array-based data storage system would not be practical in C code. While arrays are typically simpler to write code-wise, overflow of an array can cause errors that may prematurely end the simulation. Another problem that was encountered by using an array is that the data stored inside an array is reset following each iteration of the program. Thus, the program is not able to access information stored during previous runs, which is necessary for the asynchronous Euler integration process. As a result of these shortfalls, it was determined that another means of storing the accumulated data was needed.

While several means of data storage were tested, the only method that overcame the disadvantages of arrays and met all the requirements needed for data storage was to use buffers. While the size of a buffer does need to be preset, parameters in the S-function can be used to perform the task, even though they couldn't be used to preset the sizes of arrays. This means that the user does not have to open the actual C program and change lines of code if the maximum expected PVNT delay were to change. The user can simply change one number in the parameter input of the S-function block and have the maximum buffer size reset automatically by the code.

Additionally, buffers are a type of persistent memory, meaning that the data stored inside remains saved until a command to clear the buffer is given. Therefore, the data from the open loop filter can be stored in buffers and recalled at the start of the asynchronous Euler integration. While the coding of the buffers is more involved than setting up a group of arrays, the requirements of the system make buffers the ideal method of data storage.

2. General Filter

Since a model or equation for the road may not be known ahead of time in most real-time situations, the general non real-time filter simulation from the beginning of this

chapter was first prepared for implementation in the S-function. The asynchronous filter portion along with all data storage from the asynchronous filter needed to be implemented in C code and moved inside the S-function.

a. The True Target Model and PVNT Update Generator

The simulation of the real-time system will not use an actual tracked ground target, so a model needs to be used during the testing process.

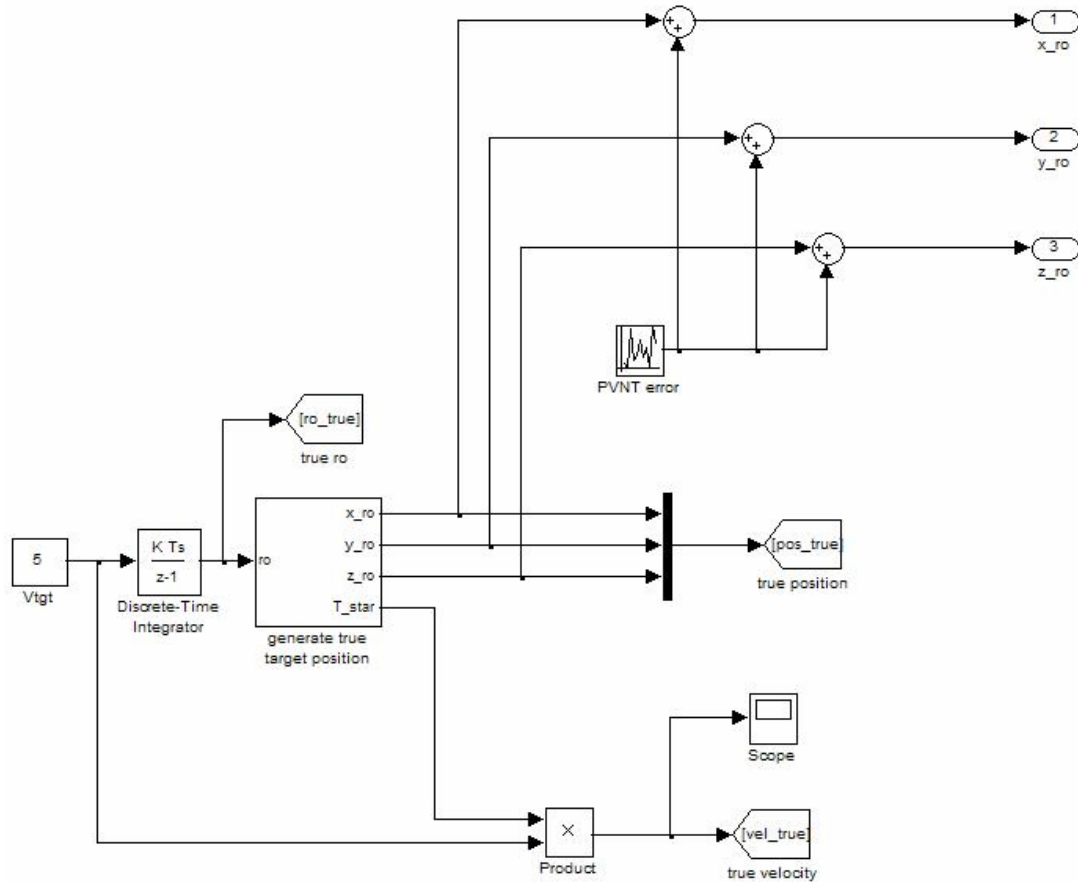


Figure 13. True target model and PVNT update generator for general filter

The diagram in Simulink is nearly identical to the one used in Figure 5 in the system simulation from earlier in the chapter. The target model begins with a velocity that is integrated and then sent to target position generating subsystem to determine the target position coordinates. The subsystem defines a preset road model on which the

target will travel. In the case of this system, only the x and y coordinates are used for the 2-D target tracking, while the z coordinate is set to zero, allowing it to be included for possible future use.

Additionally, the true target model doubles as a generator for the PVNT position updates. The outputs from the position generation subsystem are utilized as a portion of the PVNT update as well. Since the assumed accuracy of the position update is ± 1 meter, the model incorporates this deviation by means of a random number generator (with a mean value of zero and a range of ± 1) through a summing junction. The modified PVNT position update is passed to the S-function for later use.

A small difference in the true target model from the original non real-time subsystem block shown in Figure 5 is the removal of all continuous state blocks. The integrator block is one such tool that had to be altered during the transition to a system capable of real-time calculations. Since samples are only taken every dT seconds by the system during simulation and a fixed step solver is used by the Simulink model, only discrete state blocks can be used. Therefore, every continuous time integrator block used in the true target model and filter subsystems had to be swapped with discrete state integrators.

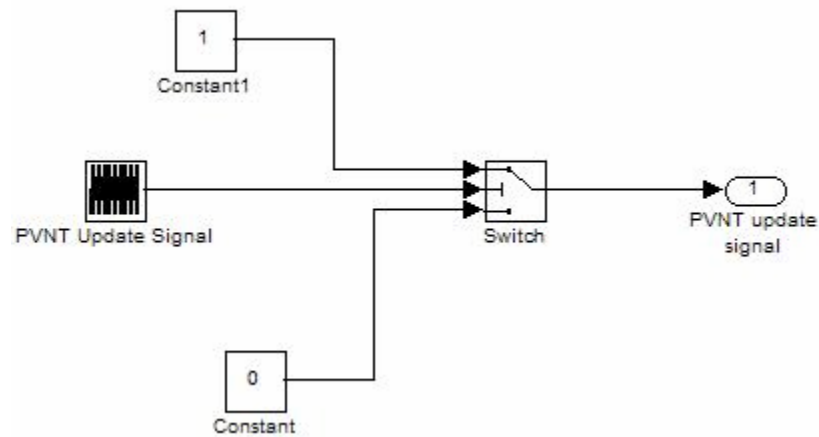


Figure 14. PVNT update signal subsystem

Based on previous information regarding the PVNT computation time, it is known that the delay associated with the position update can range from one to ten seconds. Therefore, a simple pulse generator could not be used due to the need for a varying delay time. The PVNT update subsystem shown above solves the problem by offering a pseudo-random sequence covering the full range of delay times.

b. The Open-Loop Filter

During periods of operation when a PVNT update is not present, the open loop, single integration filter performs the dead reckoning calculations for target position. Unlike its road following counterpart, the general filter system does not have a ρ value based on the known road model with which it can simplify calculations. Therefore, each individual coordinate has to be passed through its own open loop integrator in order to compute the updated position. So, even though all the open loop filters are connected to the same reset trigger, the x , y , and z open loop filters receive their own respective position and velocity initial conditions. The entire subsystem is contained in a function call block that can be initiated by the S-function.

c. Overall Real-Time Design and Function with S-Function Block

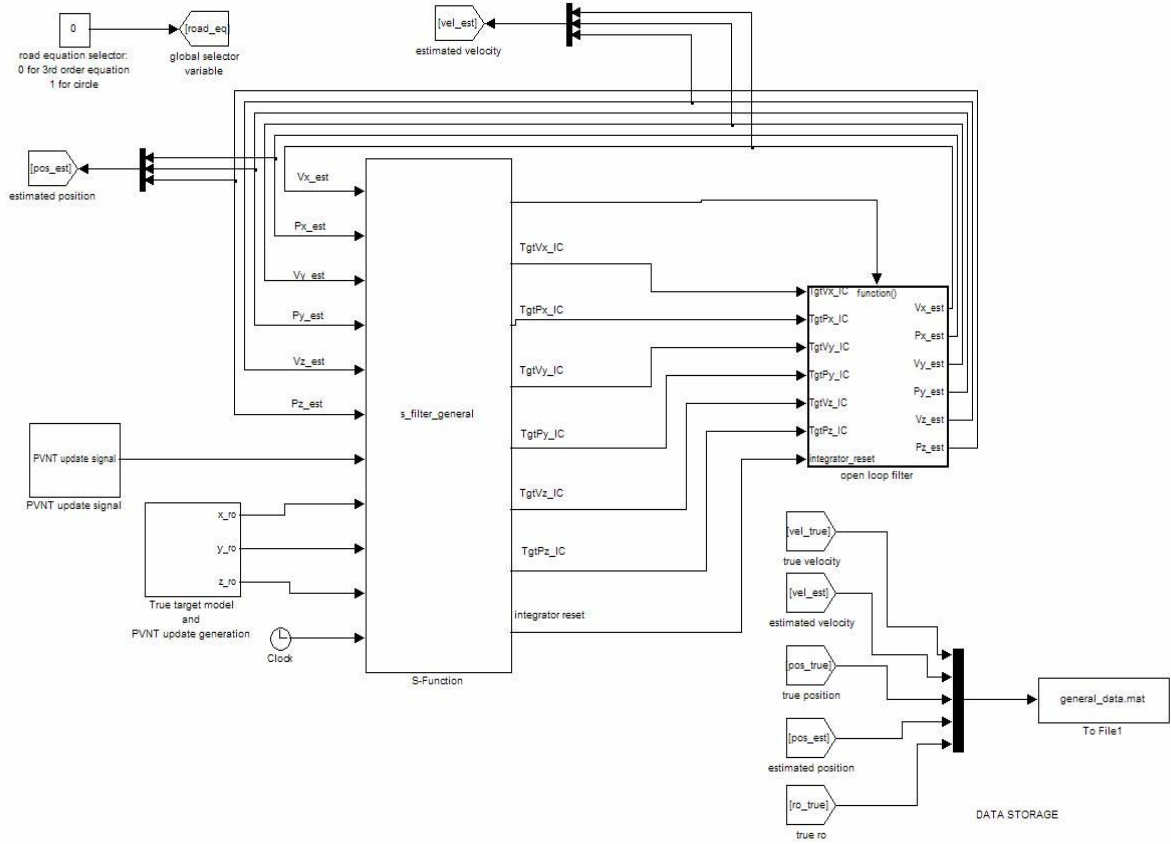


Figure 15. General filter real-time model

There are more inputs and outputs to the individual subsystem blocks without the added simplicity of the ρ road equation variable. The real-time general filter model really begins with the true target model subsystem and PVNT update signal blocks. The PVNT update signal is a repeating sequence that simulates a varying PVNT update delay from one to ten seconds. Until the signal goes high, indicating a PVNT position update is available; the data from the PVNT update portion of the true target model is ignored. The last known x , y , and z coordinates and velocities are passed to the open loop filter where they are sent through the single integration system. The updated positions are then fed back into the S-function to be stored in the proper buffers before repeating the process.

However, upon the receipt of a PVNT update, the path of data slightly changes. The position update for time τ arrives at the inputs of the S-function block and is taken into the C code. Then, the time from the last PVNT update is calculated (delay) and used to determine the value for time τ . Using the persistent memory characteristic of the buffers, the estimated position of the target at time τ is then compared to the PVNT update and sent to a C function that performs Euler integration up to the current time t , storing the new position and velocity data in buffers after each iteration. The final estimated position and velocity data from the Euler integration function for time t are passed on as the initial conditions for the open loop filter subsystem. The integrator reset is also triggered before the open loop filter calculations continue until the next PVNT position update.

During the model simulation, a storage block is used to send all the pertinent data to a .mat file. A separate script file in MATLAB loads the .mat file and automatically plots the actual target data versus the estimated target data from the filter. The data storage section of the model diagram is for testing purposes only as this process would be altered in an actual real-time simulation to avoid errors associated with storing of the immense amount of data.

3. Road Following Filter

Compared with the general filter design described above, the real-time road following filter design was greatly simplified by the pre-known road equation. This equation allowed the x , y , and z position coordinates to be combined into one parameter: ρ . Not only did the road following model appear less cluttered, the C code was also somewhat simpler since only one calculation was needed in most cases where three were required before.

a. The True Target Model and PVNT Update Generator

The true target model, PVNT update generator, and PVNT update signal generator for the road following filter system are identical to the subsystem for the

general filter design shown in Figure 13. The only difference concerning the PVNT position update is in the optimization function contained in the S-function's C code:

(1). Optimization Method. While the PVNT update that is entered into the S-function does not change from the real-time general filter model to the real-time road following filter model, there is an additional set of calculations that takes place afterwards. Located in a function declaration in the C code of the S-function, the optimization loop finds the closest point on the known road equation to the given PVNT update and sets that point as the new ρ update. The optimization method for the real-time model replaces the *rf_optimise.m* script file used in the MATLAB function block from the non real-time road following model. In order to keep computation time to a minimum, the optimization function in the S-function C code calculates the distance from the PVNT coordinate update to set points on the road utilizing a dichotomy algorithm. This set of equations controls the adjustments made to the boundaries of search for the minimum distance, proving to be much faster than computing the distance equation for each point along the road within a set range. This direct search method's results have a high order of accuracy while requiring a minimal amount of computation steps. Following completion of the optimization loop, the new ρ value is outputted to the rest of the S-function code.

b. The Open Loop Filter

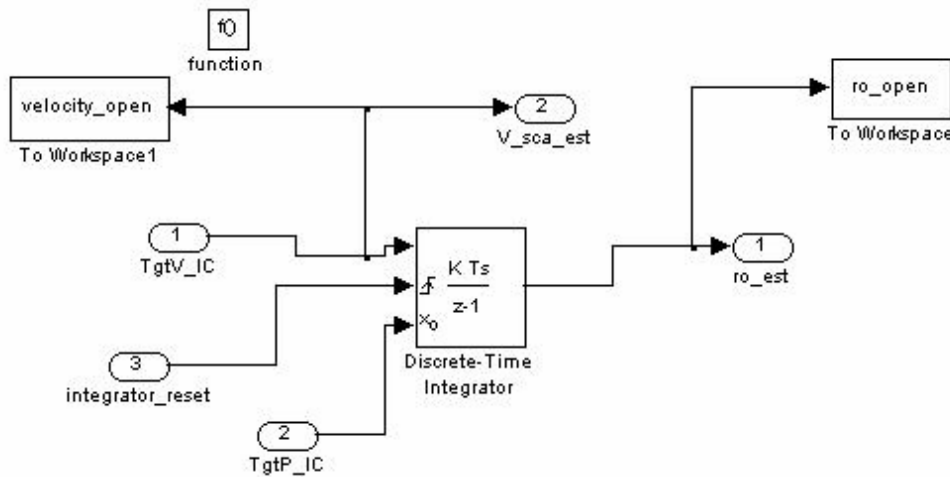


Figure 16. Open loop filter for the road following model

Figure 16 shows the simple open loop integration that calculates the update for ρ and outputs the results back into the S-function for storage and further use.

c. Overall Real-Time Design and Function with S-Function Block

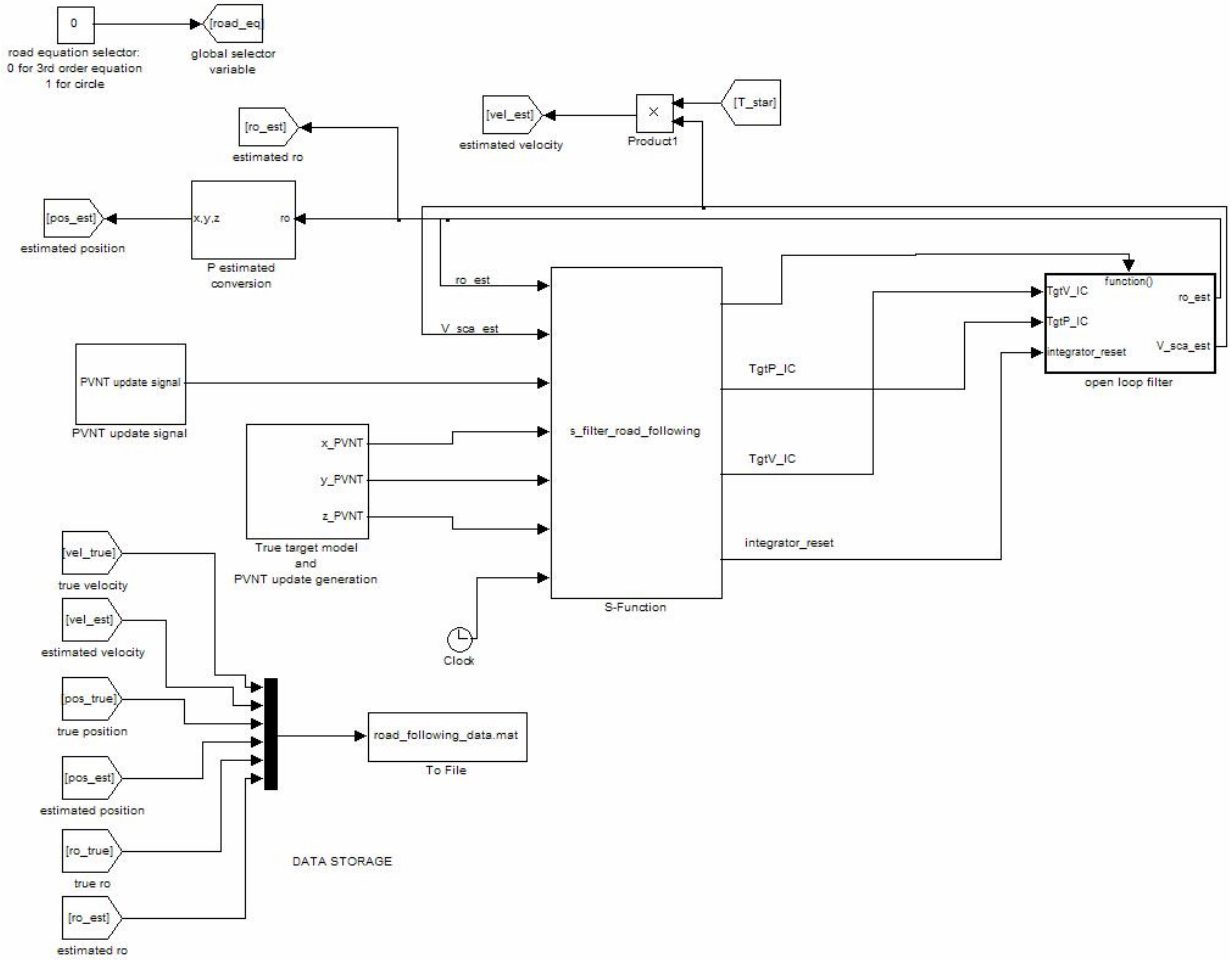


Figure 17. Road following filter real-time model

There are fewer inputs and outputs for the real-time model than the non real-time model. While the x , y , and z coordinates are combined into the ρ variable, the system function is nearly identical to the general filter. The open loop filter function call block still performs the dead reckoning integration until a PVNT position update is received and passed through the optimization function. The asynchronous forward Euler integration takes place in the C code inside the S-function but now with only the ρ

variable requiring integration from time τ to time t . As a result, only the ρ variable is stored in the buffers before being sent out as the initial condition to the open loop filter as the process repeats itself.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SIMULATION AND RESULTS

The purpose of this chapter is to test and compare the results from the developed real- and non real-time systems. First, different road models used in the simulations are defined. Next, the simulation parameters for east test set are defined as well as a short description of the gain values used in the asynchronous constant gain Kalman filters. The results portion of the chapter begins with simulation data from the non real-time road following filter using the two different types of integration (using an external Simulink model file vs. numerical forward Euler integration) discussed at the beginning of Chapter III. Finally, the real-time general and road following models are tested under a variety of conditions before the data is plotted and discussed.

A. SIMULATION

Two different road models were developed for simulation to determine the effects of different road characteristics on the performance of the general and road following filters. Additionally, the simulation parameters are defined as different values for certain parameters are needed for different road models.

1. Road Models

Using two road models allows a better comparison between the general and road following filters on a case-by-case basis. Each road model is created by a system of equations in the x and y planes, while z is set to zero.

a. *Third Order Road Model*

The first road model is a third order system based on the set of equations below:

$$\begin{aligned}x &= \rho \\y &= 0.0000192\rho^3 - 0.025\rho^2 + 9.74\rho \\z &= 0\end{aligned}\tag{4}$$

While the non real-time system simply uses an embedded MATLAB function (as seen in Figure 5) to simulate the road model, the real-time systems are not able to employ these embedded functions. To reduce the amount of computation time required during simulation, the road equation is created using Simulink blocks instead. The subsystem is found in the true target model for both the general and road following models. Additionally, the subsystem also calculates the derivative of each equation for use in the T_{star} variable, which is used in the computation of true and estimated velocities in the road following filter model.

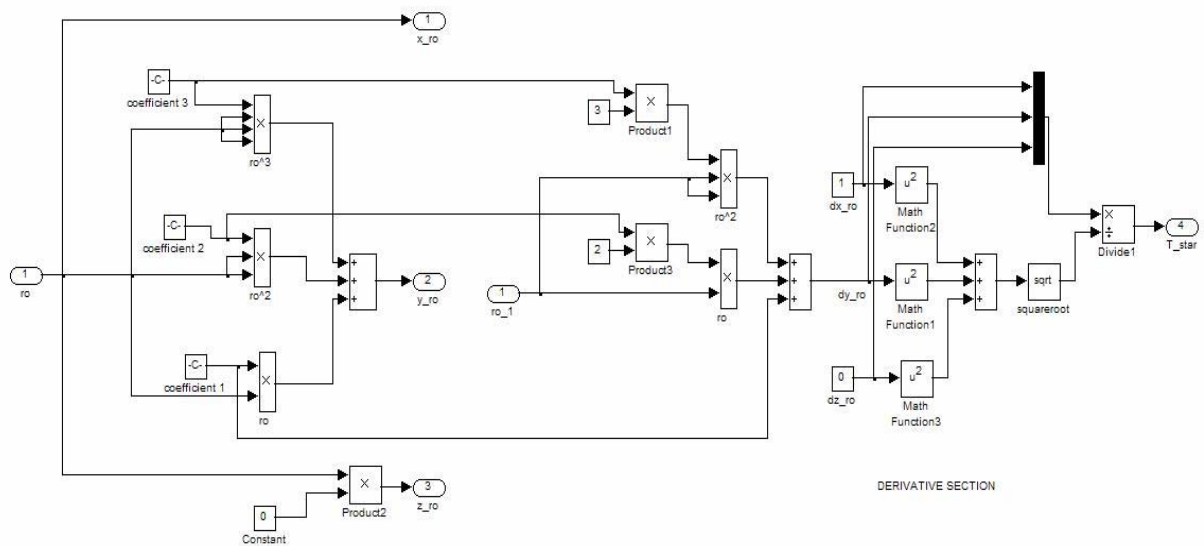


Figure 18. Third order road equation in Simulink subsystem for real-time simulation

When simulated for a three minute test, these equations resulted in the road model depicted in Figure 6.

b. Circular Road Model

It was decided that the second road model should be of a closed loop style similar to a rectangle or circle. Since the vehicle model uses a constant velocity during the simulation, a system of equations for a circle of constant radius was developed:

$$\begin{aligned}
 r &= \text{radius} \\
 x &= r + r \sin\left(\frac{\rho}{r} + \frac{3\pi}{2}\right) \\
 y &= r \sin\left(\frac{\rho}{r}\right) \\
 z &= 0
 \end{aligned} \tag{5}$$

Identical to the third order road model, the circular road model equations had to be created in Simulink without the use of embedded MATLAB functions. The circular road equations are not used in the non real-time simulations.

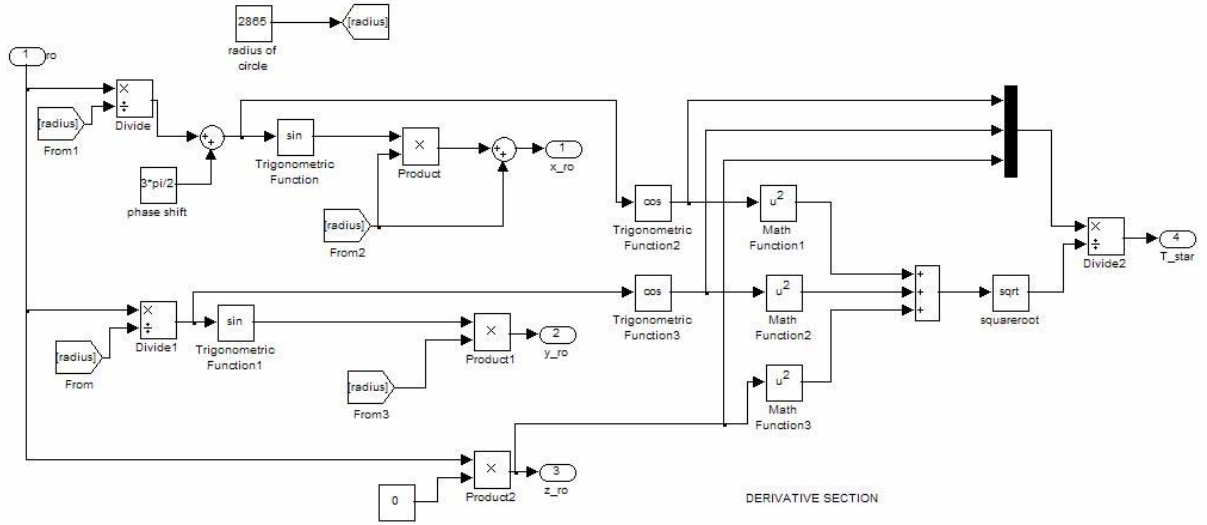


Figure 19. Circular road equation in Simulink subsystem for real-time simulation

The radius of the circle could be set to complete one loop during the simulation. A longer simulation time was chosen to display how the real-time filter does not produce errors associated with data overflow during extended tests. In the trials for this thesis, an hour long simulation was chosen, resulting in a circle radius of 2865 meters and a road model shown in the figure below:

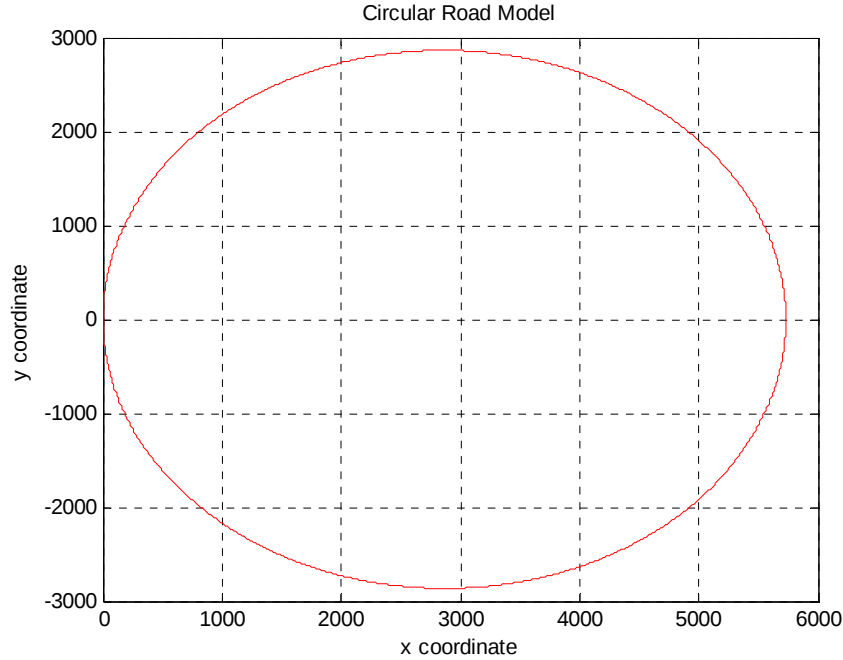


Figure 20. Circular road model

2. Simulation Parameters

a. Simulation Time

The simulation time is set to 180 seconds for the third order road model and 3600 seconds for the circular road model. NOTE: The simulations for the non real-time road following filter only use the third order road model.

b. Sample Time

The sample time used during both the non real-time and real-time simulations for the Simulink model is 0.1 seconds. Additionally, the sample time for the general and road following S-function blocks in the real-time simulations is 0.1 seconds.

c. Asynchronous Kalman Filter Gains

The gains $k1$ and $k2$ are both set equal to 0.5. While the initial response time is slightly slower than the response time for higher gain values, trial-and-error

testing for both filters in the non real-time and real-time systems has shown that the lower gain values are more robust during periods of high PVNT noise or longer PVNT time delays.

d. Reference Frame

The frame reference used for all simulations is Local Tangent Plane (LTP).

e. PVNT Parameters

The non real-time road following model used the randomized PVNT delay time shown in Figure 4 for all simulations.

The real-time general and road following simulations vary the PVNT parameters over the series of tests. The PVNT position noise is tested at three different values: ± 1 , 5, and 10 meters. The PVNT delay time is also tested for three different scenarios: a simulated pseudo-random delay covering 1-10 seconds, a repeating 5 second delay, and a repeating 10 second delay. The simulated pseudo-random delay is shown in the figure below.

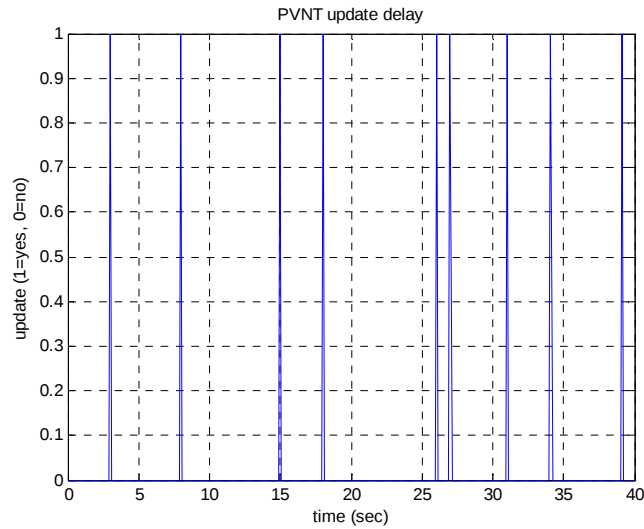


Figure 21. Simulated pseudo-random PVNT update delay

B. RESULTS

The results section is subdivided into the data from the non real-time simulations followed by the data from the real-time simulations. The non real-time simulations contain the road following model with the asynchronous integration performed by the external Simulink model compared to the numerical forward Euler integration method. The real-time simulations include the general and road following models. Each real-time model is also put through a series of tests in which certain PVNT parameters are altered, such as PVNT delay and input noise.

1. Non Real-Time Models

The results for the two non real-time models are divided into three comparisons each: position, velocity, and ρ . Both models need to show that they are incorporating the PVNT updates into the estimated target data and effectively tracking the target throughout the simulation.

a. Road Following Model with Separate Simulink Model Integration

(1) Position Comparison

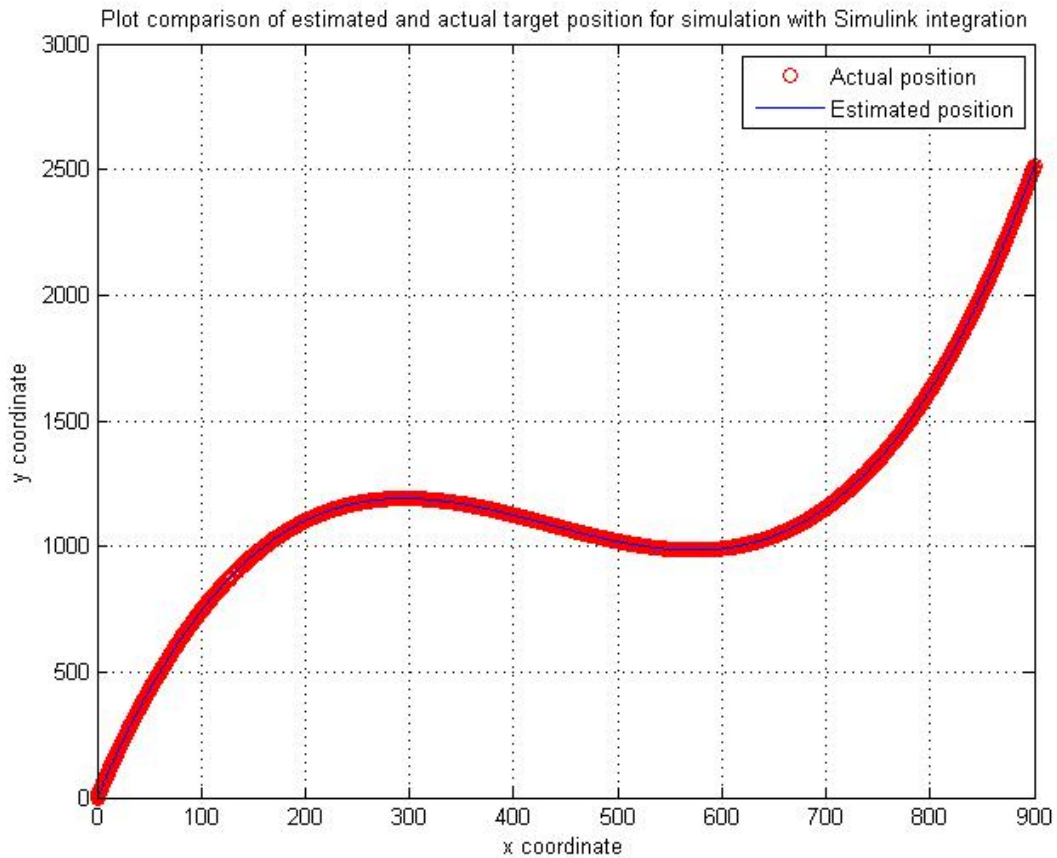


Figure 22. Comparison of actual vs. estimated target position – Simulink integration

Figure 22 depicts the results of the actual target position plotted against the estimated target position from the non real-time road following filter with external Simulink model integration. It is apparent that the position data is accurate and the model does not lose track of the target during the simulation.

(2) Velocity Comparison

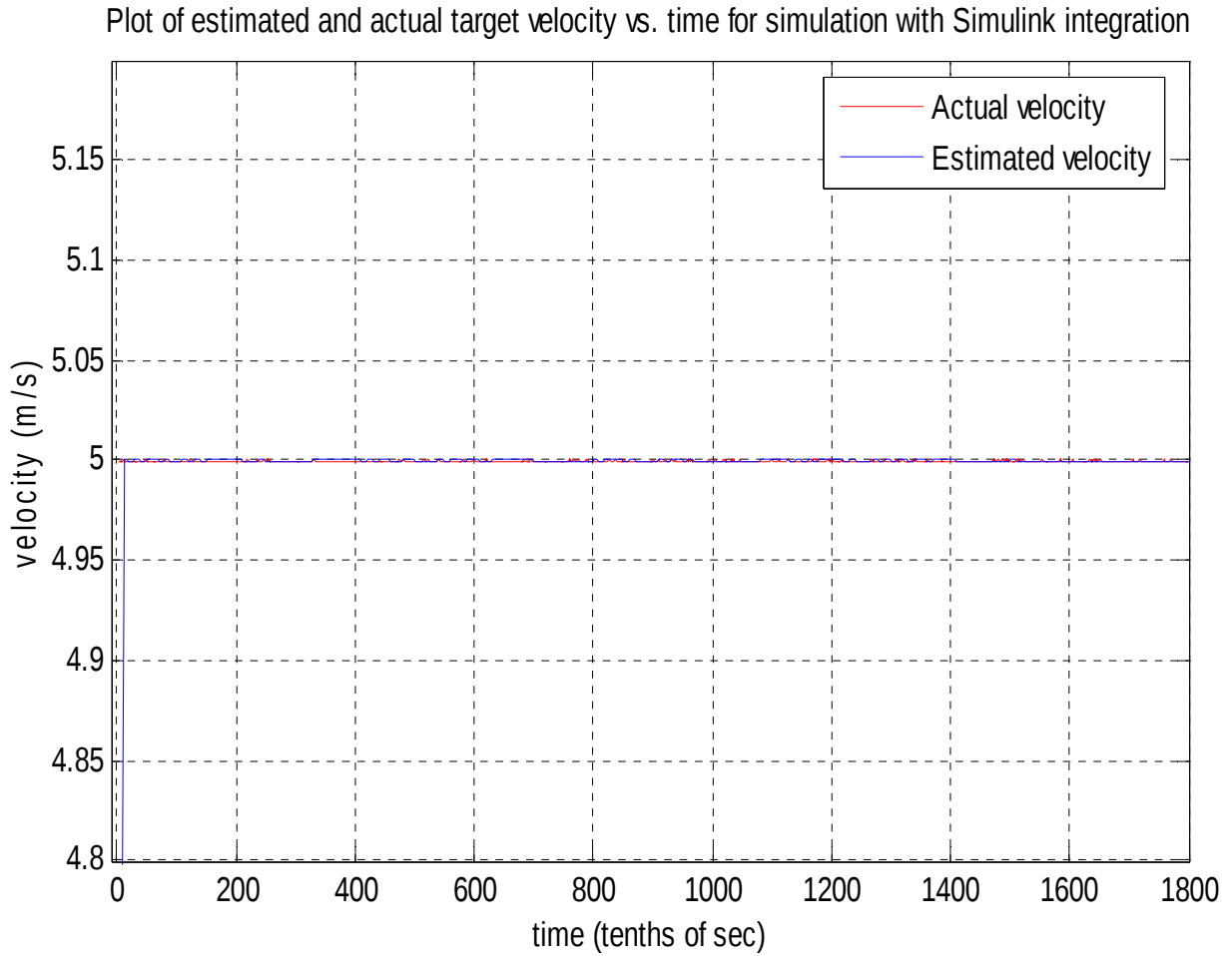


Figure 23. Comparison of actual vs. estimated target velocity – Simulink integration

Figure 23 depicts the results of the actual target position plotted against the estimated target position from the non real-time road following asynchronous filter with external Simulink model integration. Since the initial velocity of the target is assumed to be 0 m/s, the estimated target velocity does not respond until the first PVNT update. The plot is zoomed in around 5 m/s (the true target velocity) to show how the estimated target velocity obtains the correct value with the help of the PVNT updates.

(3) Comparison of ρ Values

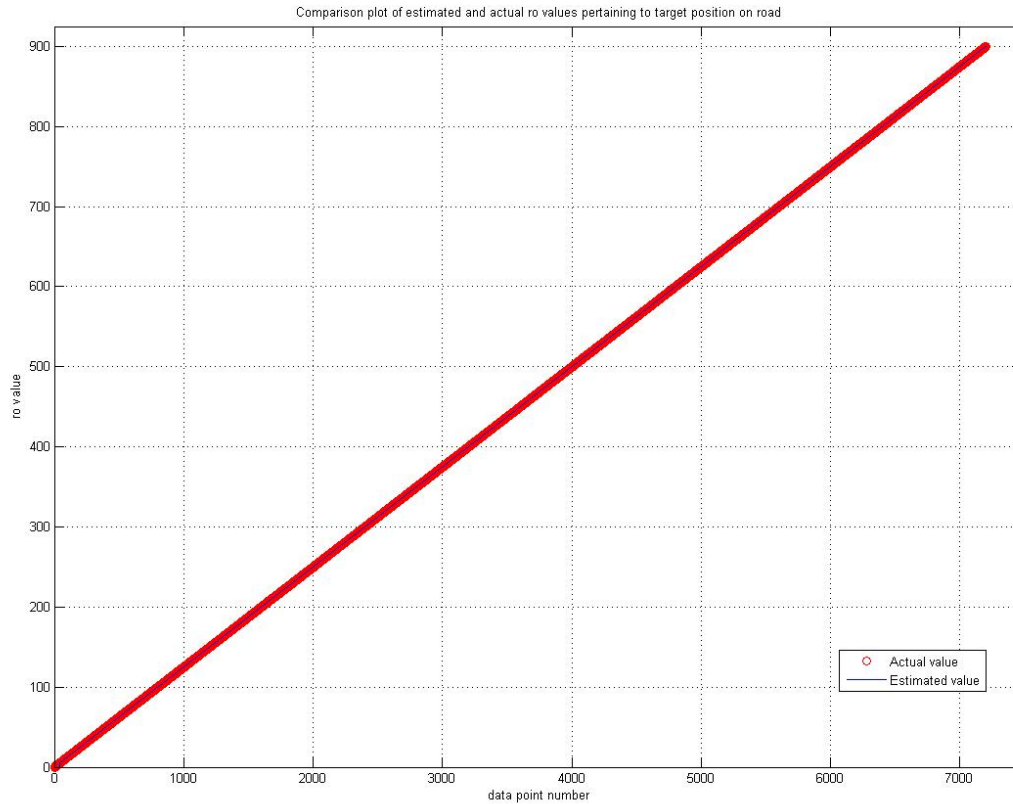


Figure 24. Comparison of actual vs. estimated ρ value – Simulink integration

The final comparison was between the actual and estimated ρ values for the target. Correlating with the accuracies found on the position and velocity comparison plots, the ρ comparison plot shows the same high degree of accuracy.

b. Road Following Model with Numerical Forward Euler Integration

The results for the non real-time system with the numerical integration technique are nearly identical to the method using the external Simulink model file.

(1) Position Comparison.

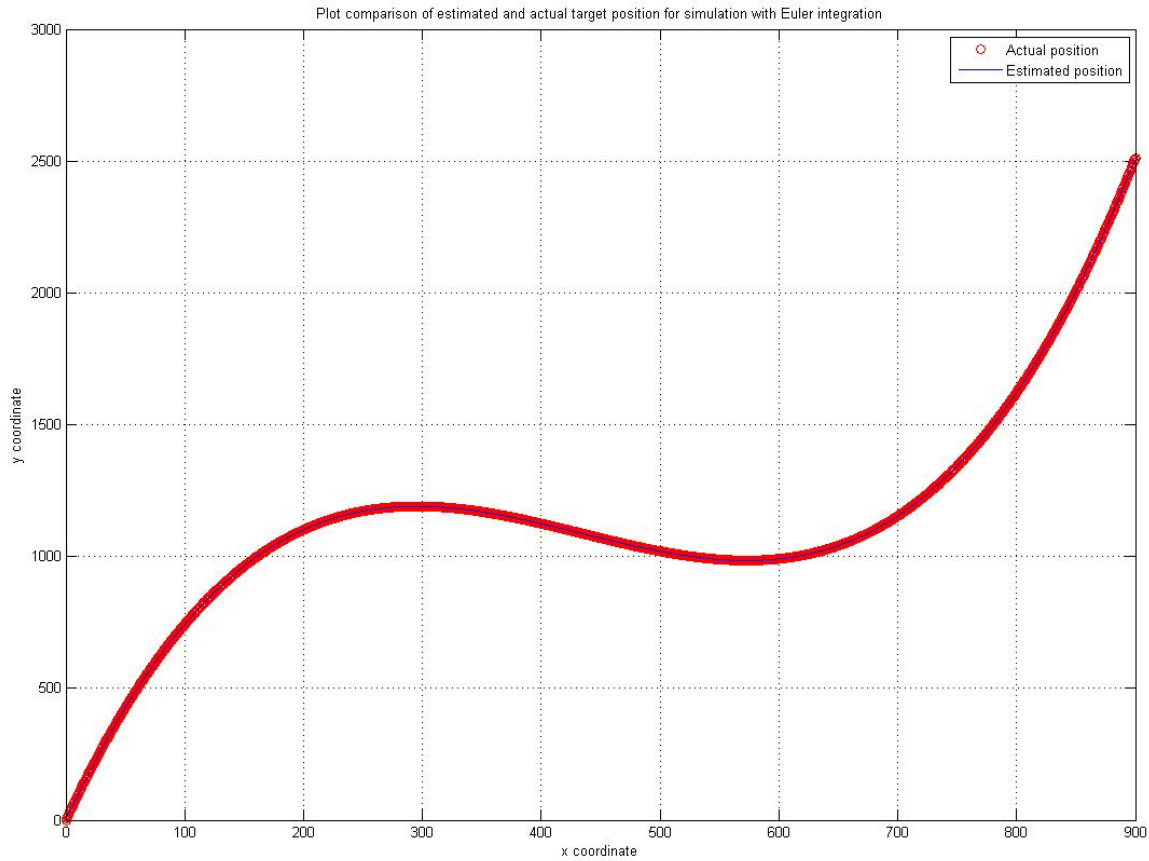


Figure 25. Comparison of actual vs. estimated target position – Euler integration

The figure above shows the actual target position plotted against the estimated target position for the non real-time road following filter using numerical forward Euler integration. The plot shows nearly identical results to the simulation with the double integration performed in the separate Simulink model. The estimated target position matches the actual target position with a satisfactory degree of accuracy.

(2) Velocity Comparison.

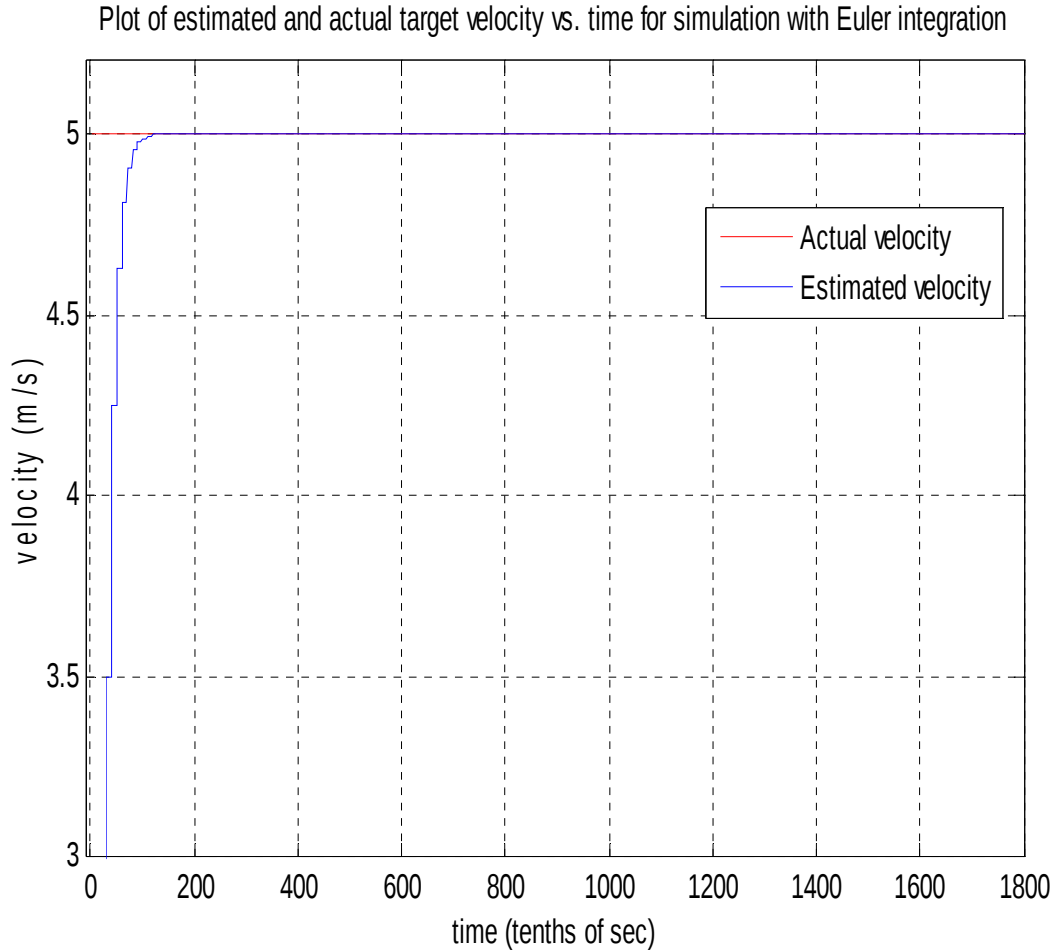


Figure 26. Comparison of actual vs. estimated target velocity – Euler integration

Figure 26 depicts the actual target velocity of 5 m/s compared with the estimated target velocity from the numerical Euler integration. While the response is not as fast as the separate Simulink model double integration, the results show that the steady state error remains at zero and the model effectively computes the estimated target velocity. If the response were deemed too slow for the environment in which the system was placed, the gain values (specifically K2) in the integration loop could be adjusted to compensate.

(3) Comparison of ρ Values.

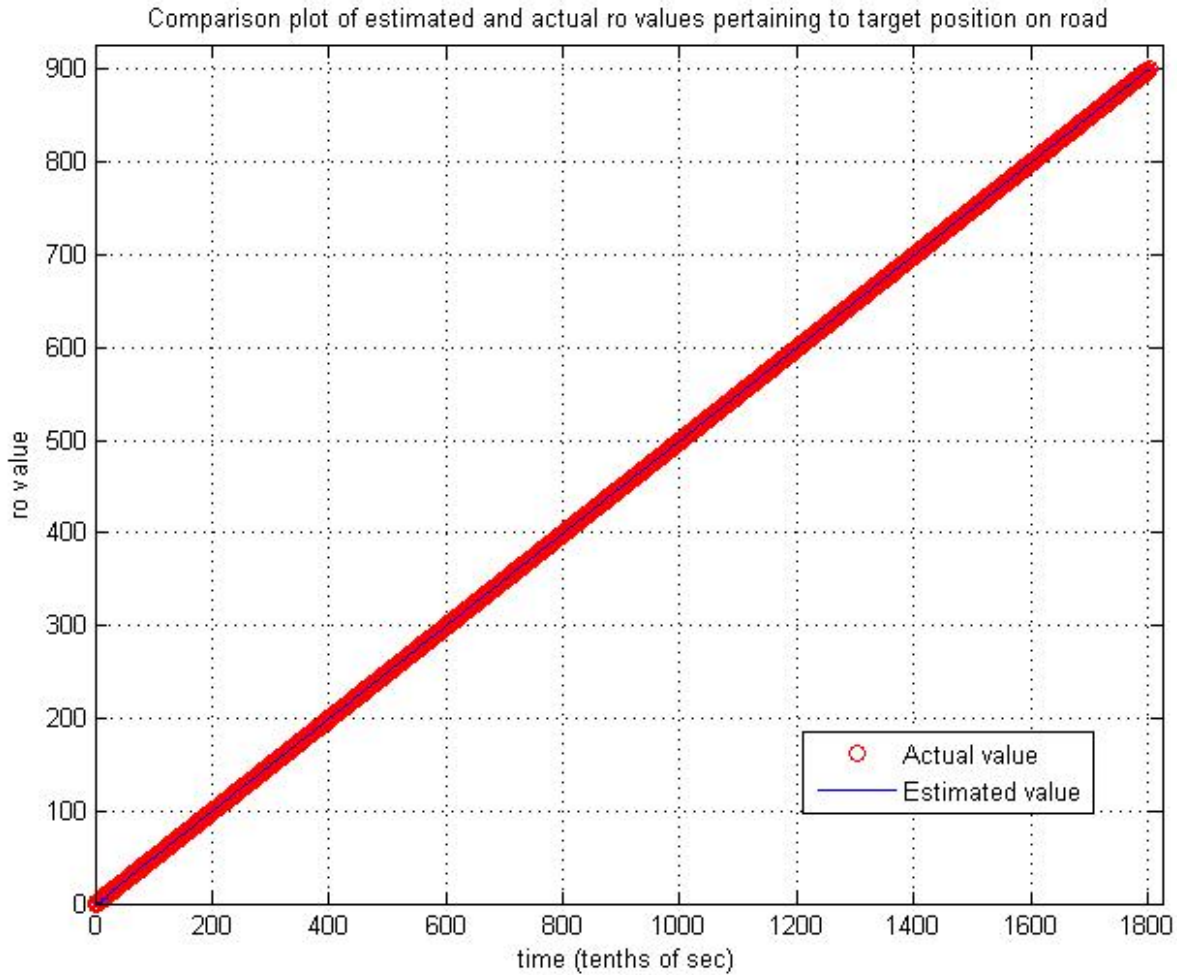


Figure 27. Comparison of actual vs. estimated ρ velocity – Euler integration

Figure 27 shows the actual target ρ value plotted against the estimated target ρ value for the simulation using numerical forward Euler integration. Further confirming that the Euler integration contained in the MATLAB function code is accurate, the data shows nearly identical results.

Overall, the previous three figures show that the simulation can be accurately run using numerical forward Euler integration instead of the double integration process being contained in a separate Simulink model.

2. Real-Time Models

a General Filter

(1) Ideal Conditions. Ideal conditions are defined as a PVNT noise value covering a range of ± 1 meter and a simulated random PVNT delay.

(a) Third Order Road Model

The first test for the general filter uses the third order road model under ideal conditions with a 180 second simulation time. After completion of the simulation, the results are loaded from the .mat file and comparison plots are created.

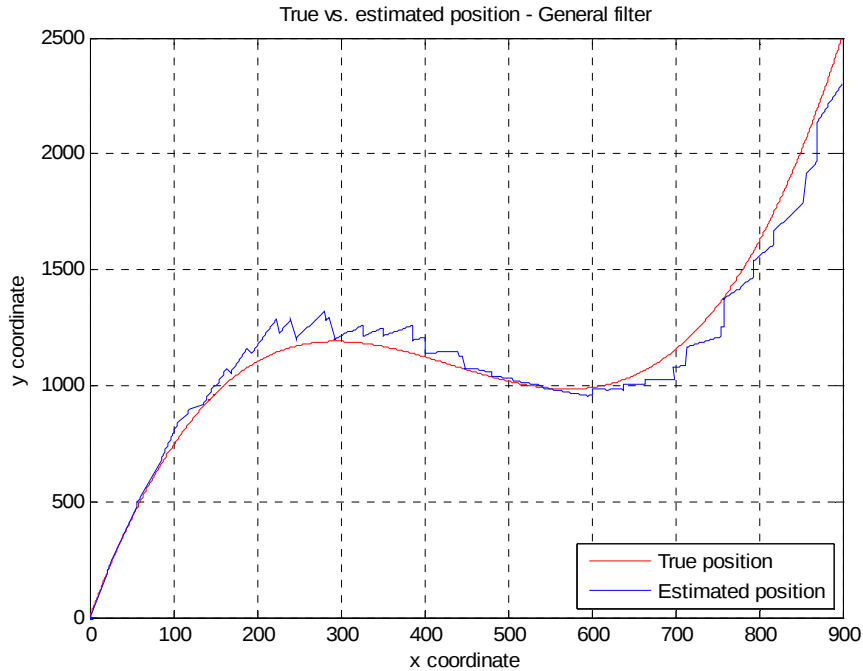


Figure 28. General filter position comparison plot – Third order road model – Ideal conditions

Figure 28 shows the comparison of the target's true position versus the general filter's estimation for the real-time general filter model. The estimated position from the filter is quite accurate for the straighter portions of the road model and less accurate for the curved sections. A reason for the decrease in estimation

accuracy is due to the lack of an optimization function in the general filter s-function. Since the general filter design does include a known road model on which to base the incoming PVNT position updates, the resulting estimated position is heavily reliant on PVNT noise. A plot of position estimation error vs. time is shown below:

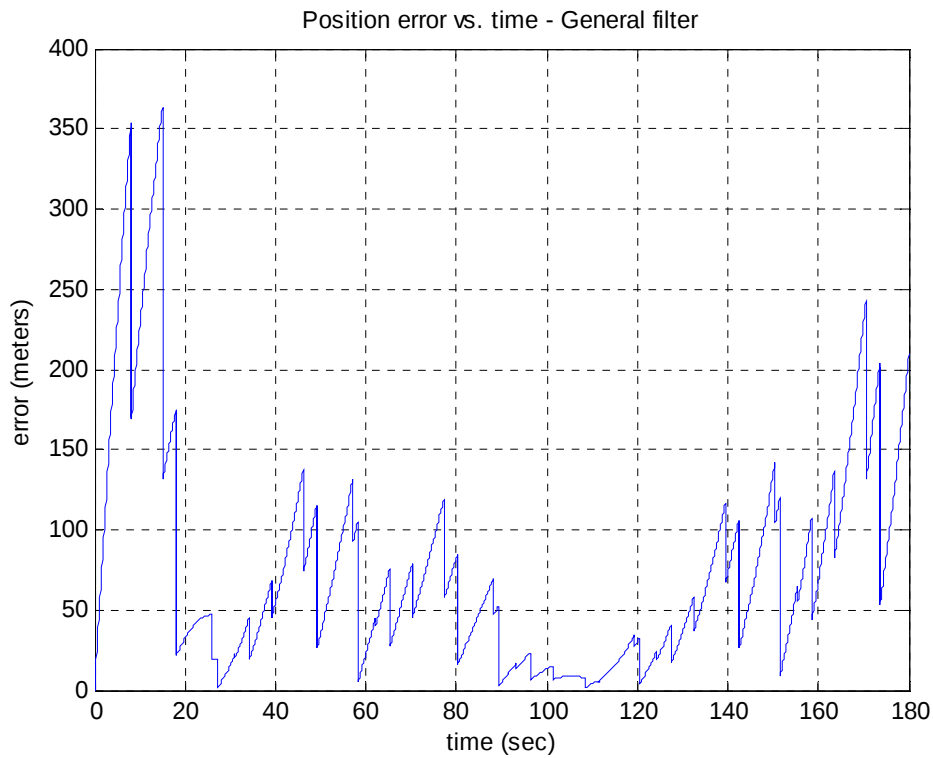


Figure 29. General filter position error vs. time – Third order road model

Further confirming the position comparison plot in Figure 28, the error is greatest at the curved sections of the road and least during the straighter portions.

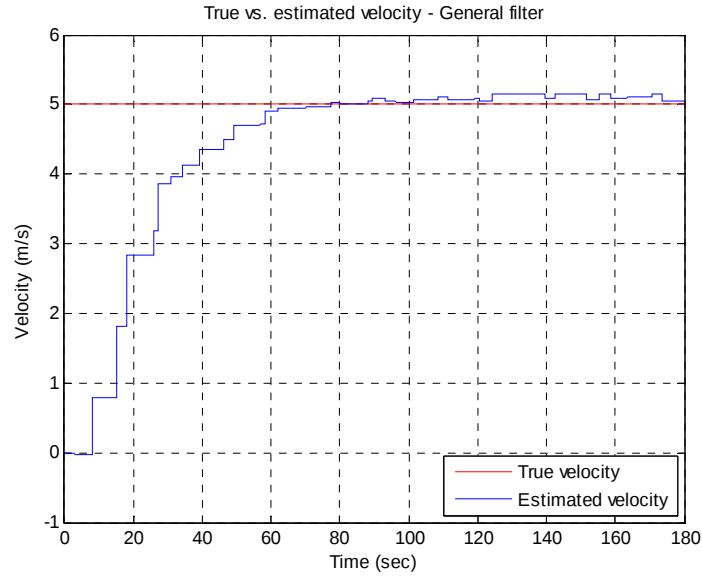


Figure 30. General filter velocity comparison plot – Third order road model – Ideal conditions

The velocity comparison plot shown above shows a small estimation error following acquisition of the target coinciding with the position plot. The overall velocity estimation accuracy is good as it stays at or near the target true velocity of 5 m/s.

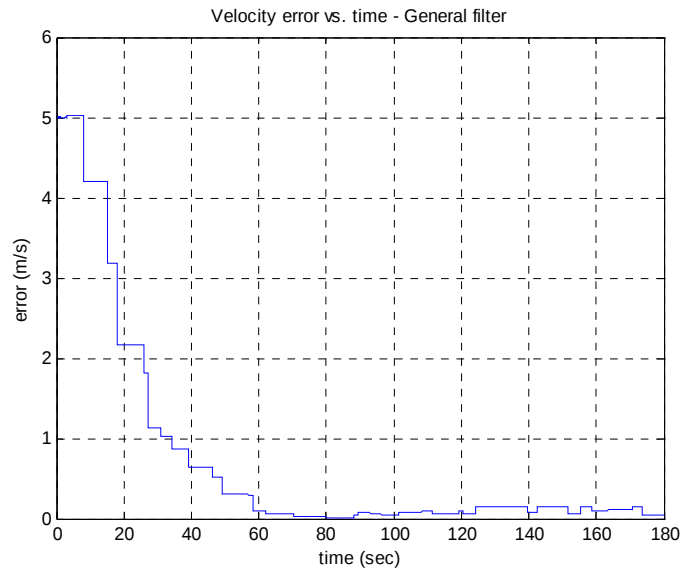


Figure 31. General filter velocity error vs. time – Third order road model

The above figure shows the relationship between estimated velocity error from the real-time general filter and simulation time. After the initial target acquisition, the overall RMS error remains below 0.5 m/s.

(b) *Circular Road Model*

The circular road model simulation is run for one hour of simulation time, allowing the target model to complete one loop of the circular track.

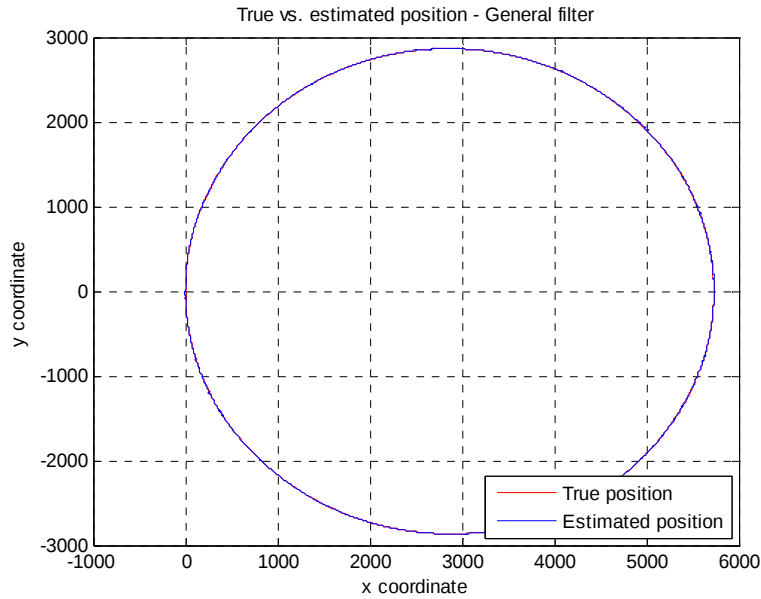


Figure 32. General filter position comparison plot – Circular road model – Ideal conditions

The position comparison plot for the circular road model appears to be much better than the third order road model. One reason for this involves the fact that the target is following a path that does not include any abrupt changes in curvature. Instead the target is engaged in one constant, gradual turn and the dead-reckoning portion of the real-time general filter is able to accurately follow the vehicle's movement.

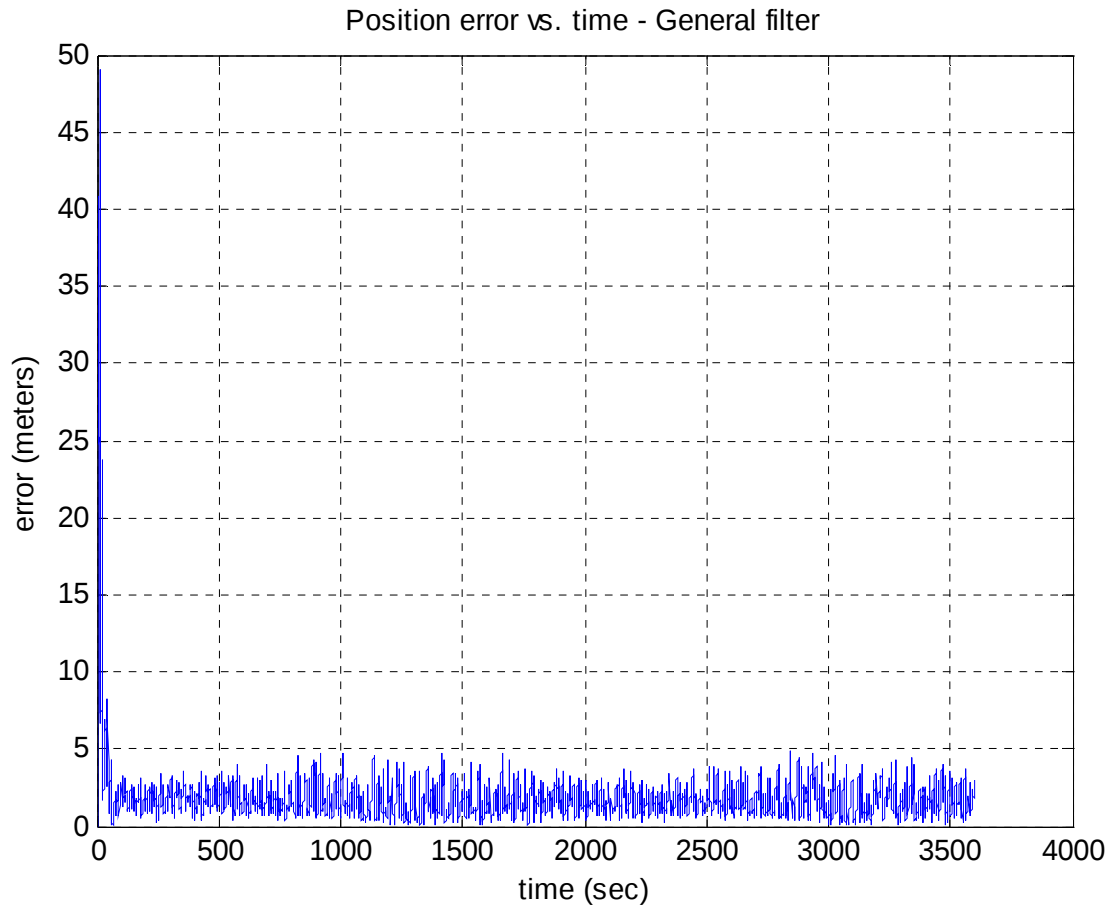


Figure 33. General filter position error vs. time – Circular road model

The real-time general filter is much more accurate for the circular road model than it is for the third order road model as shown in the above figure. The RMS position error is rarely above five meters and is centered at around one meter error due mainly to PVNT noise.

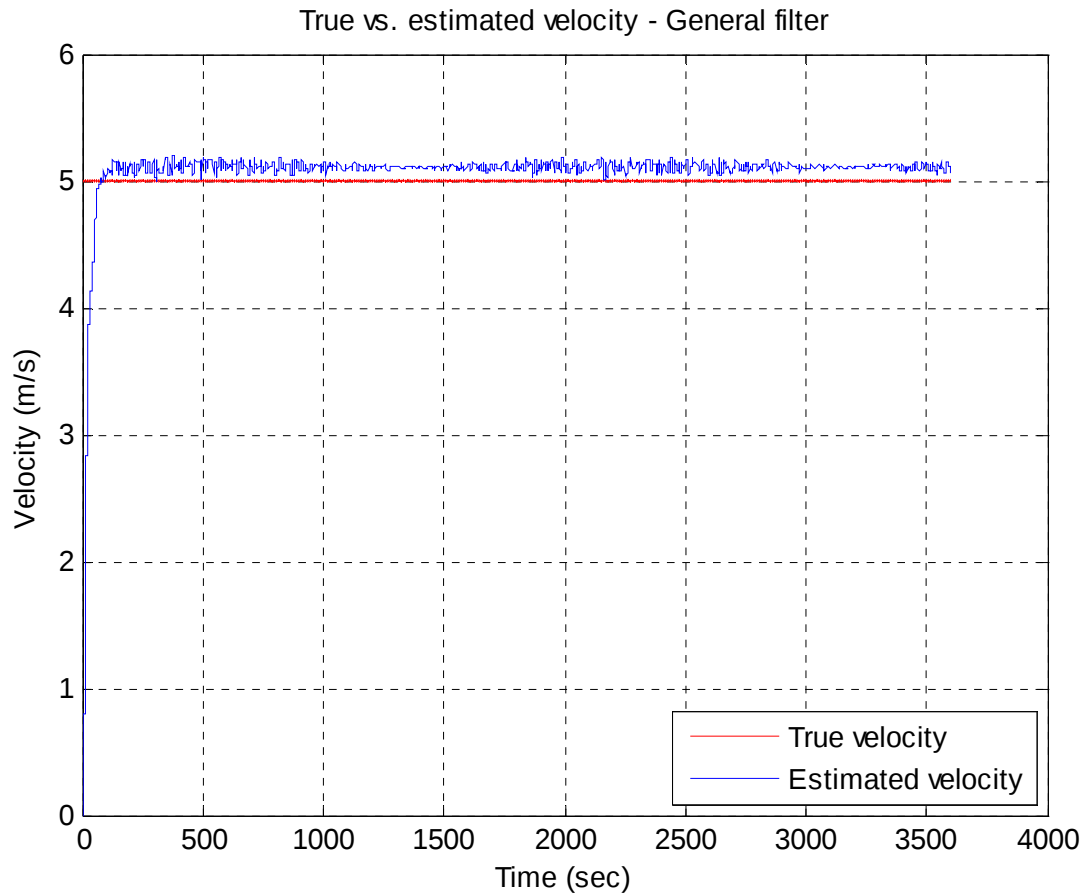


Figure 34. General filter velocity comparison plot – Circular road model – Ideal conditions

The velocity comparison plot for the circular road model is very similar to the velocity plot for the third order road model. The results from the real-time general filter show a fairly accurate estimated velocity that never strays above 5.5 m/s or below 4.75 m/s.

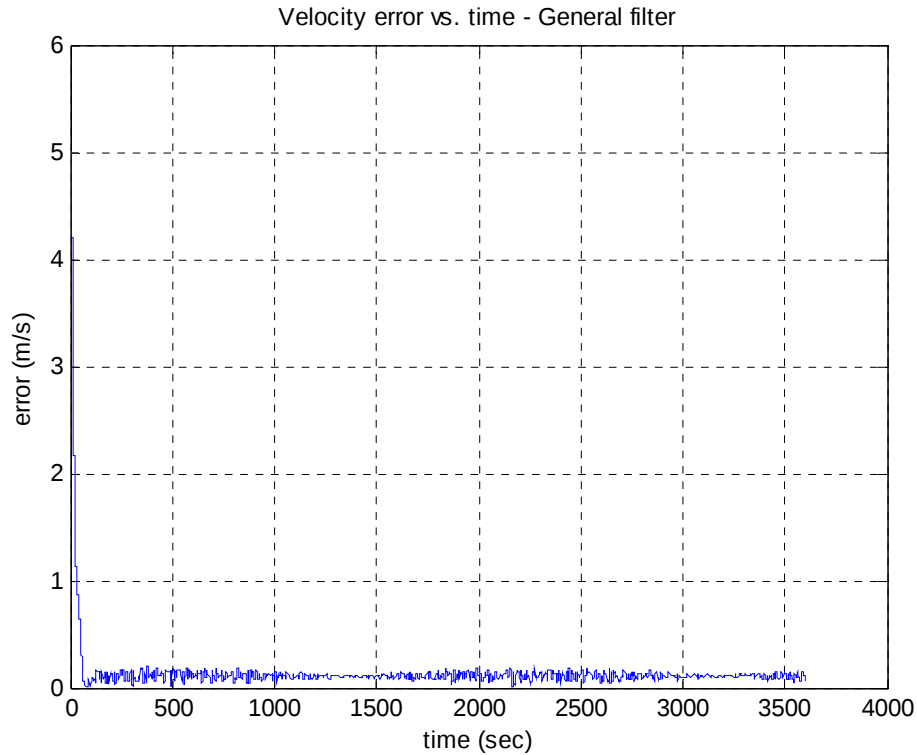


Figure 35. General filter velocity error vs. time – Circular road model

The estimated velocity error plot coincides with the velocity comparison plot for the circular road model. The velocity estimation performed by the real-time general filter is slightly more accurate for the circular road model than it is for the third order road model with a lower RMS error value over the system simulation time.

(2) PVNT Update Delay Variance. The next testing phase for the real-time general filter is to alter the delay time from the PVNT update signal subsystem to view the effects on target motion estimation. Instead of using the simulated pseudo-random update signal, a signal generator block is used to simulate a repeating five and ten second PVNT position update delay.

(a) Third Order Road Model

The first simulation run involved a repeating PVNT update delay of five seconds.

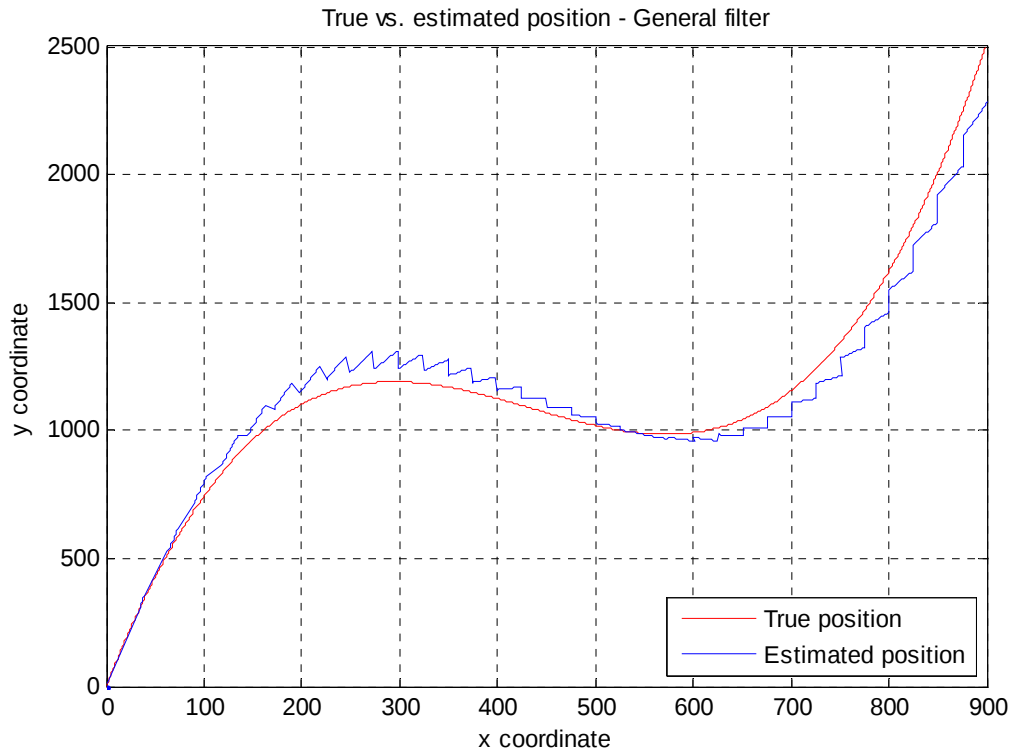


Figure 36. General filter position comparison plot – Third order road model – 5 second PVNT delay

Figure 36 shows the effects of a repeating five second PVNT delay on the general filter model. The result of PVNT updates arriving once every five seconds slightly decreases the estimated target position accuracy, especially around the areas of greater curvature in the road model. The error increases near the end of the simulation due to the exponential road profile equations. Typically, an update with a shorter delay time allows the model to correct itself to be closer to the actual road model in between the larger delay times of five seconds or greater. The position accuracy therefore suffers without the less delayed PVNT updates to fill in the gaps.

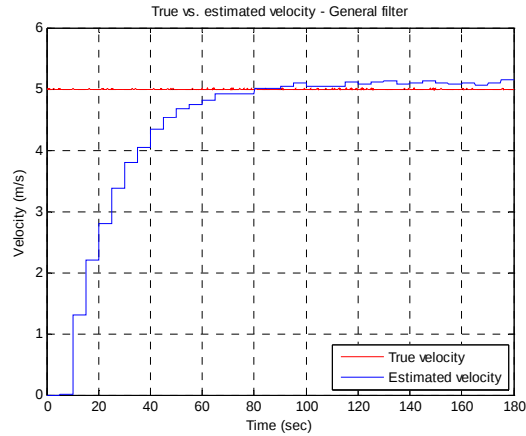


Figure 37. General filter velocity comparison plot – Third order road model – 5 second PVNT delay

The velocity estimation plot shows little or no change from the random PVNT delay times. The real-time general filter remains fairly accurate with a slight bias due to the inputted PVNT noise.

Next, the real-time general filter using the third order road model is subjected to a repeating ten second PVNT update delay. Based on the PVNT background information, a ten second delay is the longest expected delay associated with the PVNT computation time.

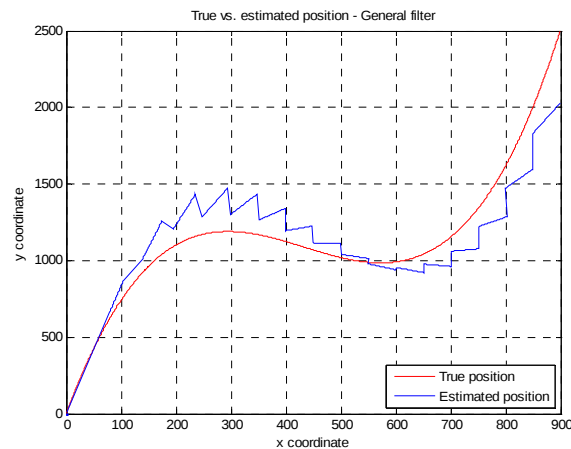


Figure 38. General filter position comparison plot – Third order road model – 10 second PVNT delay

The repeating ten second PVNT update delay greatly decreases the accuracy of the general filter model. The trend of the filter accuracy declining during areas of increased curvature turns along the road continues here as the greatest variances in estimated position accuracy are at the first turn in the third order road model.

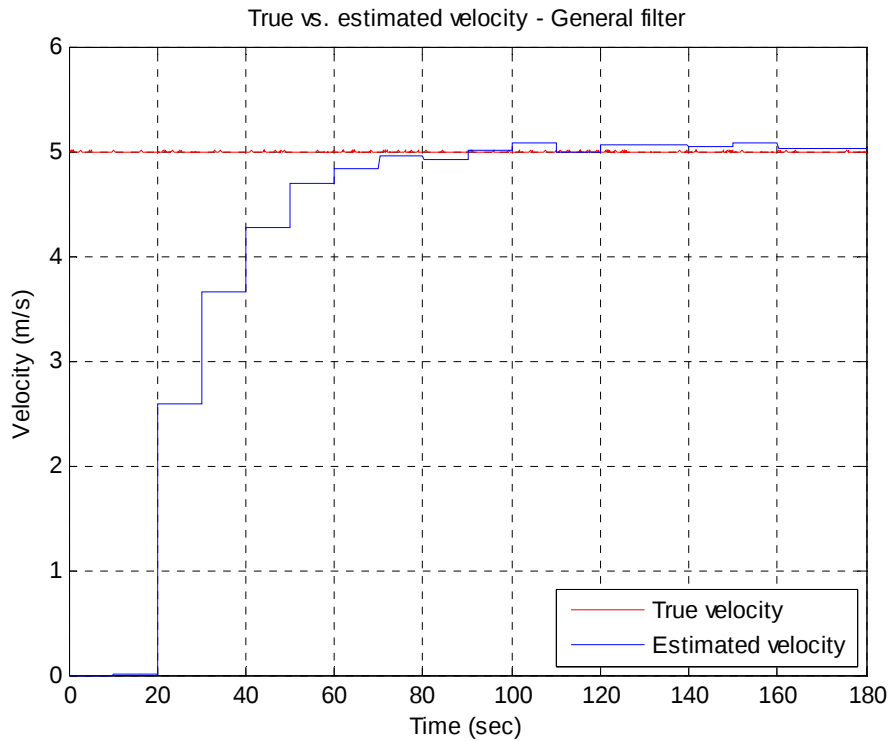


Figure 39. General filter velocity comparison plot – Third order road model – 10 second PVNT delay

The velocity comparison plot for the ten second PVNT update delay shows similar results when compared to the five second delay test. After the target acquisition, the filter shows good velocity estimation close to the target's true velocity of five meters per second.

(b) *Circular Road Model*

The varying simulation parameters that were used for the third order road model are also used for the circular road model. The repeating five second PVNT delay results are discussed first.

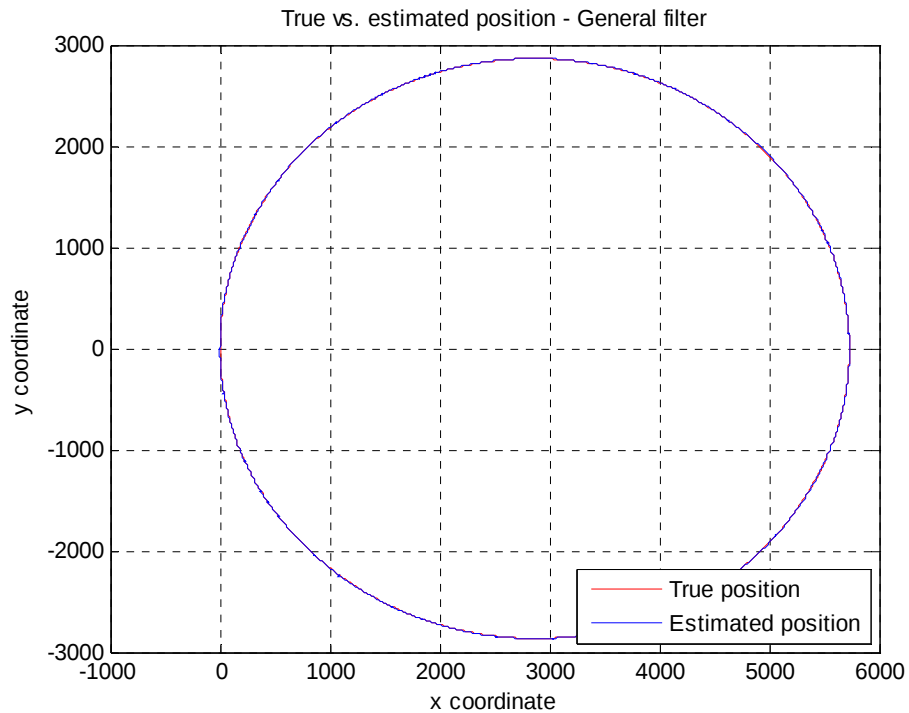


Figure 40. General filter position comparison plot – Circular road model – 5 second PVNT delay

The position comparison plot for the repeating five second PVNT delay simulation shows an estimated position that closely matches the true target position. It is necessary to view the position error vs. time plot, though, to see the true relationship due to the large sample time and axes scales.

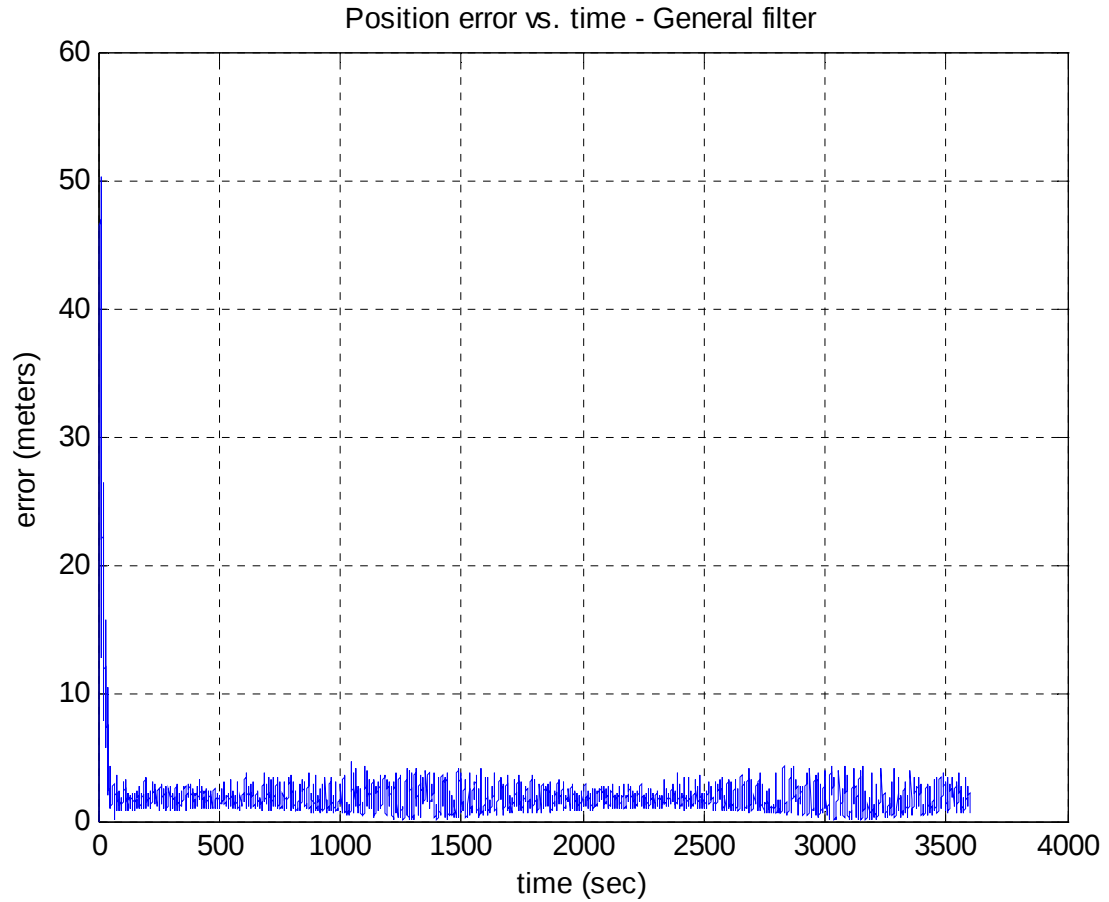


Figure 41. General filter position error plot – Circular road model – 5 second PVNT delay

As shown by the plot, the estimated position error from the real-time general filter remains largely unchanged with a five second PVNT delay when compared to the same trial under ideal conditions. After the initial acquisition period, the RMS error pertaining to the estimated position remains under five meters.

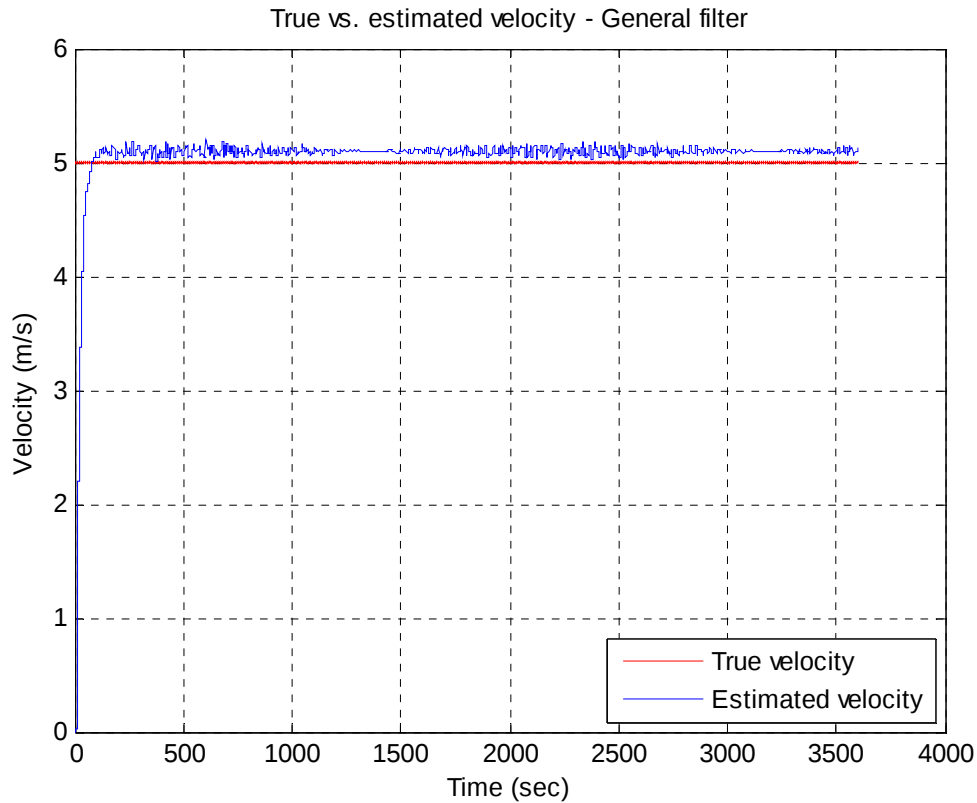


Figure 42. General filter velocity comparison plot – circular road model – 5 second PVNT delay

The velocity comparison plot shows an estimated velocity that is only slightly off of the true target’s five meter per second velocity. Based on the data from Figures 40, 41, and 42 and the circular road model simulation under ideal conditions, the real-time general filter does not lose any accuracy with the repeating five second PVNT delay.

Like the third order road model, the system using the circular road model is also tested at the upper limit of the expected PVNT delay:

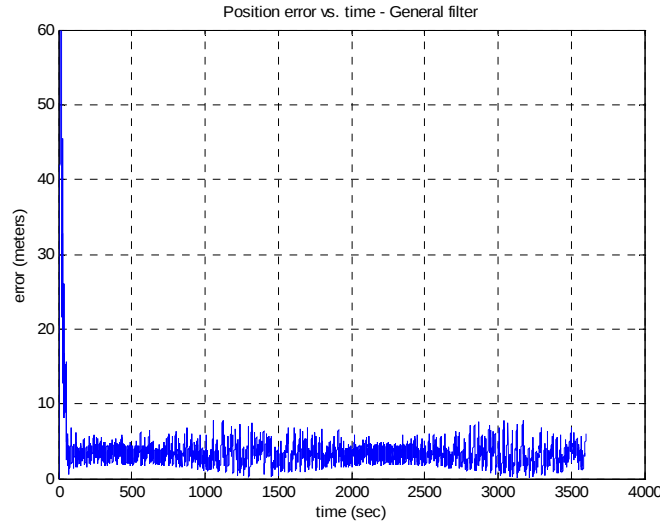


Figure 43. General filter position error plot – Circular road model – 10 second PVNT delay

The position error plot for the ten second PVNT delay simulation shows a slight increase in the estimated position error from the five second delay test. The plot in Figure 43 shows a peak error value of just less than eight meters compared to a maximum error of five meters for the five second PVNT delay error plot.

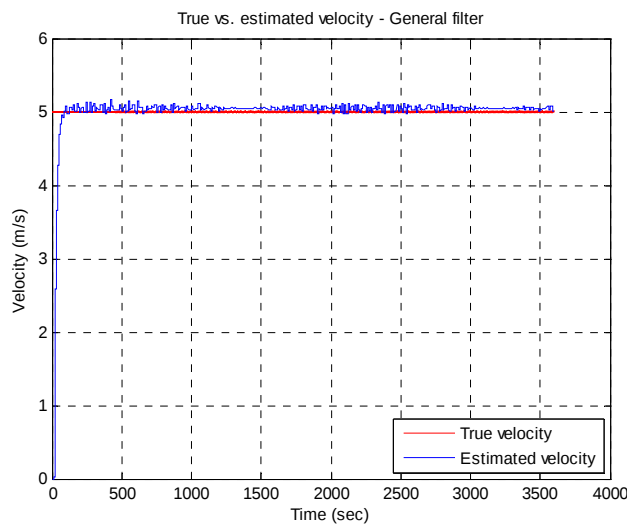


Figure 44. General filter velocity comparison plot – Circular road model – 10 second PVNT delay

The velocity comparison plot actually shows a slightly better target velocity estimate than the repeating five second delay simulation. This is one example of how the shape of the road affects the results of the simulation. While the third order road model showed no change in velocity estimation between the five and ten second PVNT delay tests, the circular road model actually showed an improvement due to its shape.

(3) PVNT Noise Variance. The final testing phase for the real-time general filter involved setting the PVNT delay back to the simulated pseudo-random delay time and adjusting the random number generator block controlling PVNT noise in the true target model subsystem block. While the ideal conditions had a PVNT noise value of ± 1 meter, the noise would be increased to ± 5 and ± 10 meters between the simulations.

(a) *Third Order Road Model*

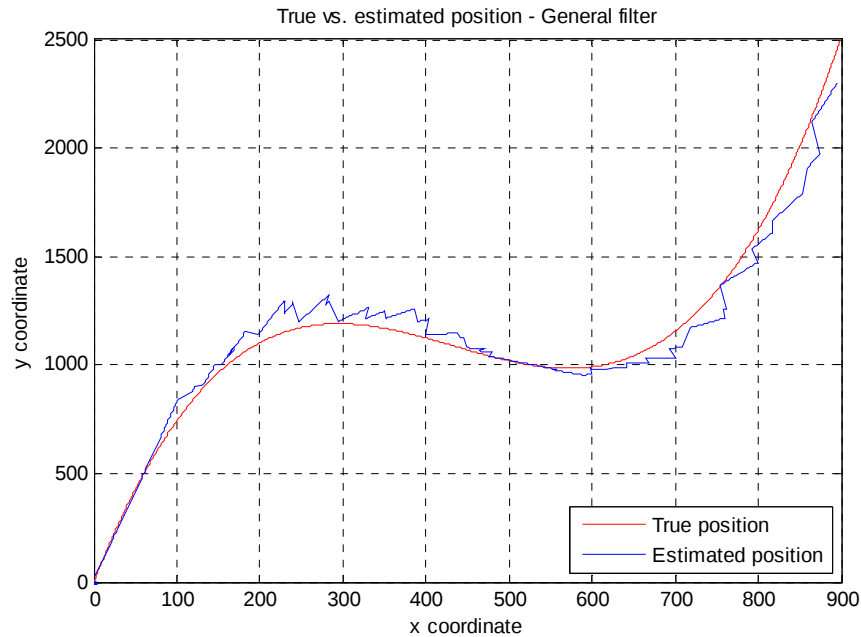


Figure 45. General filter position comparison plot – Third order road model – ± 5 m PVNT noise

First, the real-time general filter model is tested with a ± 5 meter PVNT noise and the results appear quite similar to the ideal conditions test. When comparing the position plots, a slight decrease in estimation accuracy is noticed as the position updates do not match up with the target's true position due to the extra PVNT noise.

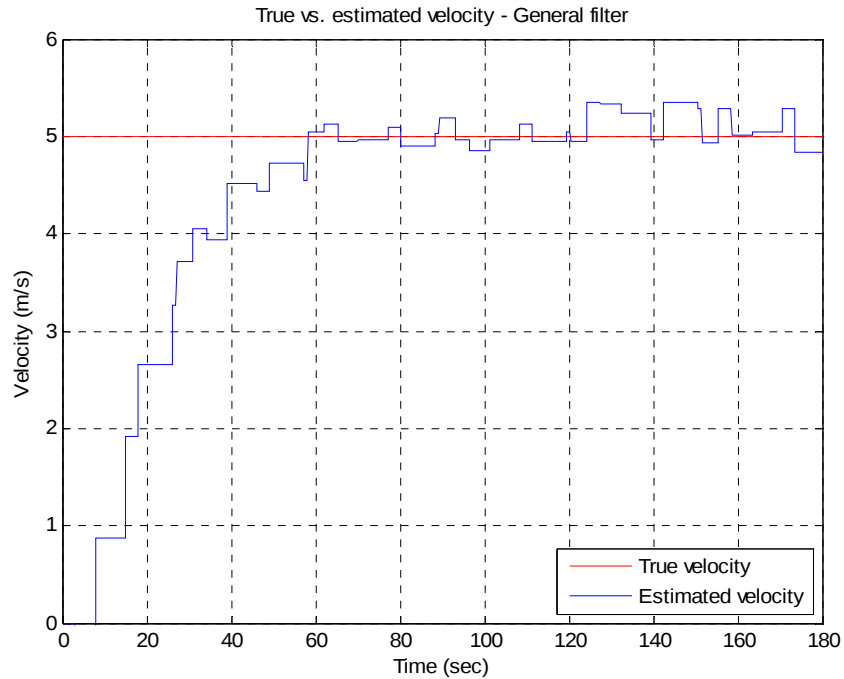


Figure 46. General filter velocity comparison plot – Third order road model – ± 5 m PVNT noise

The effects of the added PVNT noise are more noticeable in the velocity comparison plot due to the larger axes in the position comparison plot. The deviation between the true and estimated velocity is greater than the velocity difference found in the ideal conditions test.

The PVNT noise is then doubled to ± 10 m for the final set of tests for the third order road model using the real-time general filter design. This is very impractical as other filter designs with PVNT updates can boast ten meter accuracy, but it is important to show how much the filter can attempt to compensate to the inputted error [6].

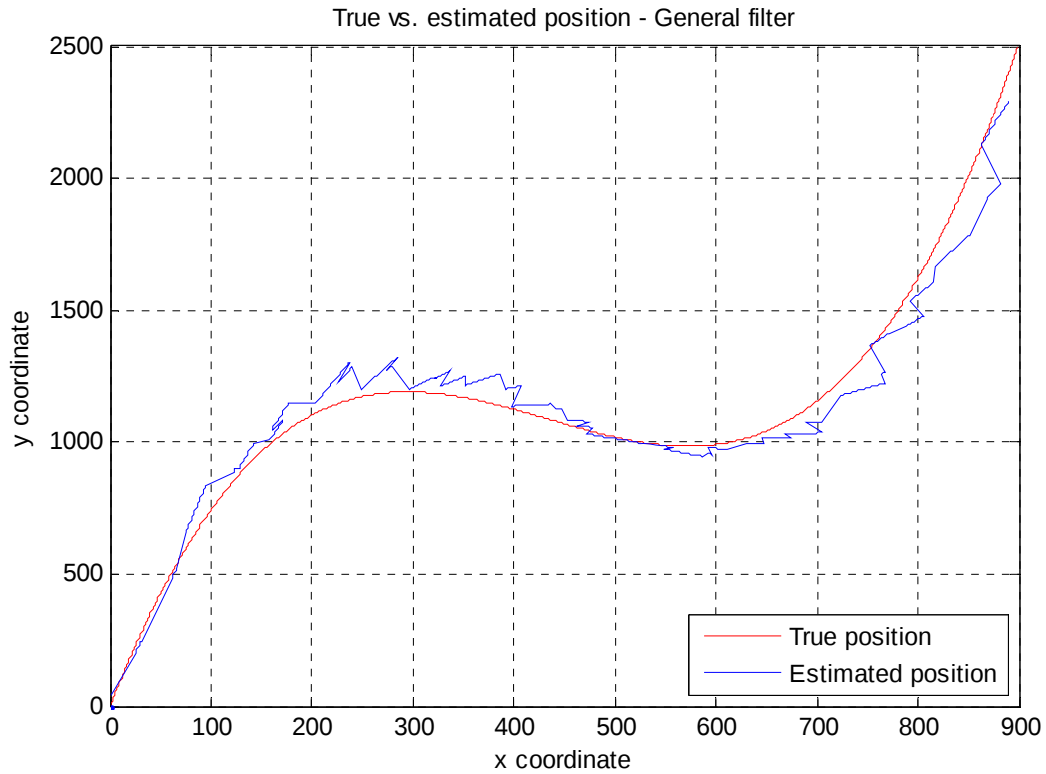


Figure 47. General filter position comparison plot – Third order road model – ± 10 m PVNT noise

Even with a PVNT noise value having a ten meter variance in either direction, the real-time general filter shows little change from the five meter PVNT noise simulation. While the overall accuracy does have room for improvement, there is minimal change in position estimation accuracy between the five and ten meter PVNT noise tests.

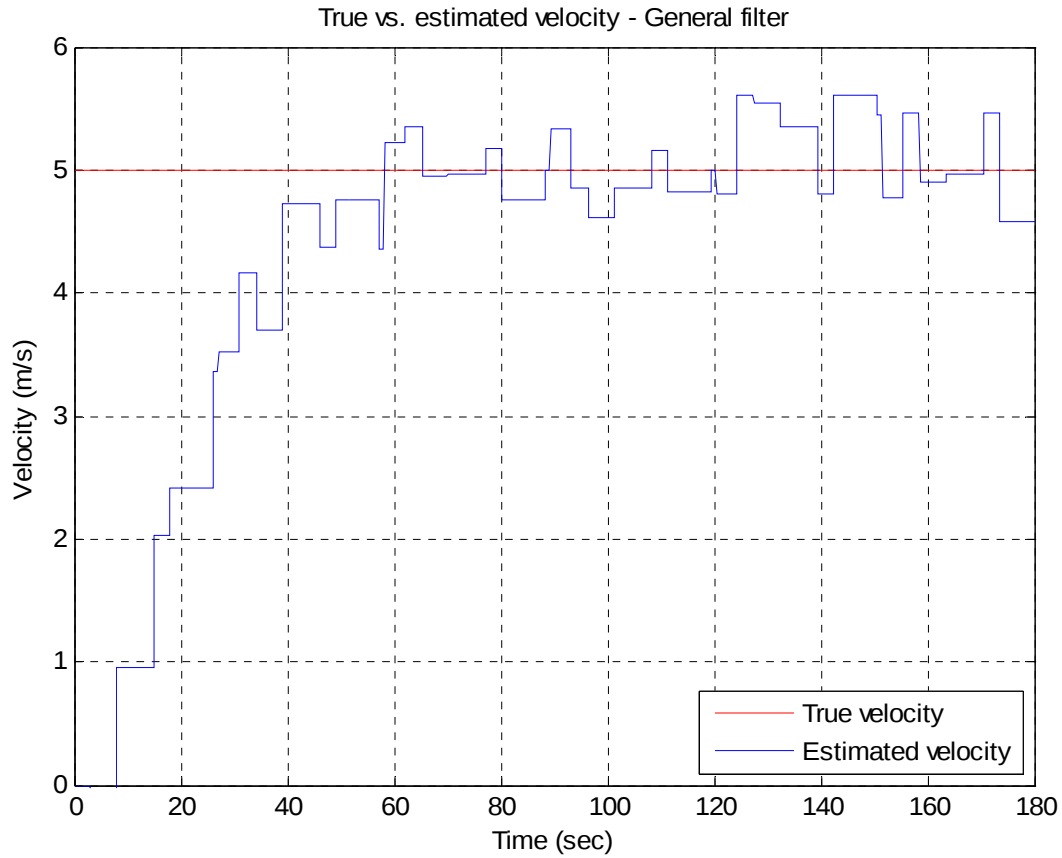


Figure 48. General filter velocity comparison plot – Third order road model – ± 10 m PVNT noise

The velocity comparison plot shows an estimated steady state velocity that is always within 0.6 m/s of the true target velocity. While the ± 5 meter test had a maximum error of 0.35 m/s, the test with the doubled PVNT input error shows less than a twofold increase in velocity estimation error.

(b) Circular Road Model

The circular road model is put through the same tests for PVNT noise variance as the third order road model for the real-time general filter system.

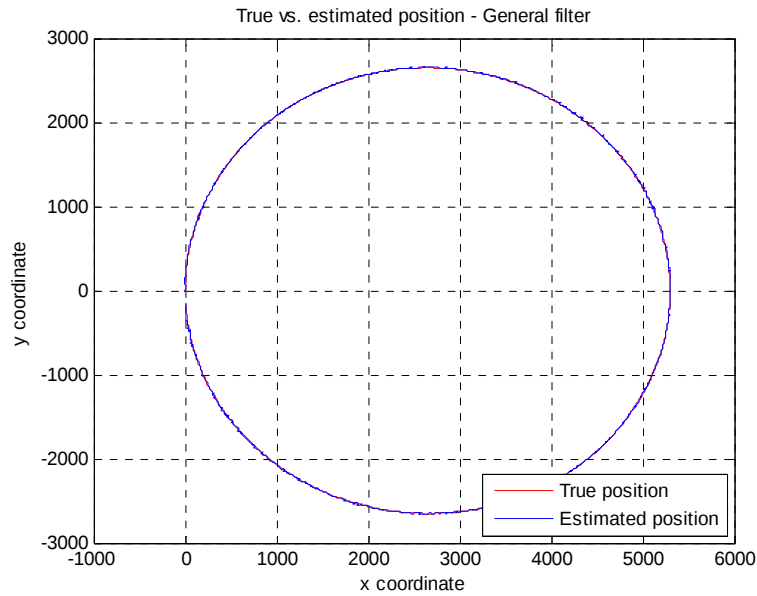


Figure 49. General filter position comparison plot – Circular road model – ± 5 m PVNT noise

The added PVNT noise seems to have a minimal effect on the real-time general filter running the circular road model but a look at the position error plot is required due to the large axes scale.

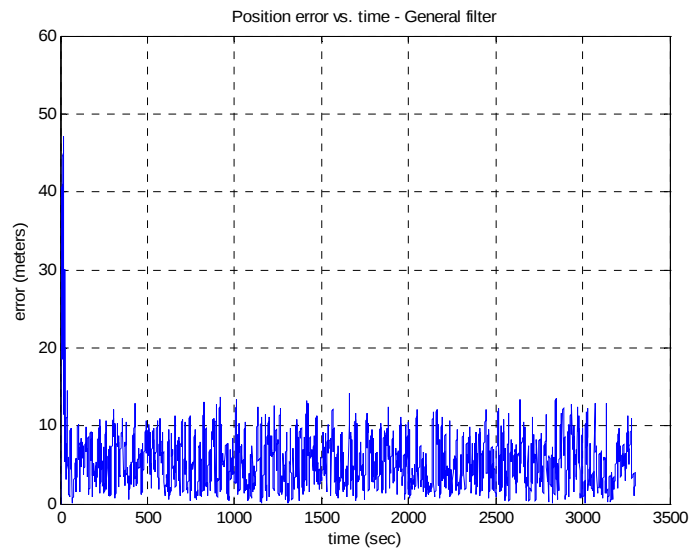


Figure 50. General filter position error plot – Circular road model – ± 5 meter PVNT noise

Figure 50 shows that the added noise from the PVNT input results in an estimated position error along the circular road model that is more than double that of the ideal conditions test. The real-time general filter shows just how dependent it is on the accuracy of the PVNT position update since it does not utilize the road equation in its calculations.

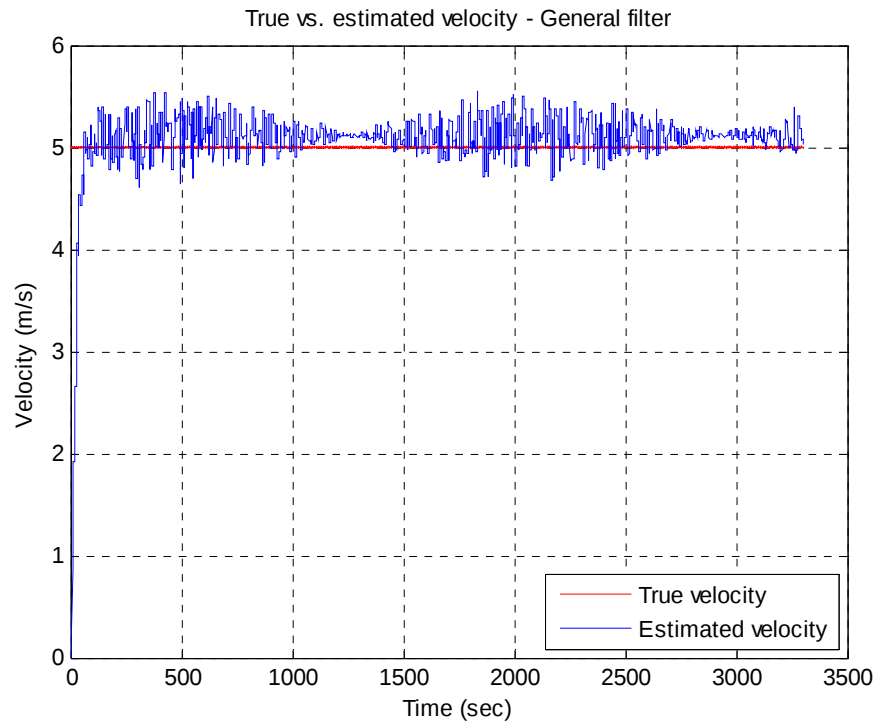


Figure 51. General filter velocity comparison plot – Circular road model – ± 5 meter PVNT noise

The velocity comparison plot shows an increase as well due to the extra PVNT input noise. There is quite a large change when compared to the velocity plot for the circular model under ideal conditions. While the ideal test resulted in a maximum estimated velocity of 5.2 m/s, the test with the PVNT noise pushed the maximum estimated velocity to over 5.5 m/s. The modular shape of the velocity estimation is due to the shape of the road profile. There are certain points in the model where there is only velocity error in the x or y direction as opposed to both the x and y directions.

Finally, the real-time general filter design using the circular road model is simulated with a ± 10 meter PVNT input noise and the results are analyzed.

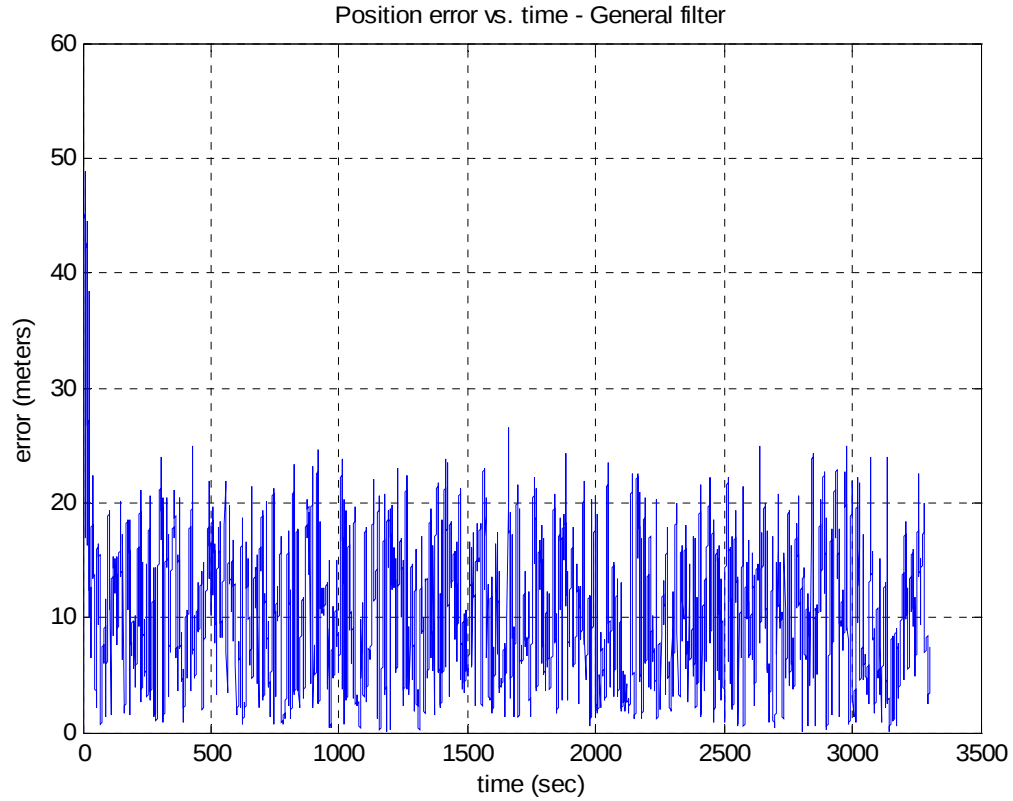


Figure 52. General filter position error plot – Circular road model – ± 10 meter PVNT noise

As expected, the RMS error increased with the doubled PVNT noise as shown in Figure 52. The peak error is just over 27 meters at around 1660 seconds into the simulation.

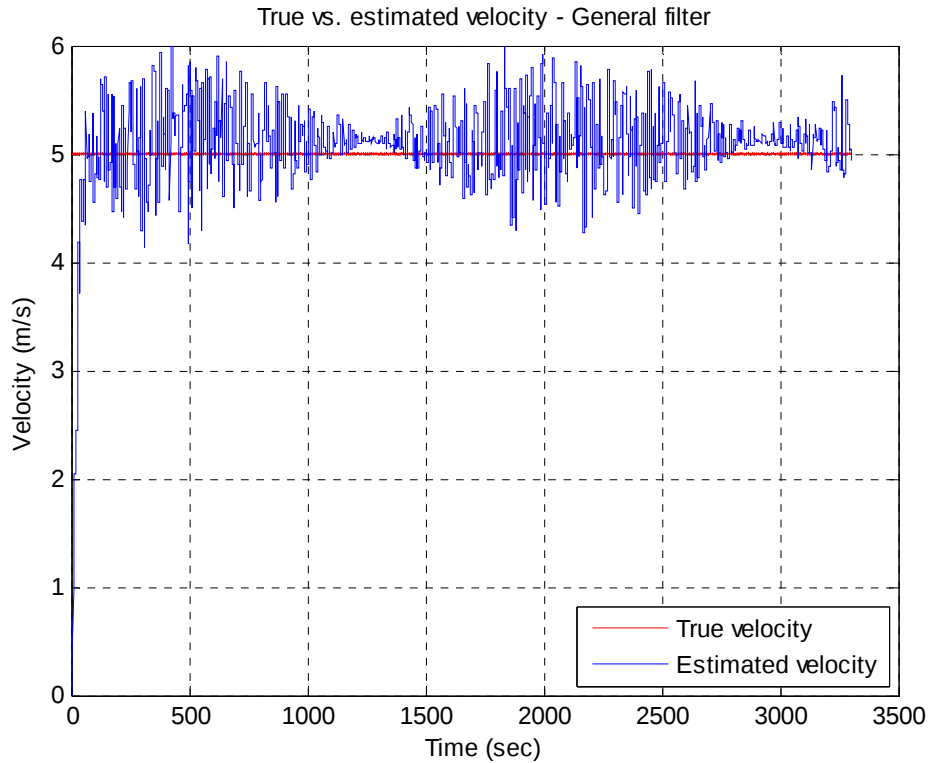


Figure 53. General filter velocity comparison plot – Circular road model – ± 10 meter PVNT noise

The velocity comparison plot for the real-time general filter during this trial shows a maximum absolute error of around one meter per second. The figure further shows that the velocity error has a direct relationship to PVNT input error.

b. Road Following Filter

The system parameters for the tests involving the real-time road following filter are identical to those performed with the real-time general filter

(1) **Ideal Conditions.** Ideal conditions are defined as a PVNT noise value covering a range of ± 1 meter and a simulated random PVNT delay.

(a) *Third Order Road Model*

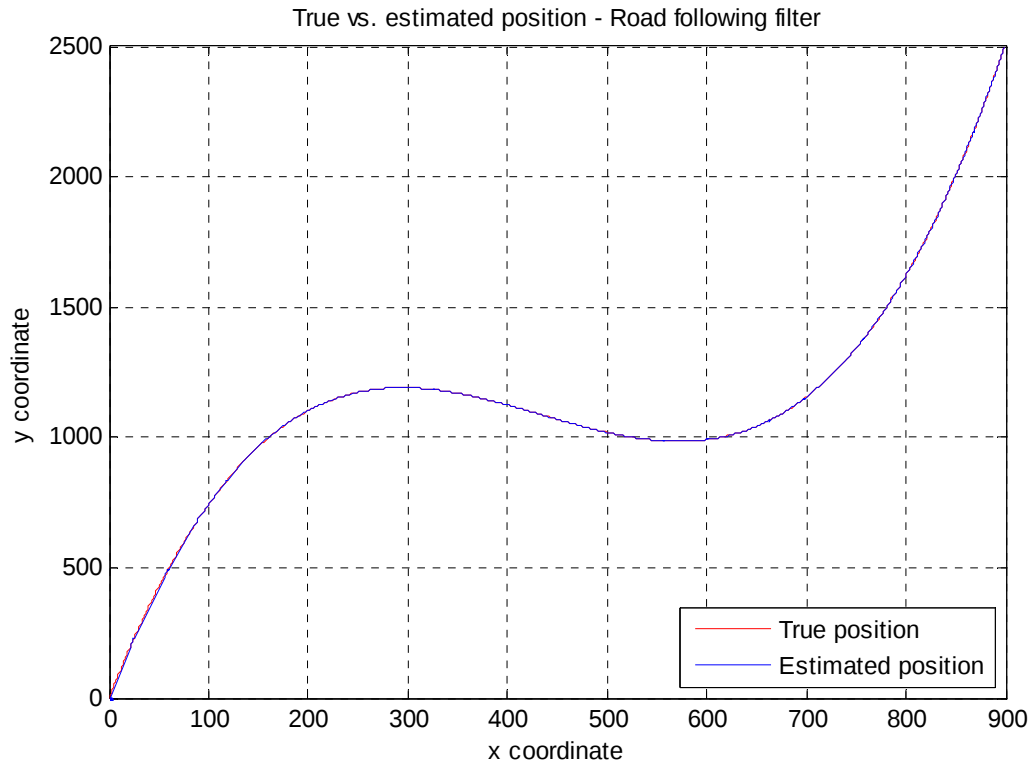


Figure 54. Road following filter position comparison plot – Third order road model – Ideal conditions

The results for the real-time road following filter using the third order road model under ideal conditions appear exponentially more accurate than the position comparison plot for the real-time general filter design under the same conditions. To confirm these results, the position error vs. time plot is examined:

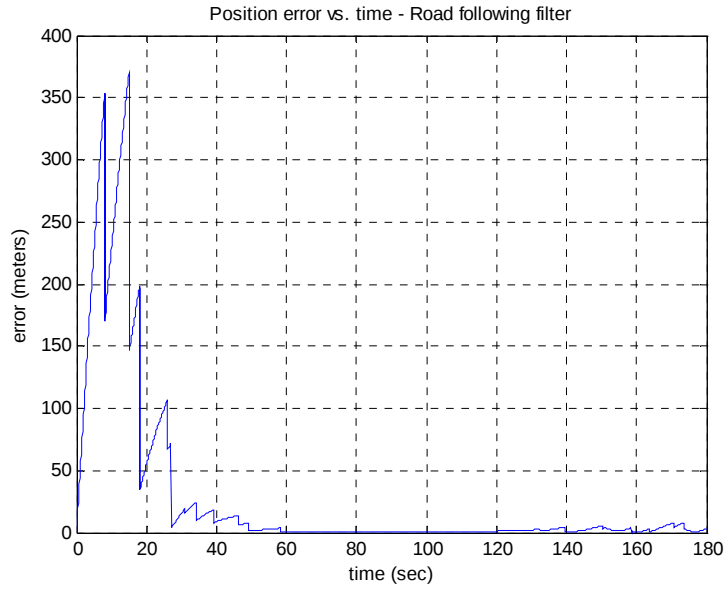


Figure 55. Road following filter position error plot – Third order road model – Ideal conditions

The position error plot shows that after the initial target acquisition time, the real-time filter is able to estimate a target position that has less than a ten meter deviation from the actual position. During the straighter sections of the road, the position estimation is even more accurate with the error dropping to less than one meter.

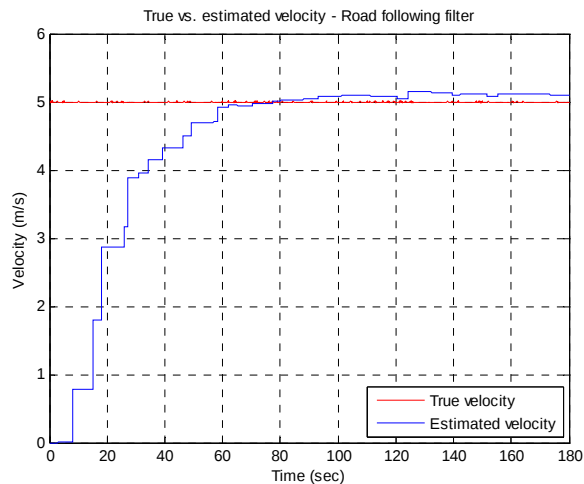


Figure 56. Road following filter velocity comparison plot – Third order road model – Ideal conditions

The velocity comparison plot is actually quite similar to the results from the real-time general filter. The low steady state error for estimated velocity is confirmed by Figure 57:

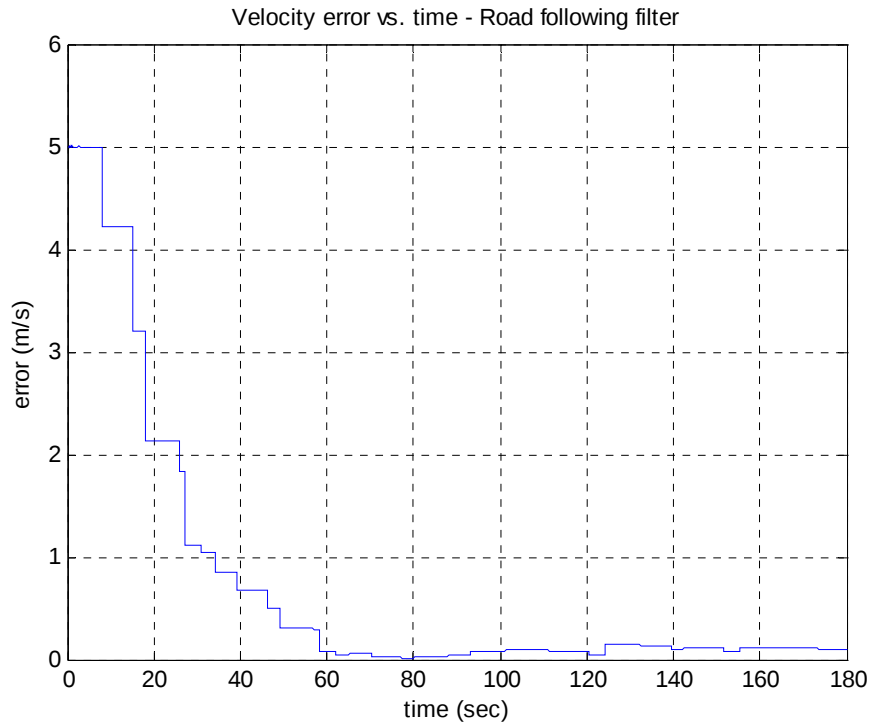


Figure 57. Road following filter velocity error plot – Third order road model – Ideal conditions

The velocity error plot is nearly identical to the real-time general filter velocity error plot for the same conditions. Following the target acquisition period, the absolute velocity error remains less than 0.2 m/s.

Additionally, the road following filter utilizes the road parameter ρ in the equations that track target movement. This allows the variance in the estimated and actual ρ value to be plotted as well. Throughout all of the tests, the true target ρ value linearly increases with time as seen in the next figure.

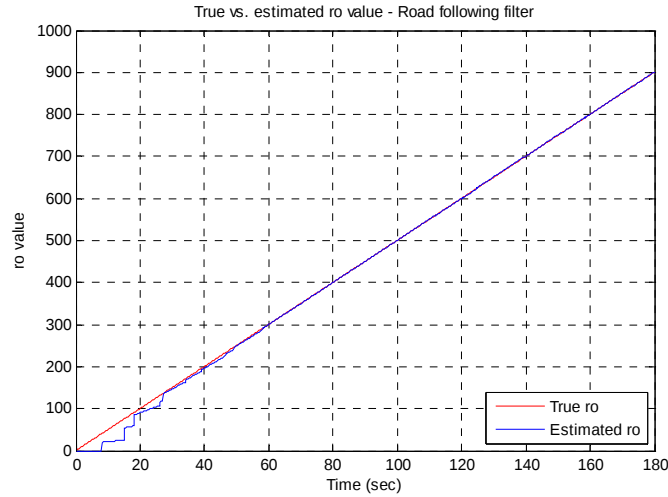


Figure 58. Road following filter ρ comparison plot – Third order road model – Ideal conditions

The difference in the estimated and actual ρ value is kept to a minimum by the real-time road following filter. The system is able to accurately estimate the ρ value through the asynchronous forward Euler integration process coupled with the optimized PVNT position input.

(c) Circular Road Model

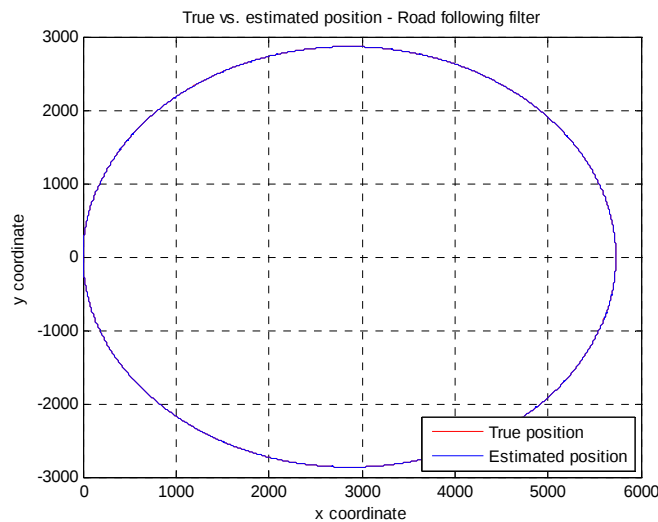


Figure 59. Road following filter position comparison plot – Circular road model – Ideal conditions

The position estimation for the real-time road following filter using the circular road model appears very accurate and the position error vs. time plot is examined:

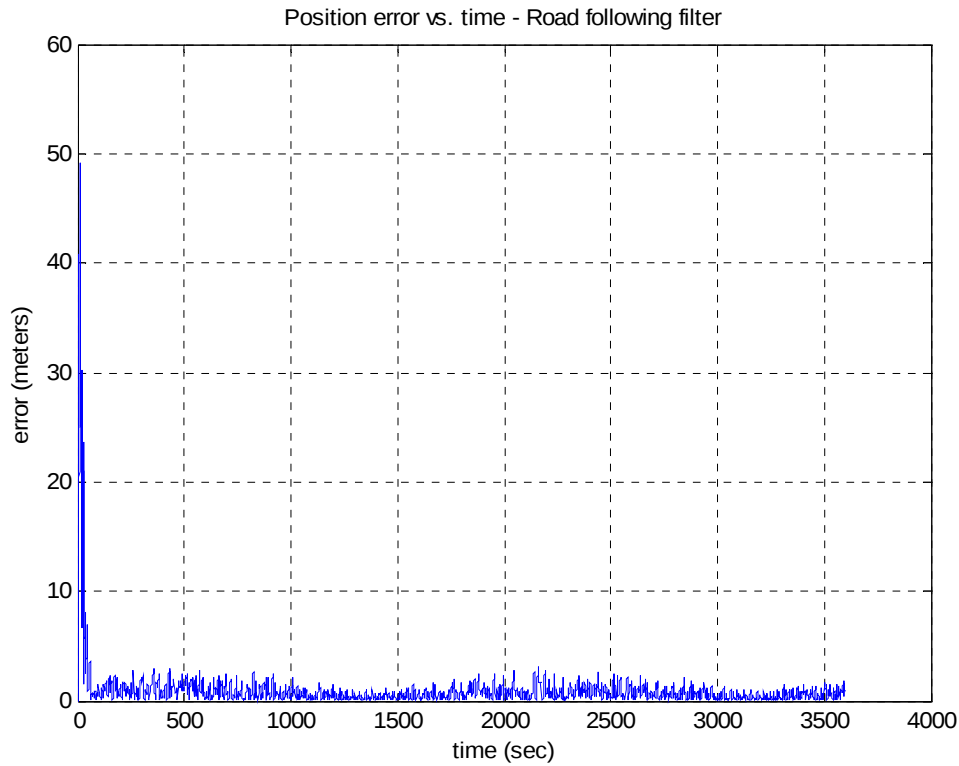


Figure 60. Road following filter position error plot – Circular road model – Ideal conditions

The real-time road following filter with its added PVNT optimization function decreases the estimated error during the simulation. There is a noticeable difference when compared to the general filter simulation using the circular road model under ideal conditions. While the real-time general filter had a maximum absolute error of five meters, the real-time road following filter only had a maximum absolute error of three meters.

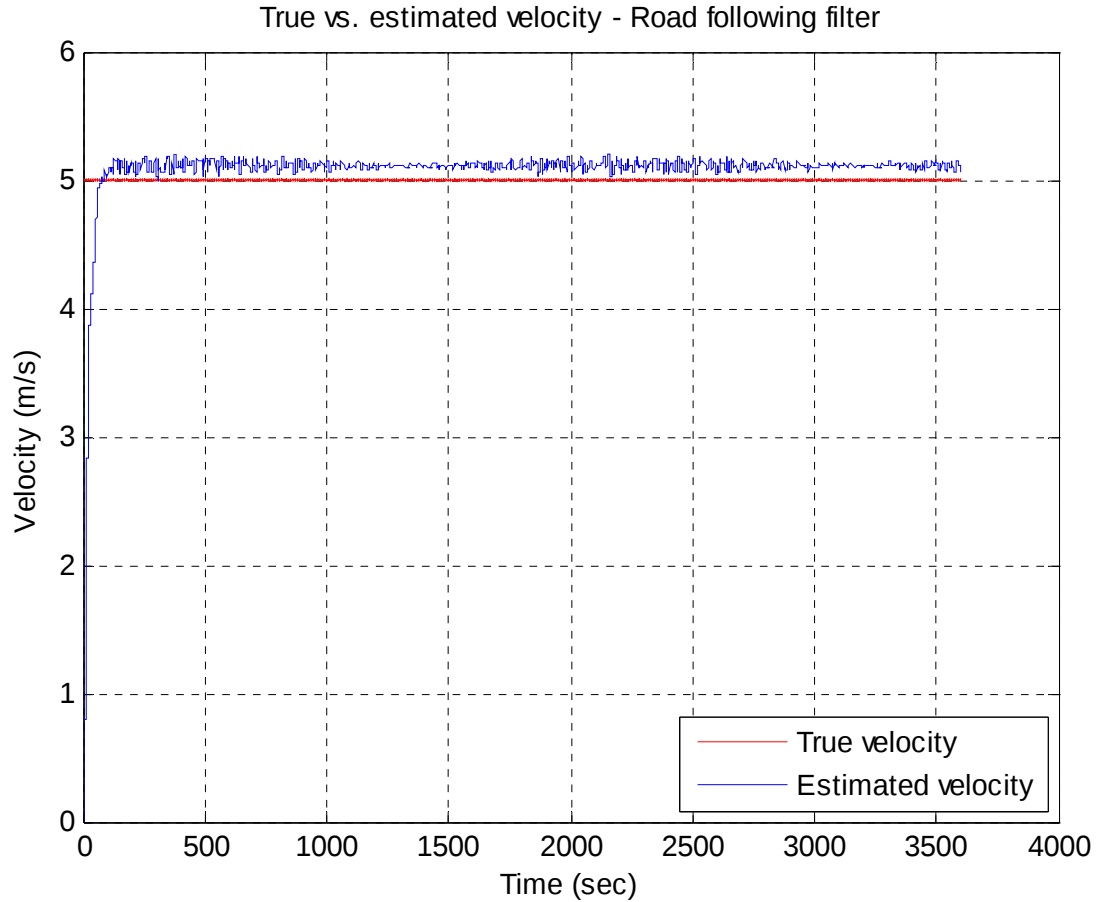


Figure 61. Road following filter velocity comparison plot – Circular road model – Ideal conditions

The velocity comparison plot shows the estimated velocity using the real-time road following filter to be very similar to the results from the real-time general filter. It is noticed that both filters have a slight steady state velocity error during the simulations even though the precision is good.

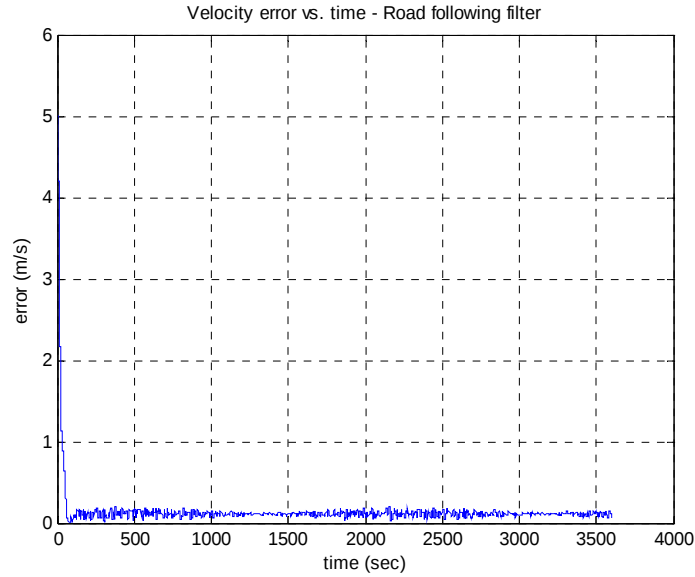


Figure 62. Road following filter velocity error plot – Circular road model – Ideal conditions

The velocity error plot for the real-time road following filter using the circular road model is nearly identical to the third order road model. The maximum absolute estimated velocity error is never more than 0.2 m/s following the target acquisition time.

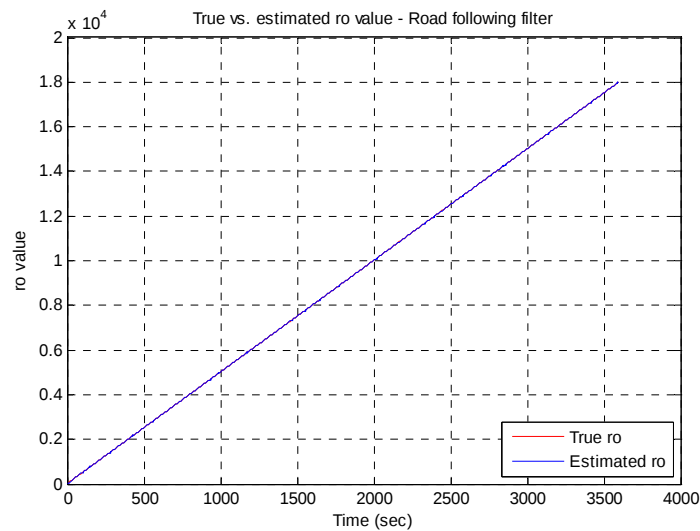


Figure 63. Road following filter ρ comparison plot – Circular road model – Ideal conditions

The ρ comparison plot coincides with the estimated position and velocity plots, showing minimal estimation error throughout the hour long simulation.

(2) PVNT Update Delay Variance. The next testing phase for the real-time road following filter is to adjust the delay time from the PVNT update signal subsystem. Instead of using the simulated pseudo-random update signal, a signal generator block is used to simulate a repeating five and ten second PVNT position update delay.

(a) *Third Order Road Model*

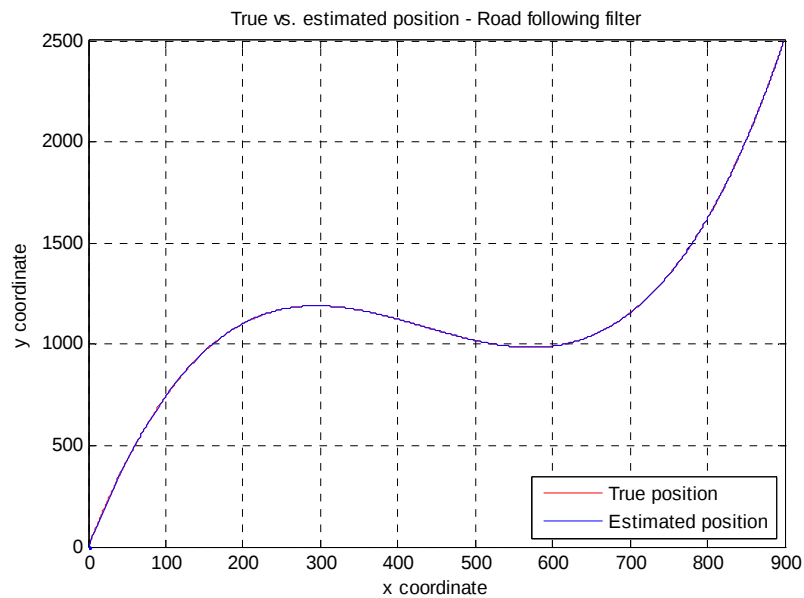


Figure 64. Road following filter position comparison plot – Third order road model – 5 second PVNT delay

The expected PVNT delay can be as long as ten seconds so the five second PVNT delay does not affect the position tracking results. The velocity comparison plot below depicts similar results:

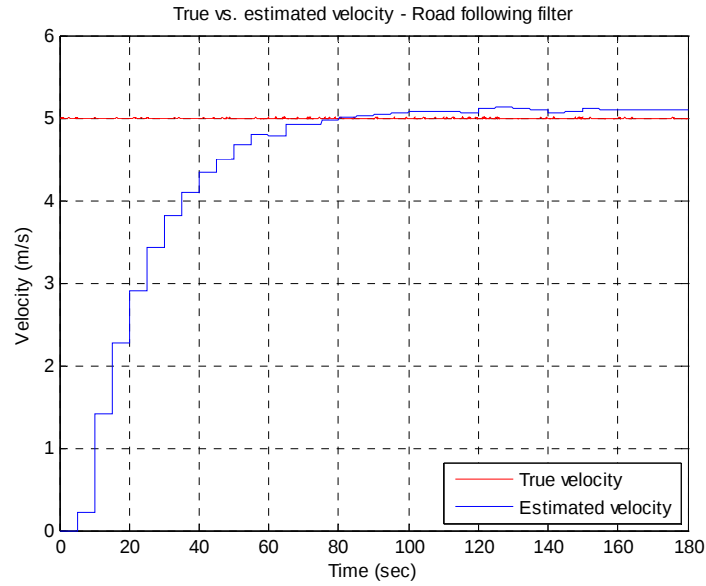


Figure 65. Road following filter velocity comparison plot – Third order road model – 5 second PVNT delay

Figure 65 shows the estimated target velocity from the real-time road following filter plotted against the actual target velocity. The results are nearly identical to the ideal conditions plot shown in Figure 56.

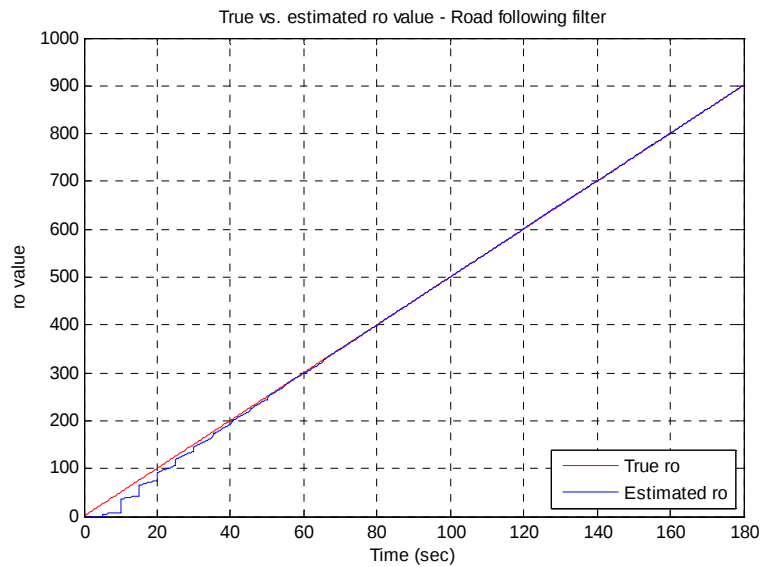


Figure 66. Road following filter ρ comparison plot – Third order road model – 5 second PVNT delay

The ρ comparison plot depicts an estimated ρ value that achieves a near zero steady state error. The repeating five second PVNT delay can be viewed during the first 30 seconds of the test as each update brings the estimated ρ value closer to the target's true ρ value.

The next step involves doubling the PVNT delay to ten seconds for the third order road model.

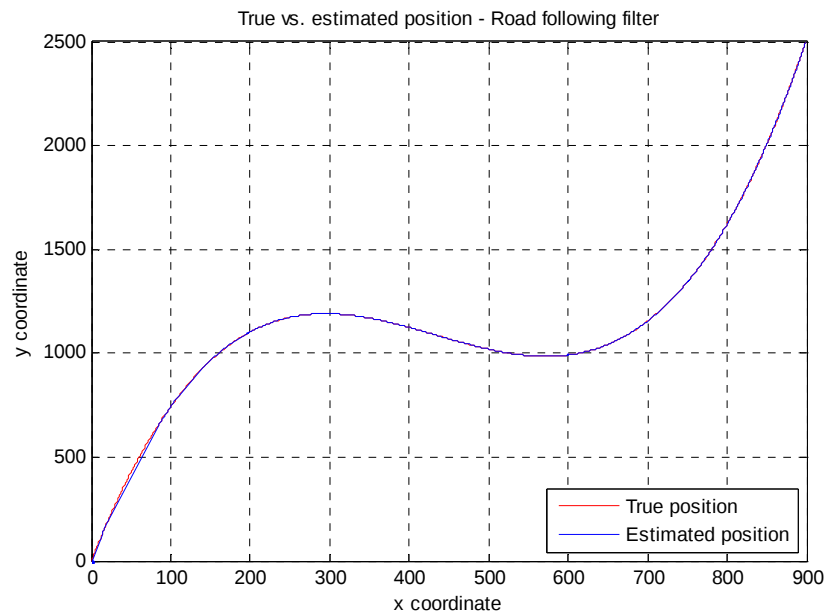


Figure 67. Road following filter position comparison plot – Third order road model – 10 second PVNT delay

The position comparison plot for the repeating ten second PVNT delay test shows the robustness of the real-time road following filter with longer delay times. As long as the inputted PVNT update has little noise, the optimization function ensures the accuracy of the new ρ value sent to the asynchronous forward Euler integration function. This results in a more accurate target tracking model even though the frequency of the updates has decreased.

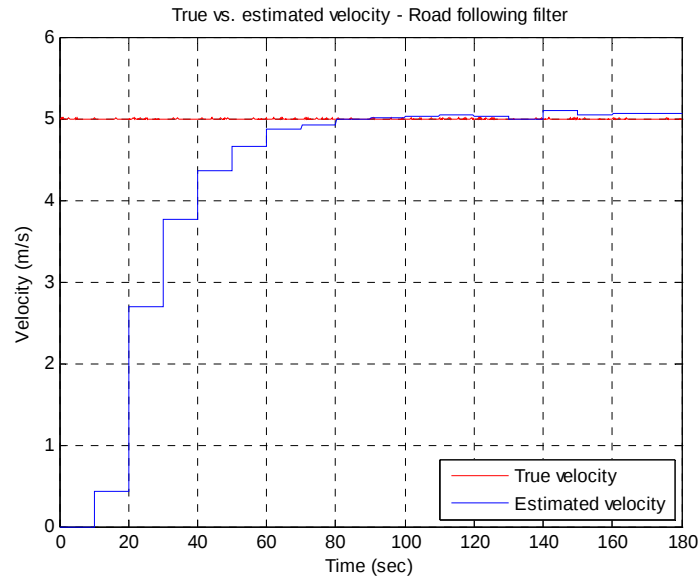


Figure 68. Road following filter velocity comparison plot – Third order road model – 10 second PVNT delay

The velocity comparison plot, like the position comparison plot, shows little or no change from the additional five seconds of PVNT update delay. The maximum absolute error of the velocity estimation remains the same following target acquisition.

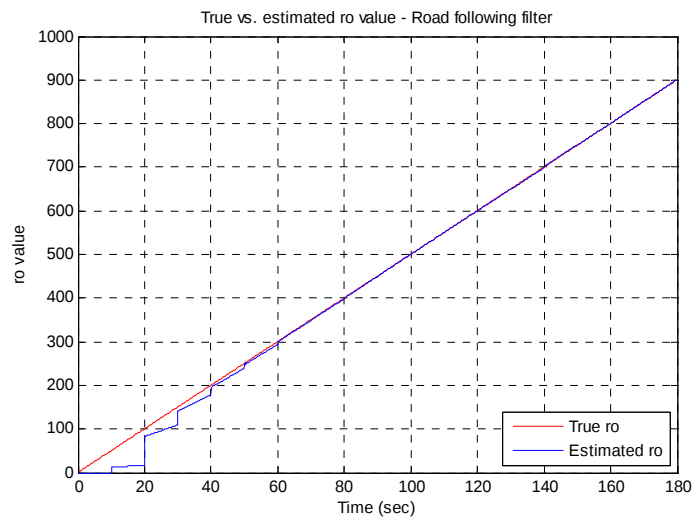


Figure 69. Road following filter ρ comparison plot – Third order road model – 10 second PVNT delay

The ρ comparison plot in the figured above is best viewed next to the ρ comparison plot for the repeating five second PVNT delay test. Even though the delay for the position updates is twice as long, the accuracy of the ρ estimate is aided by the optimization routine in the S-function. While the settling time increases slightly, the overall steady state accuracy is not affected by the increased PVNT delay.

(b) *Circular Road Model*

The same PVNT delay trials are performed with the real-time road following filter using the circular road model.

Initial impressions of the position comparison plot for the circular road model trial with a repeating five second PVNT update delay are good but the depiction of the estimated position error plot is shown below due to the large sample time and figure axes.

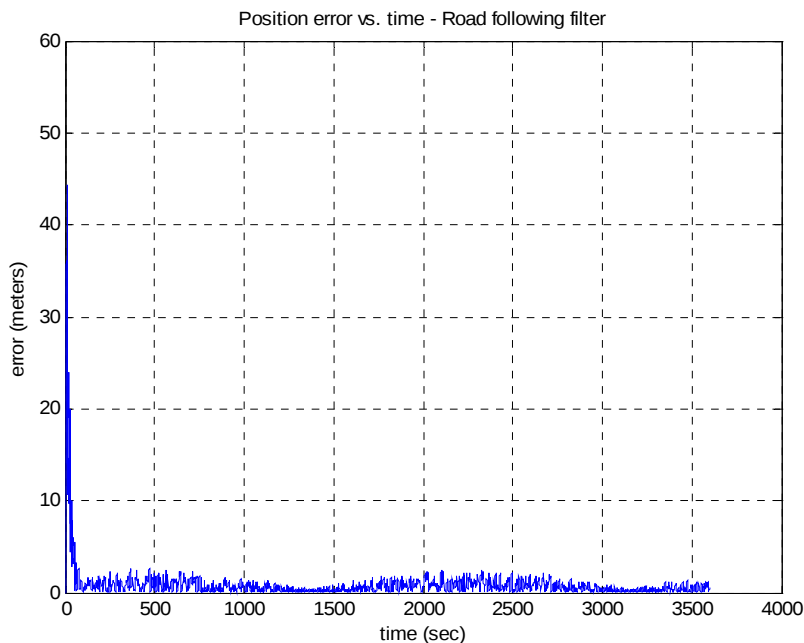


Figure 70. Road following filter position error plot – Circular road model – 5 second PVNT delay

The plot of estimated position error vs. time shows the results of the simulation with a repeating five second PVNT update delay are no different from the simulation under ideal conditions. The maximum absolute position errors are identical between the two trials as the repeating delay shows no effect on the road following filter using the circular road model.

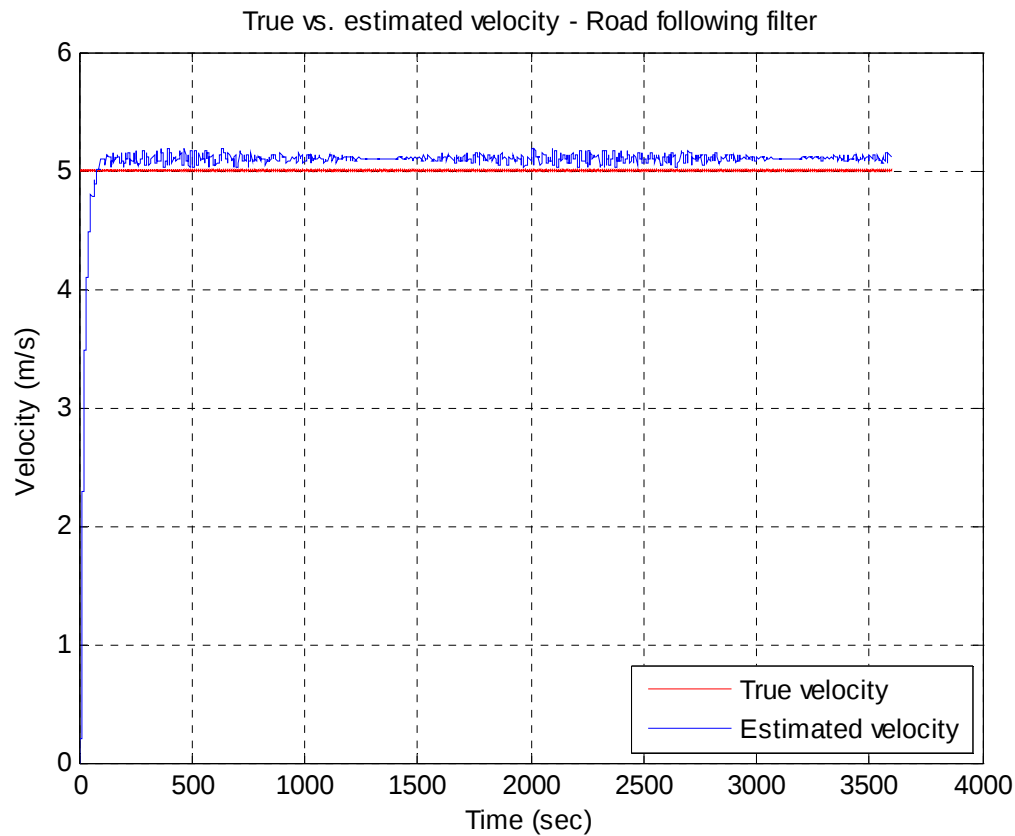


Figure 71. Road following filter velocity comparison plot – Circular road model – 5 second PVNT delay

The velocity comparison plot shown above for the real-time road following filter mimics the results for the position error comparison plot. No change is seen between the velocity estimation accuracy between the repeating five second delay and ideal conditions trials.

The next step involves doubling the PVNT delay to ten seconds for the circular road model.

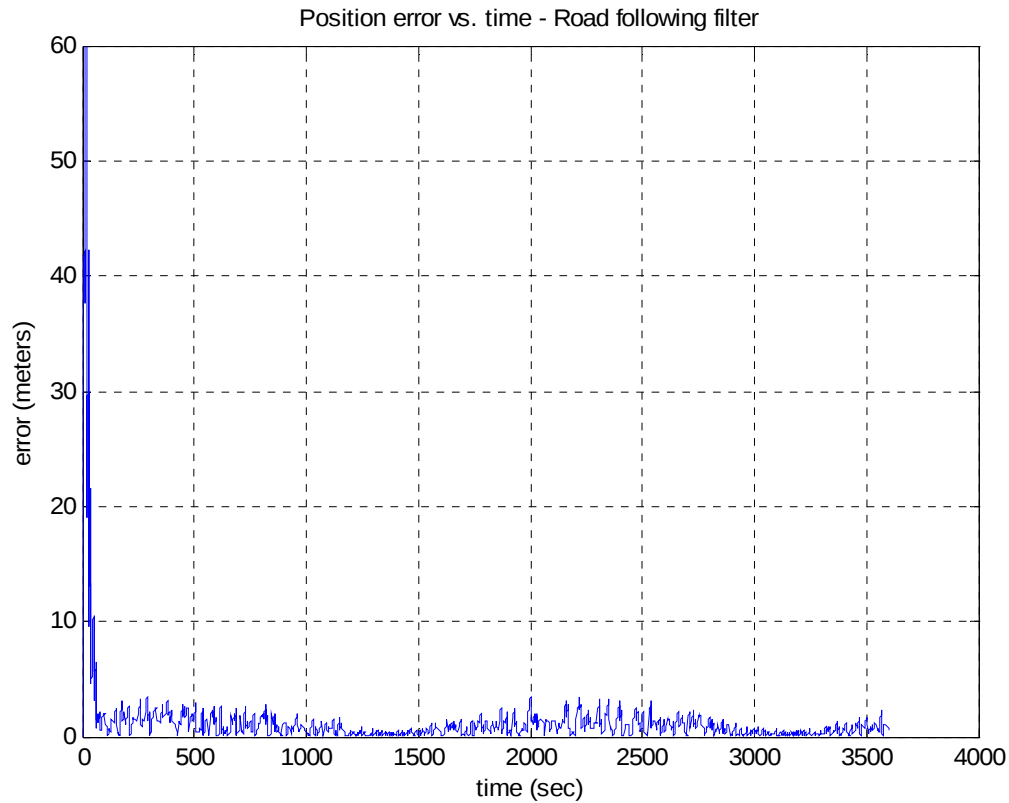


Figure 72. Road following filter position error plot – Circular road model – 10 second PVNT delay

Figure 72 depicts the difference between the estimated and actual target position with a repeating ten second PVNT delay. Even when the PVNT delay is set at its upper expected limit, the real-time model only loses one meter of accuracy during the hour long trial. Once again, the addition of the optimization function to the real-time road following filter’s code aids the robustness of the system with respect to longer periods of time between PVNT position updates.

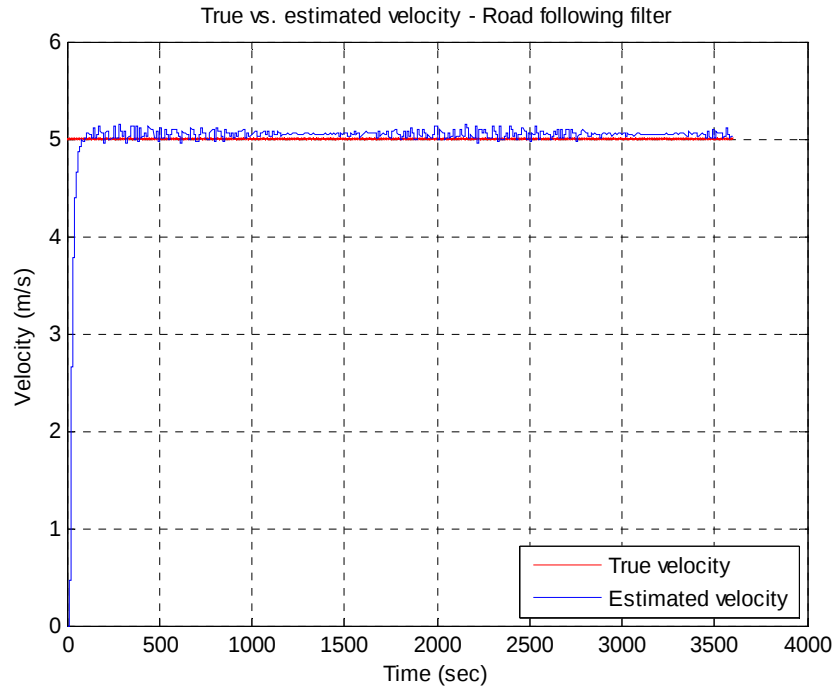


Figure 73. Road following filter velocity comparison plot – Circular road model – 10 second PVNT delay

The velocity comparison plot for the real-time road following filter with a repeating ten second delay also shows no decrease in accuracy throughout the trial. It is interesting to note that, similar to the same trial for the real-time general filter, the steady state error of the estimated velocity actually decreases with the increase in PVNT delay time. While each real-time filter's performance is mainly due to the code within their respective S-functions, the shape of the road model also plays a role in the accuracy of the target motion estimation.

(3) PVNT Noise Variance. The final testing phase for the real-time road following filter involves setting the PVNT delay back to the simulated pseudo-random update and adjusting the random number generator block controlling PVNT noise in the true target model subsystem block. While the ideal conditions had a PVNT noise value of ± 1 meter, the noise would be increased to ± 5 and ± 10 meters.

(a) *Third Order Road Model*

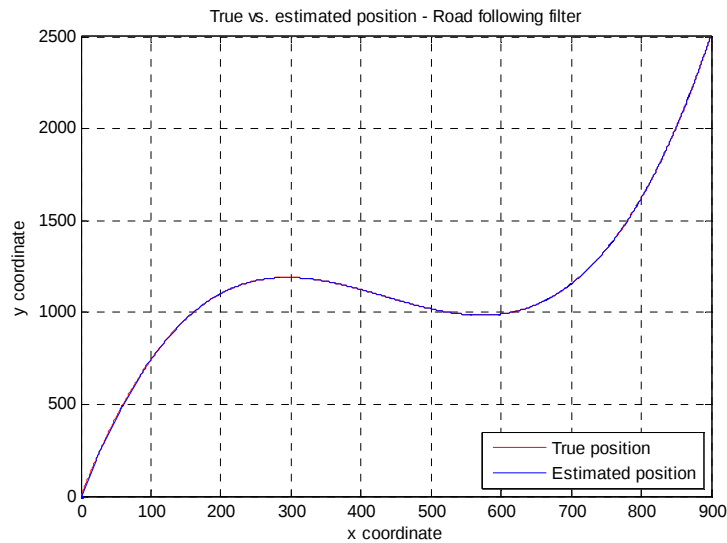


Figure 74. Road following filter position comparison plot – Third order road model – ± 5 m PVNT noise

Figure 74 shows that the real-time road following filter is still able to quite accurately track the target model with the additional PVNT input noise. The five meter variance is not enough to see any noticeable differences in position estimation precision.

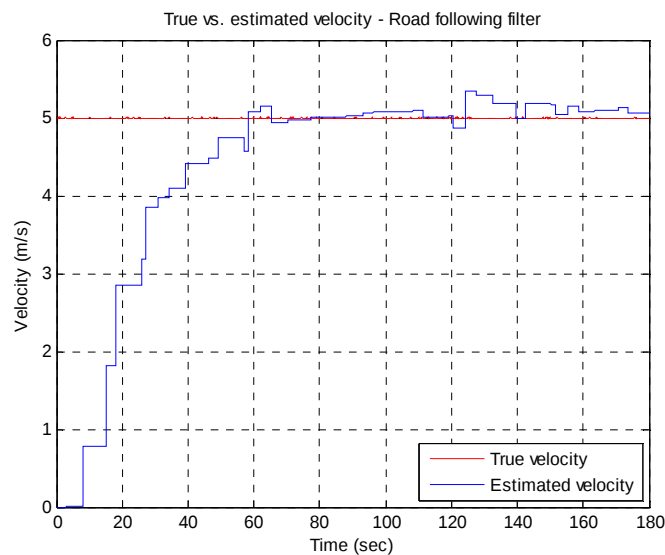


Figure 75. Road following filter velocity comparison plot – Third order road model – ± 5 m PVNT noise

Similar to the position comparison plot, the figure above shows a very slight variance in estimated velocity error during the curved portions of the road. The small change in velocity estimation accuracy, however, does not seriously affect the performance of target tracking.

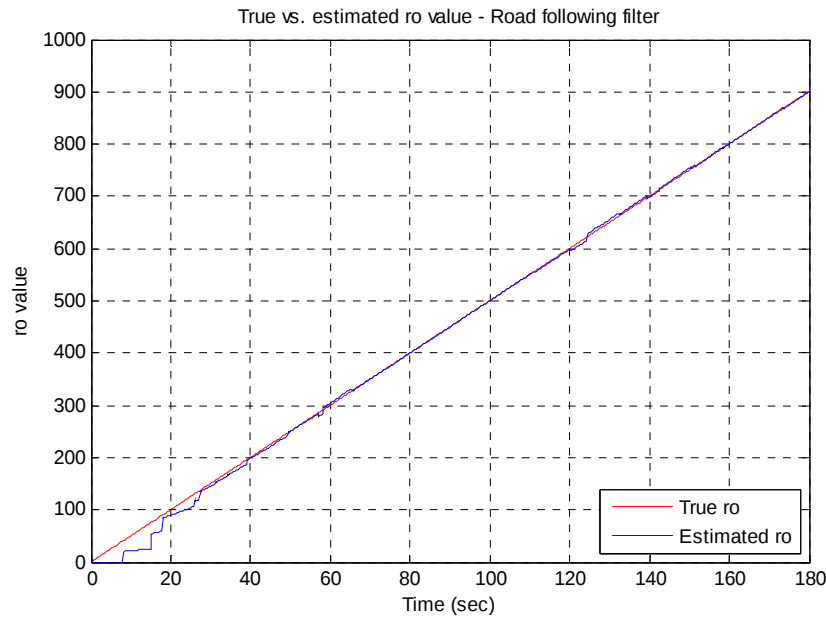


Figure 76. Road following filter ρ comparison plot – Circular road model – ± 5 m PVNT noise

The ρ comparison plot shows that the overall accuracy of the real-time filter remains unchanged except for slight errors around one and two minutes into the simulation. When compared to the position plot, it is found that these times correspond with the major areas of greatest curvature in the road model.

The PVNT noise is then doubled to ± 10 m for the final set of tests for the third order road model using the real-time road following filter design.

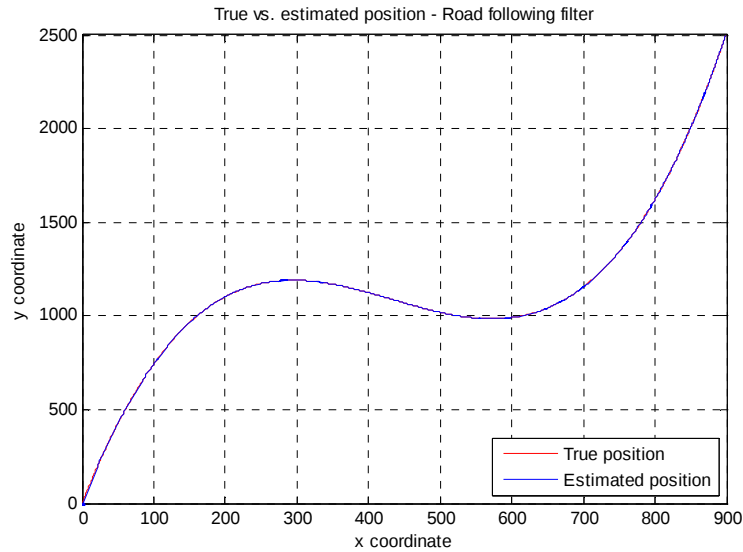


Figure 77. Road following filter position comparison plot – Third order road model – ± 10 m PVNT noise

The effects of the added PVNT noise still not quite noticeable in the position comparison plot even after the maximum variance of the PVNT input error is doubled. The system still appears to track the target without a significant drop in accuracy.

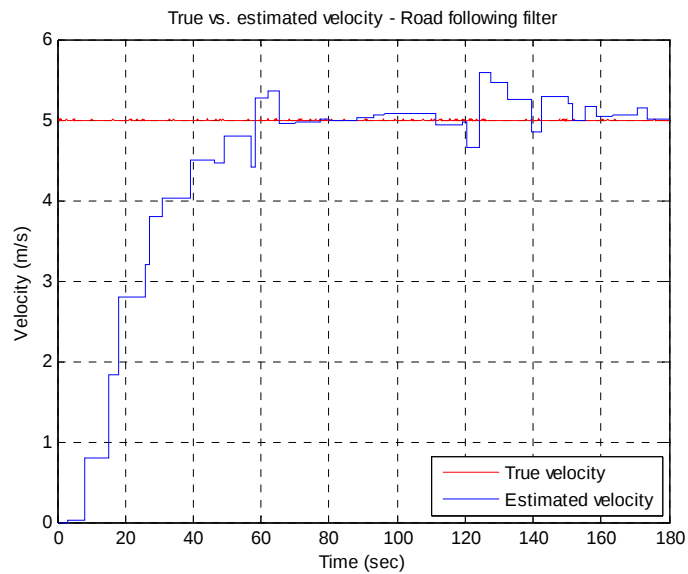


Figure 78. Road following filter velocity comparison plot – Third order road model – ± 10 m PVNT noise

The velocity comparison plot with the results from the trial with ± 10 meters of PVNT noise finally shows the effects on the system. The optimization function located in the road following filter's C code takes the x , y , z coordinate input from the PVNT update and finds the closest point on the pre-known road model to the PVNT input. The optimization loop ensures that the new position update lies along the road model by converting the new x , y , z coordinates into a ρ value, but it cannot guarantee the accuracy of the new estimated ρ value. Therefore, while the robustness of the real-time road following filter with respect to PVNT noise is better than the real-time general filter, the target motion estimation accuracy still decreases with larger amounts of input noise.

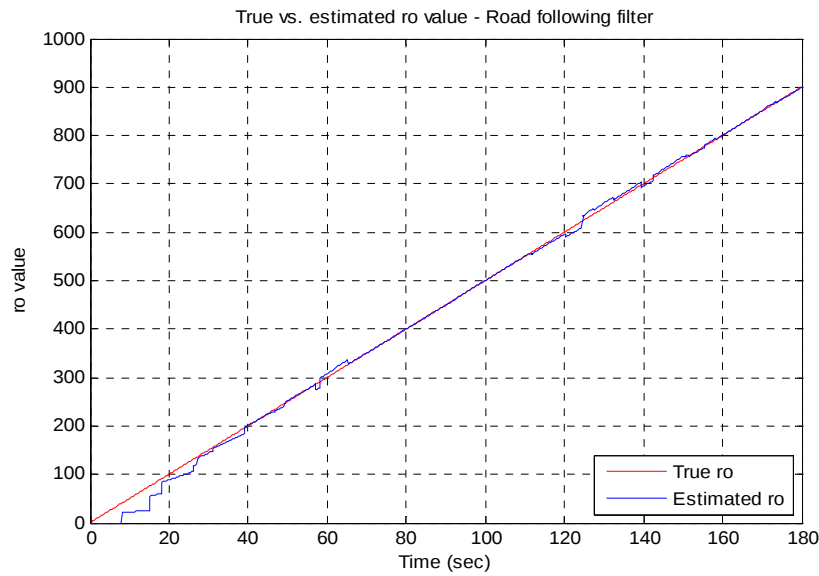


Figure 79. Road following filter ρ comparison plot – Circular road model – ± 10 m PVNT noise

The findings from the previous figures are confirmed with the ρ comparison plot for the ± 10 m PVNT noise trial. The deviations in estimated and true target ρ values are more noticeable than in the previous test. The optimization loop in the S-function code is able to greatly reduce error, but it cannot eliminate all of the variation between the actual ρ value and the resulting ρ value from the PVNT position update.

(b) *Circular Road Model*

The same test parameters are used with the circular road model as with the third order road model.

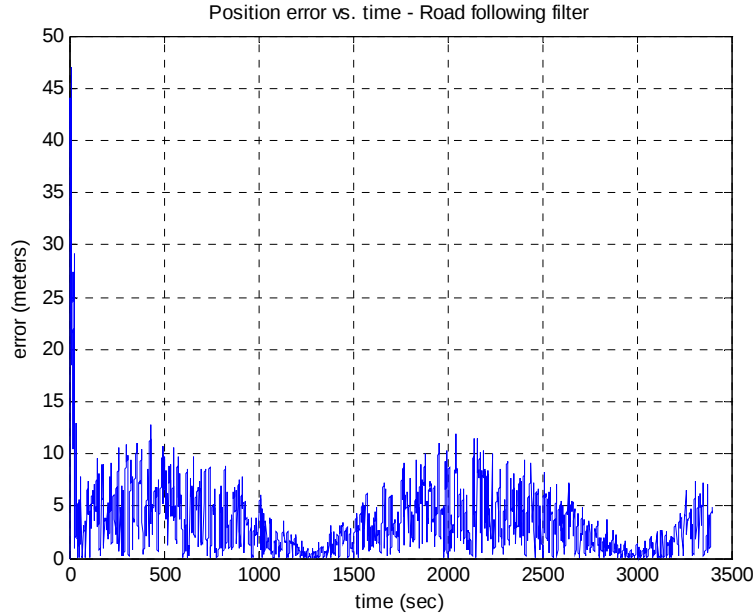


Figure 80. Road following filter position error plot – Circular road model – ± 5 m PVNT noise

The position error plot for the circular road model shows a large increase in RMS error from the ideal conditions test. While the RMS error for the ideal conditions trial is around two meters, the RMS error shown in Figure 81 is roughly seven meters.

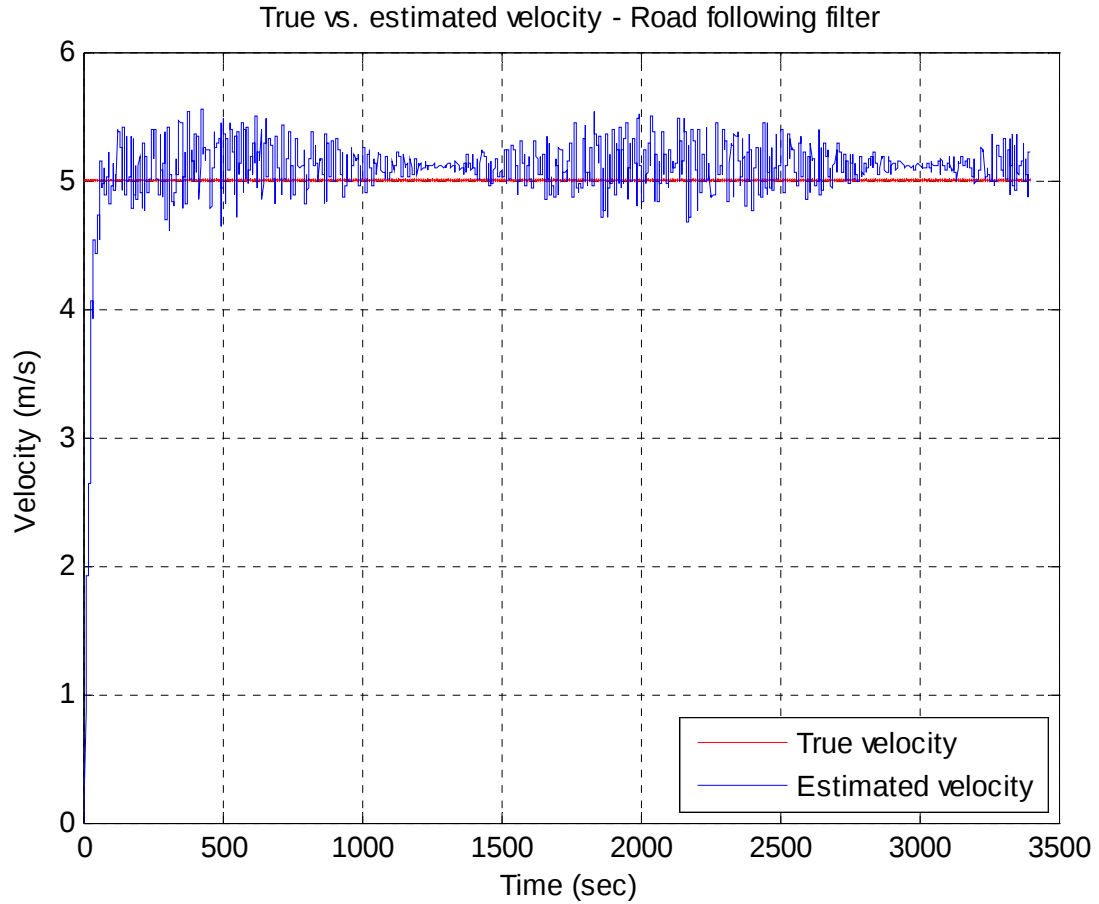


Figure 81. Road following filter velocity comparison plot – Circular road model – ± 5 m PVNT noise

The position comparison plot directly coincides with the velocity comparison plot. The precision of the velocity estimates increase at around 1250 and 3000 seconds into the simulation, resulting in better position estimation.

The PVNT noise is then doubled to ± 10 m for the final set of tests for the circular road model using the real-time road following filter design.

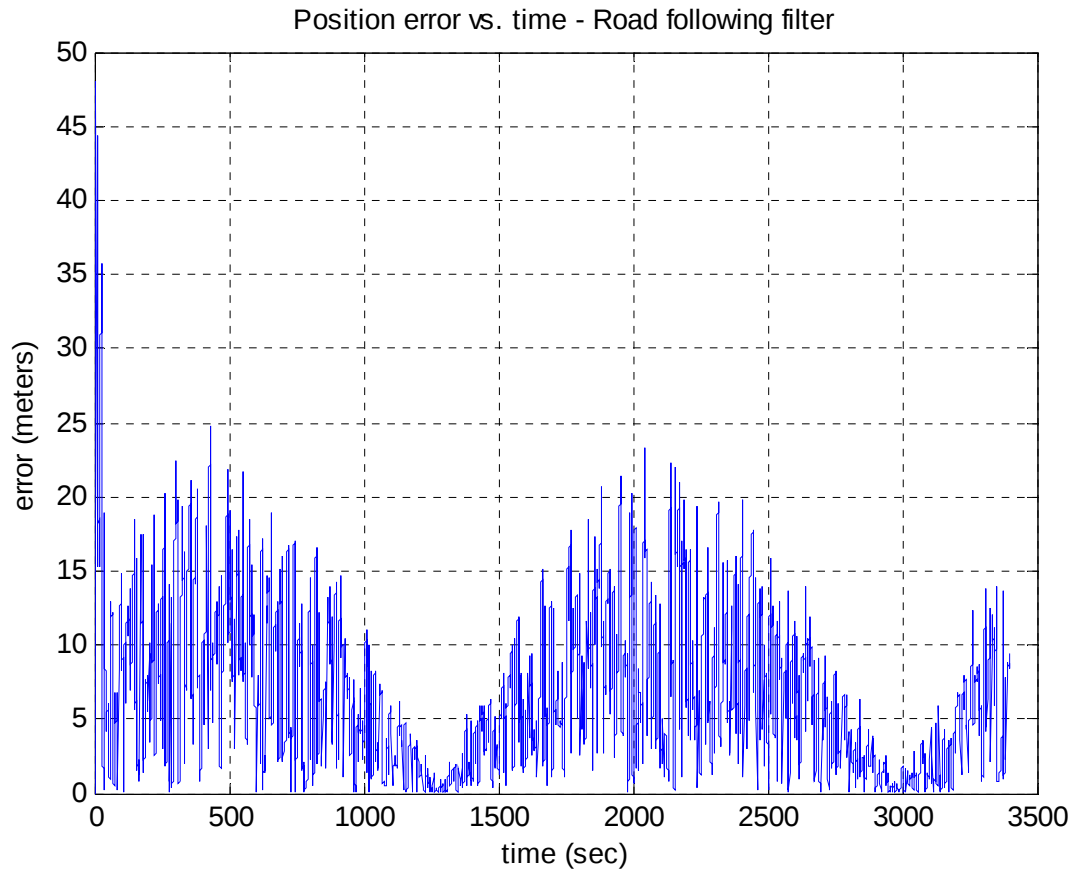


Figure 82. Road following filter position error plot – Circular road model – ± 10 m PVNT noise

The extra five meters of PVNT deviation greatly affect the position estimation results of the real-time road following filter for the circular road model. The peak absolute error value is only two meters less than the peak absolute error value for the same test parameters using the real-time general filter design. The RMS error, however, is much less for the real-time road following filter.

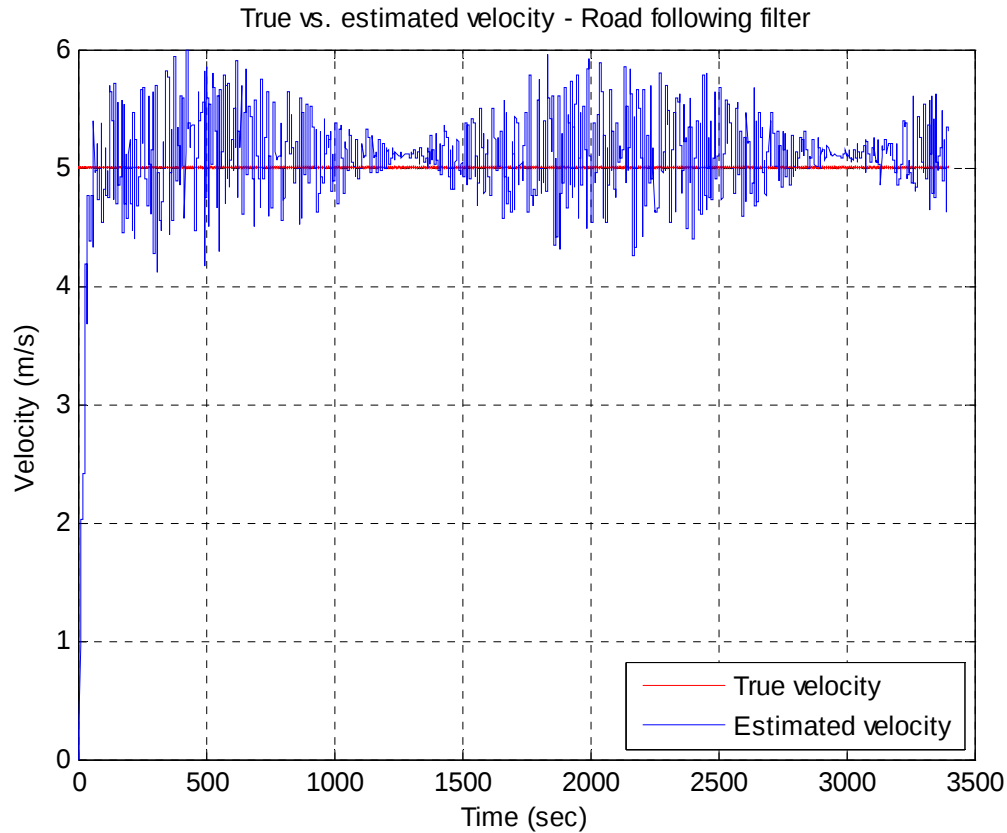


Figure 83. Road following filter velocity comparison plot – Circular road model – ± 10 m PVNT noise

The velocity comparison plot for the real-time road following filter is nearly identical to the velocity plot for the real-time general filter shown in Figure 54. The likeness of the two plots is a perfect example of how real-time road following filter's robustness depends not only on inputted errors, but the road model as well.

c. Additional Road Models and Worst Case Scenarios

The previous examples of the real-time models show that the accuracy of the target motion estimation is greatly affected by the amount of curvature present in the road model. The road following filter design is able to compensate for higher order road models and greater curvatures than the general filter design due to the fact that the road equations are used in the filter code. To show just how much of a difference there is

between the road following and general filters, the real-time simulations are run under ideal conditions using four road models with increasing amounts of curvature. The equations for the road models are shown below:

$$\begin{aligned} x &= \rho \\ z &= 0 \end{aligned} \quad \text{for all road models}$$

$$\text{Road model 1: } y = 2.7922222x \quad (6)$$

$$\text{Road model 2: } y = 0.0000192x^3 - 0.025x^2 + 9.74x \quad (7)$$

$$\text{Road model 3: } y = 0.000033642291x^3 - 0.0444961x^2 + 15.5884511x \quad (8)$$

$$\begin{aligned} \text{Road model 4:} \quad (9) \\ y = -1.98707 \cdot 10^{-10} x^5 + 3.0300064 \cdot 10^{-7} x^4 - 9.832032 \cdot 10^{-5} x^3 - 0.0219297 x^2 + 11.64832x \end{aligned}$$

The four road models are plotted in the following figure to show the amount of curvature for each set of equations.

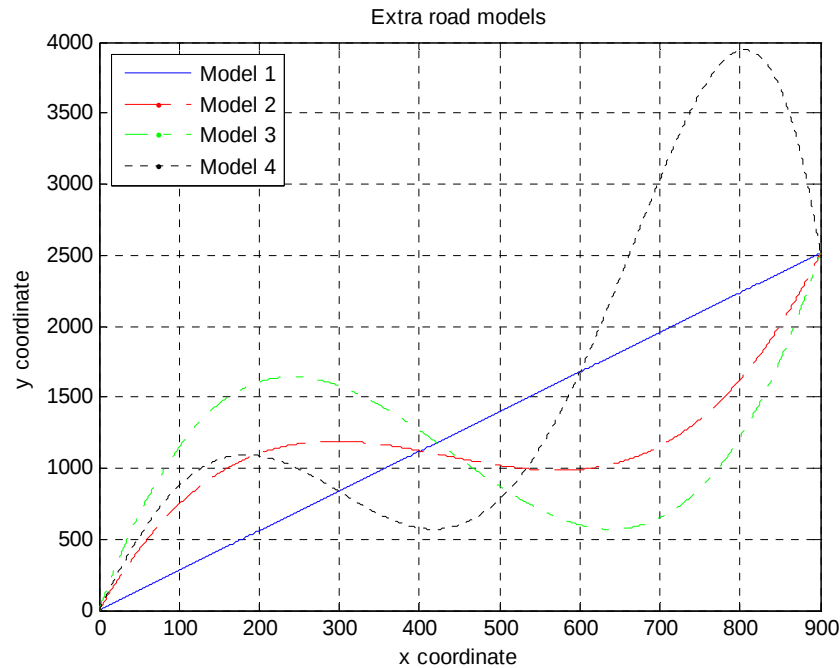


Figure 84. Road model comparison

The real-time road following and general filters are both run for three minute simulations and their position estimation, position error, and velocity error plots are directly compared.

(1) Road Model 1. The first road model depicts a linearly dependent first order plot where there is no curvature in the shape of the road.

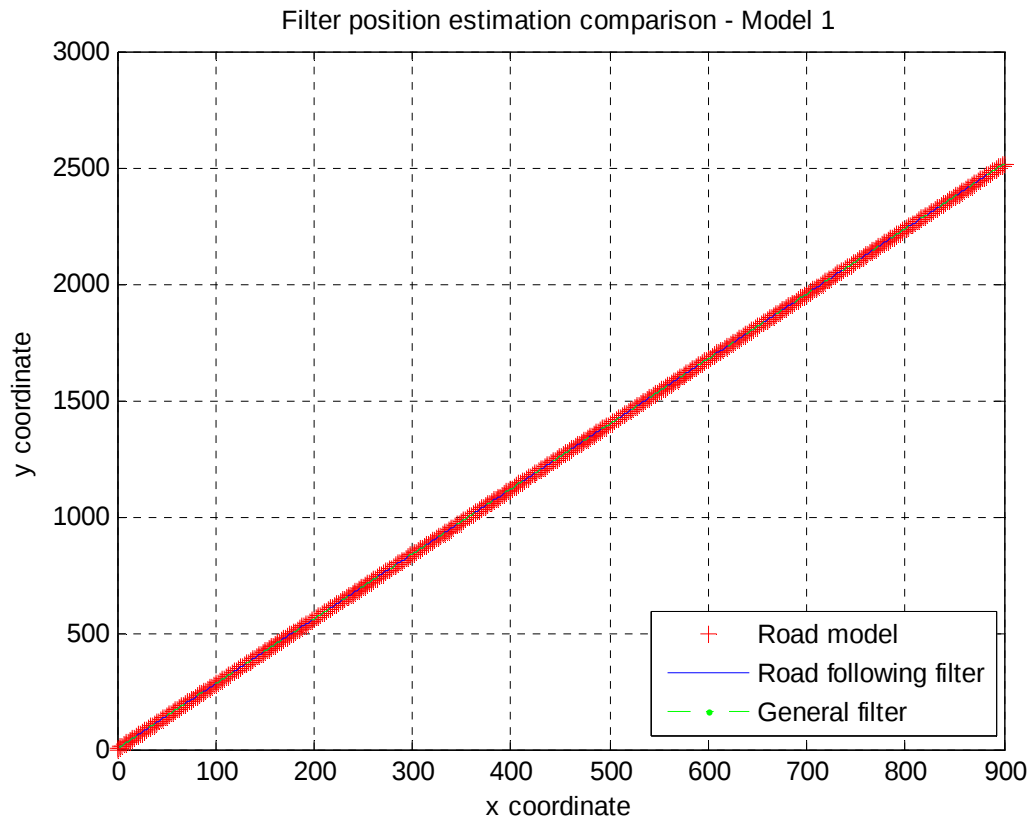


Figure 85. Filter position estimation comparison – Model 1

As expected, there is no difference between the estimated positions from the real-time road following and general filters. This is a rare situation in which the dead reckoning style integration is enough to provide an accurate position estimate for both filters throughout the simulation.

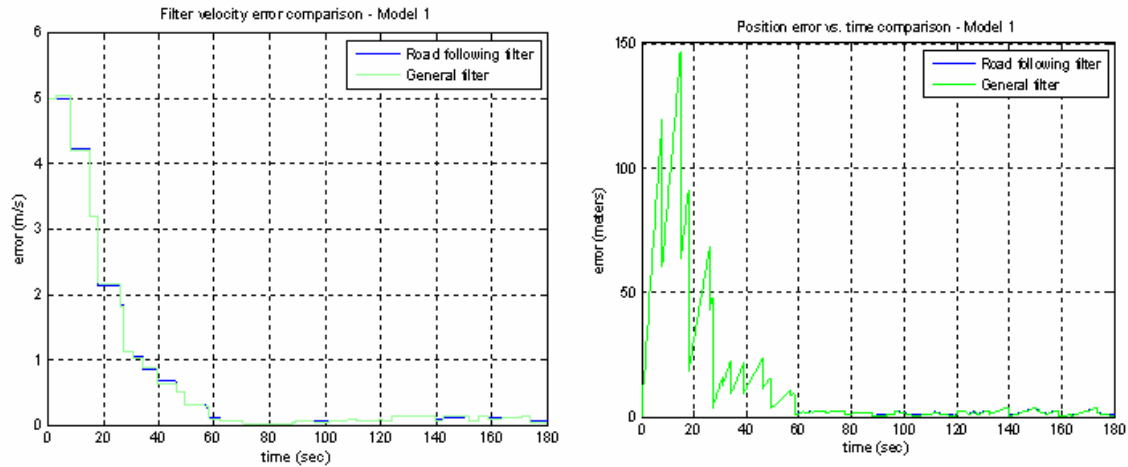


Figure 86. Filter estimation error comparison – Model 1

The error comparison plot between the two filters confirms the results from the position estimation plot. The estimated velocity and position values from the filters are nearly identical throughout the simulation.

(2) Road Model 2. The second road model is the same system of equations used as the “third order road model” in the previous chapters. It is a third order system with a modest amount of curvature throughout the simulation run time.

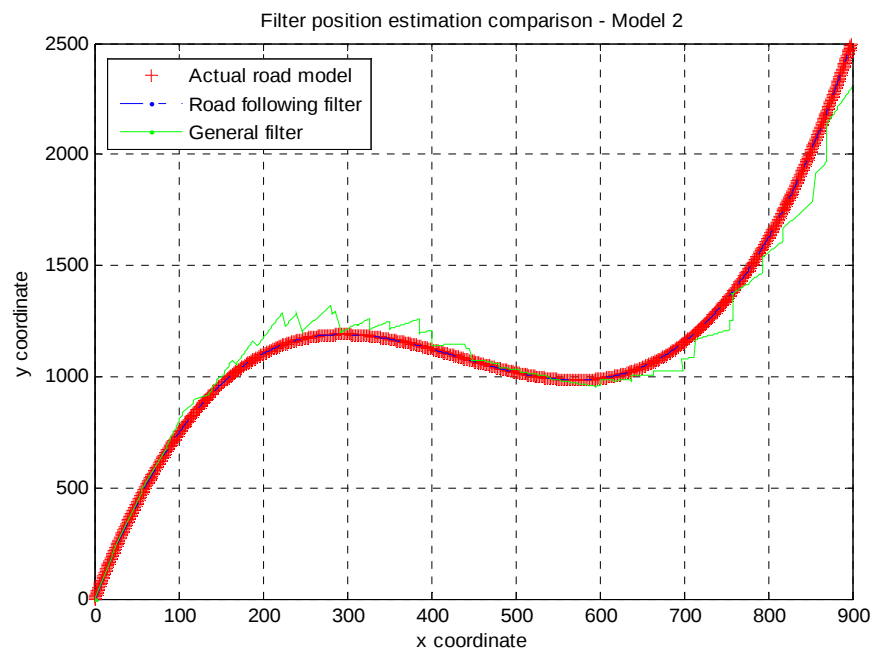


Figure 87. Filter position estimation comparison – Model 2

The figure above shows the amount of deviation the third order equation has between the results of the two filters. During the periods of maximum curvature, specifically around $x=250$ and $x=700$, the real-time general filter is noticed lagging in its position estimates. The real-time road following filter, on the other hand, displays an excellent target approximation compared to the actual road model throughout the test.

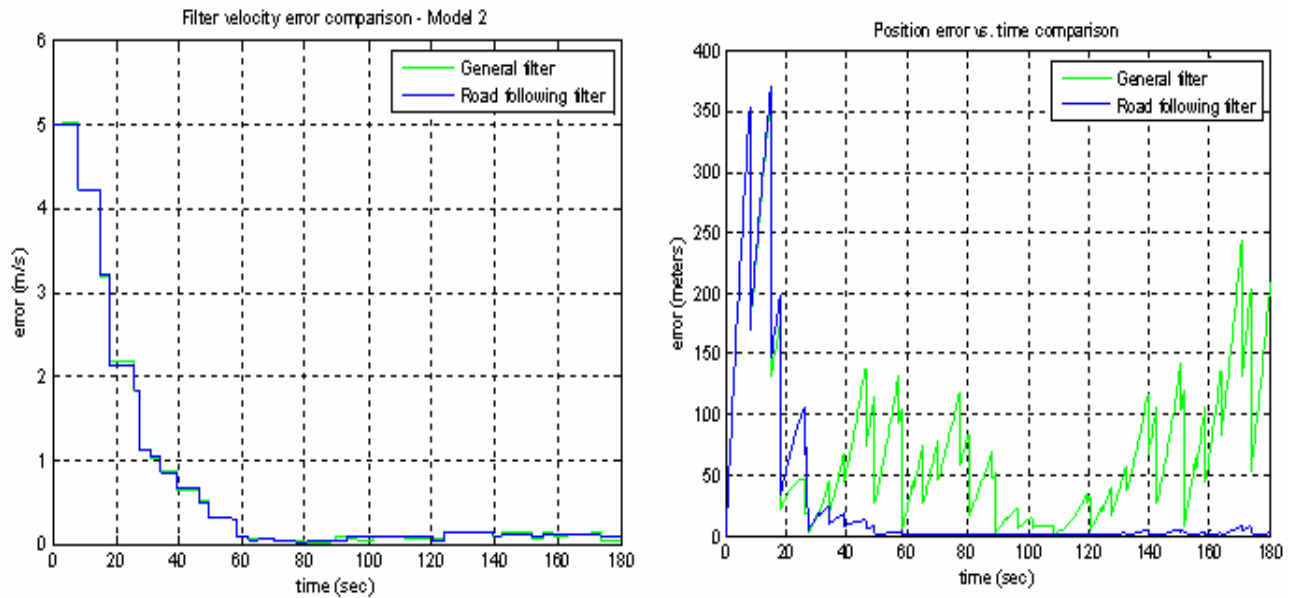


Figure 88. Filter estimation error comparison – Model 2

The comparison of estimated velocity error is quite similar between the two filters while the estimated position error shows a huge difference. Following the target acquisition portion of the run, the errors in estimated velocity stay below 0.25 m/s.

(3) Road Model 3. The next road model tested is a third order system that has an increased amount of curvature when compared to the second road model.

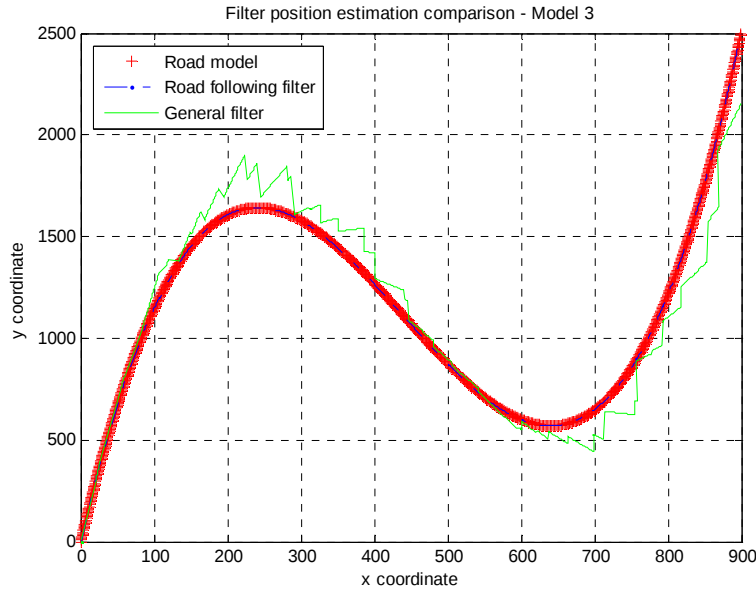


Figure 89. Filter position estimation comparison – Model 3

The third road model shows the real-time road following filter still performing quite well when compared to the actual target track. The dead reckoning estimation from the real-time general filter, however, is worse than the previous road model test. The sections of sharp curvature in the road create large position errors in the real-time general filter's estimation.

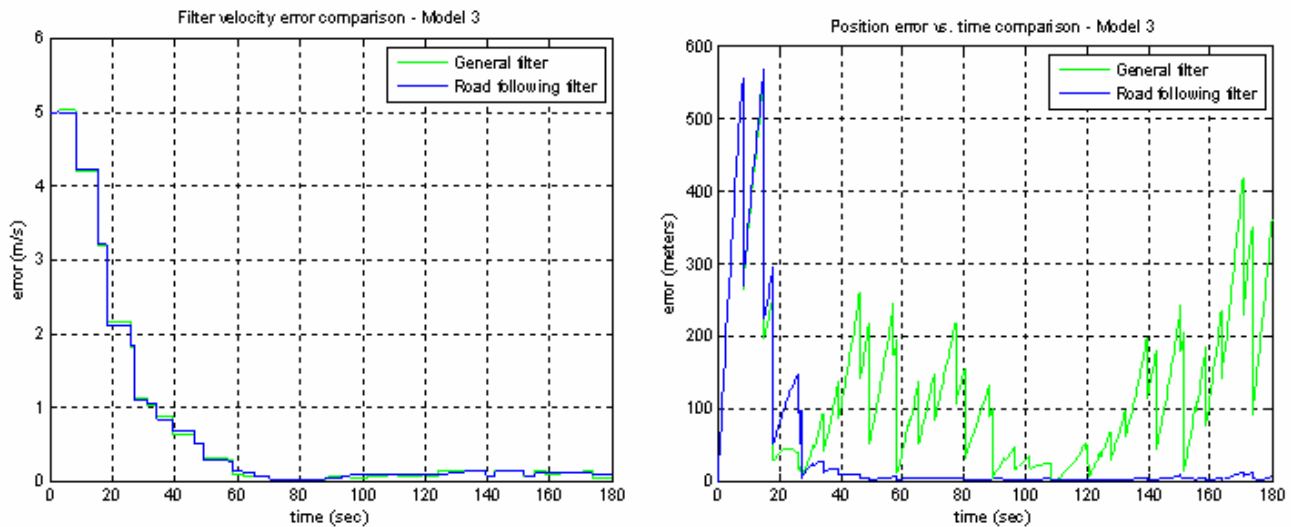


Figure 90. Filter estimation error comparison – Model 3

Despite the large differences in the position error plot, the velocity error comparison figure shows that the real-time road following filter provides only a slightly better estimation than the real-time general filter.

(4) Road Model 4. The final road model tested is meant to greatly increase the amount of curvature seen in the previous models. This last road model features a fifth order system of equations to compare the results from the two filters.

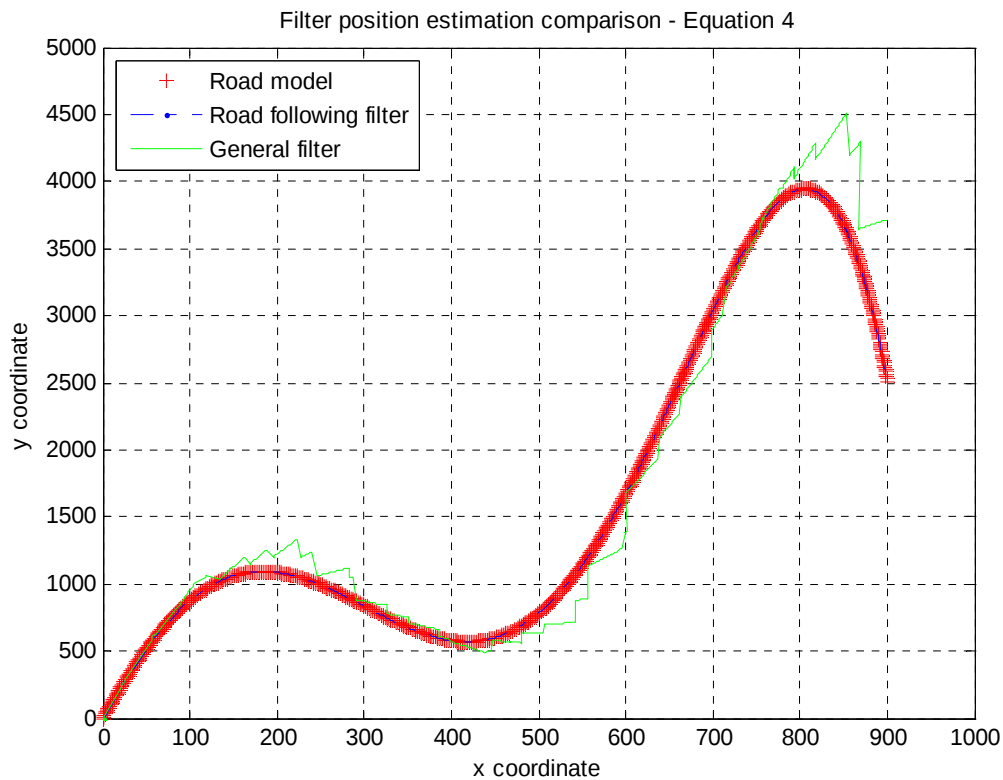


Figure 91. Filter position estimation comparison – Model 4

The extreme amount of curvature present in the fifth order road model clearly decreases the accuracy of the position estimates from the real-time general filter. The estimated target track is far outside the actual target track, especially noticeable around the final, sharp turn at $x=850$. The real-time road following filter, on the other hand, seems to be completely unaffected by the additional turns in the road model as the robustness of the filter to road model curvature is displayed.

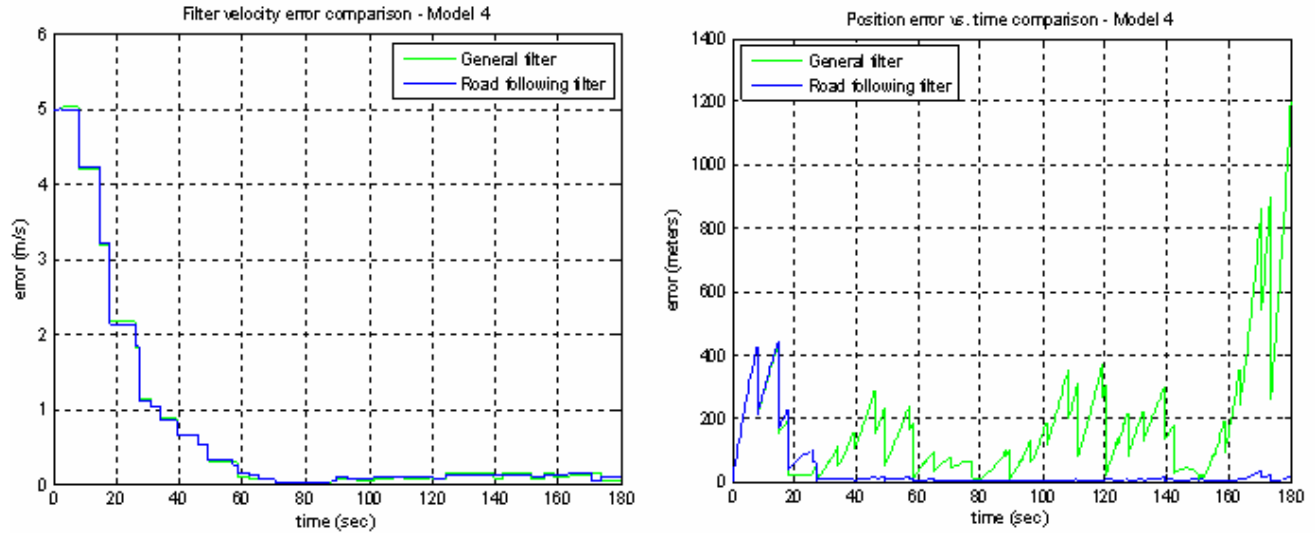


Figure 92. Filter estimation error comparison – Model 4

Similar to the previous trials, the velocity estimation errors are nearly equal between the two filters while the position error plots display large variances. Despite the similarities on the velocity plot, though, the real-time road following filter proves that it is a much better predictor of target motion than the real-time general filter for road models with varying amounts of curvature.

(5) Worst case scenario. A worst case scenario is chosen to show that even though the real-time road following model shows overall great target motion estimation, there are still limits to the amount of PVNT input noise and delay that it can overcome. To show a true failure of the filter, the PVNT input delay is set to 50 seconds for the real-time road following model using the circular road model.

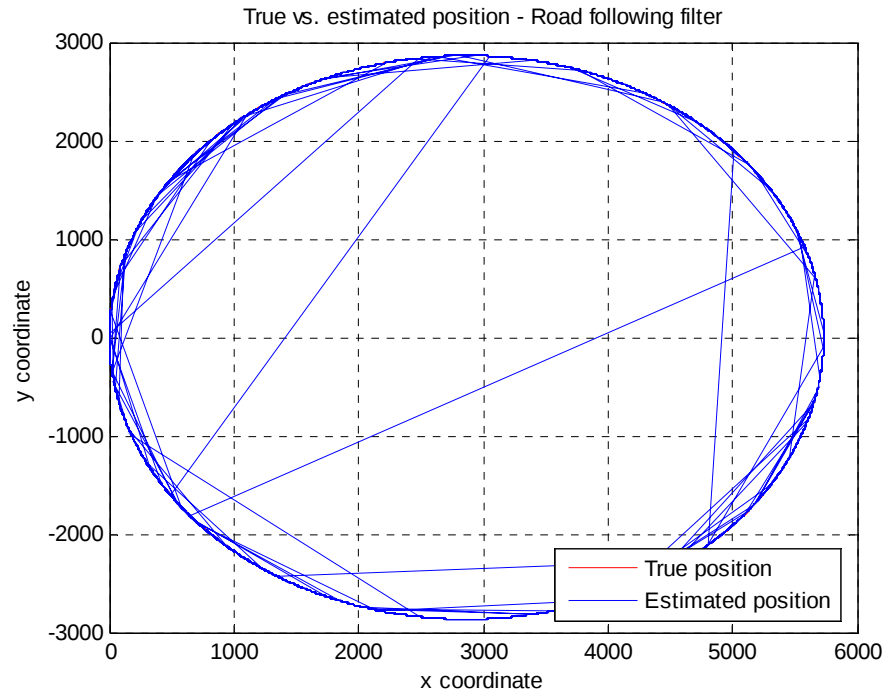


Figure 93. Position comparison plot – Worst case scenario

As shown in Figure 93, the estimated position of the target model is very poor. The 50 second PVNT delay is simply too long for the filter to accurately predict target motion.

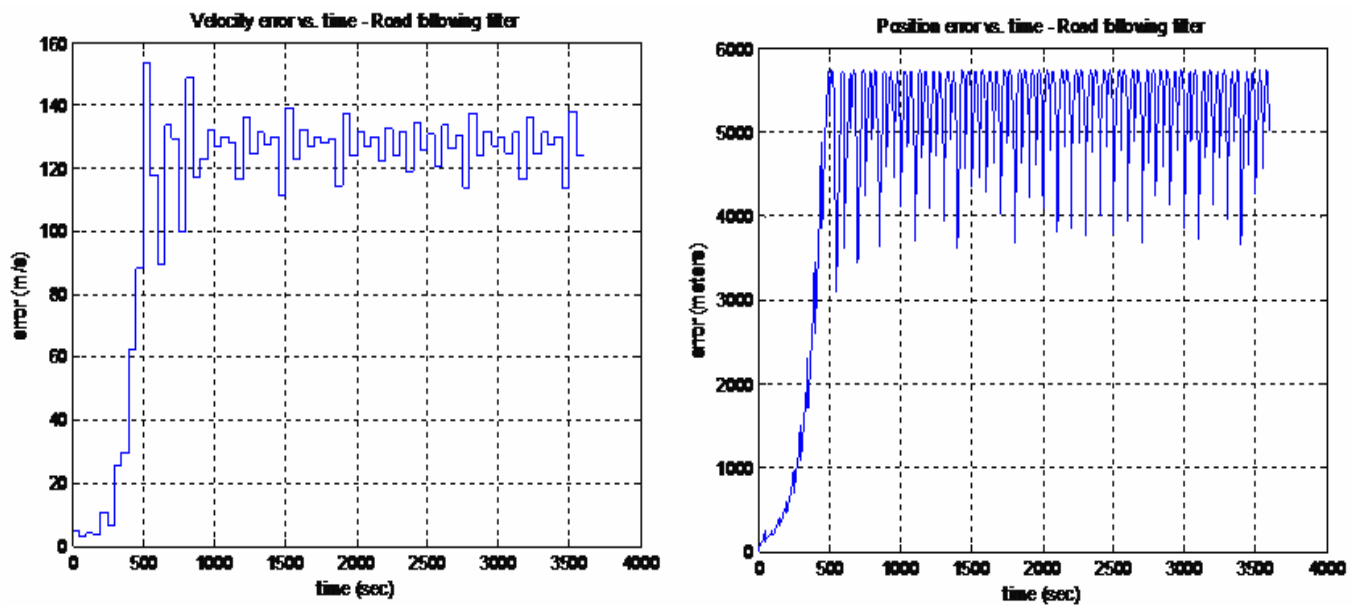


Figure 94. Filter estimation error comparison – Worst case scenario

The velocity error plot shows a steady state error of around 130 m/s while the position error plot depicts nearly a five kilometer position estimation error. Remembering that the circular path is only 2865 meters in radius, these results illustrate a complete failure of the real-time road following model.

V. CONCLUSION AND RECOMMENDATIONS

A. CONCLUSIONS

The overall goals set for this thesis were accomplished. The non real-time road following model was successfully developed and tested to ensure proper function. The problems associated with converting the non real-time systems to real-time were solved with the use of buffers and the implementation of S-function C code. Finally, the real-time general and road following systems were successfully modeled and simulated. All of the results for the real-time models were then compiled and analyzed to provide a definite set of conclusions.

Throughout the simulations in the thesis, the road following filter design shows that it is a better target motion estimator than the general filter design. The simulations display the relative robustness of the real-time road following model to several forms of PVNT input errors while the real-time general filter model results were less accurate. Additionally, the ability of the real-time road following model to provide accurate position and velocity estimation results along simulated roads of increasing curvature were shown. The real-time general filter faltered on road models containing larger amounts of curvature as the dead reckoning integration without the optimization technique was not enough to give accurate results. Finally, while the real-time road following filter performed well in all the practical simulations put forth in the thesis, it was shown that the filter can fail in a worst case scenario involving exceptionally large PVNT input errors.

B. RECOMMENDATIONS

There are quite a few opportunities for further work on the subject of this thesis. The methods used are a solid foundation on which improvements can be made. One simple test that can be worked on includes adjusting the $k1$ and $k2$ gain values for the asynchronous integration loop. Similar to a proportional compensator, lower gain values result in smaller overshoot with slower response time while higher gain values improve

response time but increase overshoot. Some tweaking may be required to find the best compromise for filter performance that will more accurately represent a field testing environment.

Another possible improvement concerning filter accuracy can be made by examining the asynchronous filter integration process. Currently, the model utilizes forward Euler integration to cycle back from the delayed PVNT update to the current simulation time. Future work may involve using trapezoidal or higher order of integration to see if this improves overall target motion estimation accuracy.

Other areas for immediate work include improvements to the S-function code to ensure the minimal amount of required computation time along with storing and plotting the data from each iteration of the asynchronous filter. Further testing can provide results with different types of road models to see what direct relationships exist between road models and filter performance.

Eventually, the simulated real time system can be loaded into hardware and bench-tested. The final goal is to have a program that is able to run in real time on an unmanned aerial vehicle during field testing.

APPENDIX

This appendix presents the ANSI C code for the general and road following filter S-functions as well as a manual explaining the programs' operation.

The purpose of the S-function is to provide an alternative method to MATLAB functions that will allow the system to perform real-time simulations. Each filter design performs a number of different operations with the goal of providing accurate target motion estimation. The filters receive delayed PVNT updates, perform asynchronous forward Euler integration from the update time to current time, and then output the results to the open loop filter. The open loop filter then runs until the next PVNT update arrives.

A. GENERAL FILTER

1. Manual

File: *s_filter_general.c*

Lines 26-49

Complete basic program initializations, library calls, and global variable input.

Lines 26 and 27

Designate the file name and indicate that the file is in C code, to be converted into MEX format and run in MATLAB.

Lines 29-34

Make all the necessary library calls that are required in the program.

Lines 40 and 41

Take in the two S-function parameters, *MAX_DELAY* and *TIME_STEP*, from the S-function block in the Simulink model. *MAX_DELAY* is the maximum amount of expected delay in between PVNT updates while *TIME_STEP* is the time step to be used by the C code. NOTE: The time step parameter value must match the discrete time step value found on the simulation parameters menu in Simulink.

Lines 45 and 46

Convert the parameters into “real_T” format for use in numerical calculations later.

Line 49

Defines the global variable *MAX_INDEX*, used to ensure that buffer overflow does not occur.

Euler_integration function**Line 58**

Lists the inputs to the function along with buffers marked by an asterisk in front of their names.

Lines 65-72

Perform forward Euler integration for the x, y, and z variables, assigning the new position and velocity values to the buffers beginning with “*temp*.”

mdlInitializeSizes

Sets up the sizes of the various vectors used in the code.

Line 82

Means that there will be two parameters inputted into the S-function block in Simulink.

Lines 83-86

Return an error to MATLAB if the incorrect number of parameters is found.

Line 88

Defines zero continuous states since the model is running with a preset, fixed step time.

Line 89

Defines eleven discrete states which must match the number of input ports found in **line 91**.

Lines 92-102

Set the size of each input port

Lines 103-113

Denote each input port as a direct feed through port.

Line 115

Defines eight output ports from the S-function.

Lines 116-123

Define the width of each port.

Line 125

Defines one sample time to be used.

Lines 126-129

define the number of real, integer, pointer, and mode work vectors to be used in the program. The work vectors can be thought of as a value of a certain type (real, integer, pointer, etc.) that is stored in persistent memory. This means that the value will be stored even while the program is called multiple times.

Line 130

Defines the number of zero crossings to be zero as it is not used in the filter program.

mdlInitializeSampleTimes function**Line 144**

Defines the program's sample time to be set to dT , which come from the second parameter input to the S-function block in **line 45**.

Line 145

Indicates a 0.0 second offset time

Line 147

Indicates that a function call is made on the first element of the first output port.

mdlStart function

Defines all of the variables that need to be initialized only once, i.e. the very first time the program is run in the simulation.

Line 161

Predefines the integer work vectors for the index counter and the integrator flag that indicates when the discrete integrator blocks in the open-loop filter subsystem need to be reset.

Lines 162 and 163

Predefine the real work vectors for the initial x, y, and z positions and velocities.

Lines 168-170

Predefine the buffers that are used in the code for data storage.

Lines 175-189

Initialize the buffers to a number of positions equal to *MAX_INDEX* (from **line 49**) with each position having enough memory to store a piece of data with the size *real_T*. The *calloc* command also initializes every position in the buffers to zero.

Lines 191-199

Define the first value for the index counter, integrator reset flag, and position and velocity initial conditions to be zero.

Lines 203-219

Set the pointer work vectors to point to the first position of each of the buffers.

Lines 221-230

Set and store the initial integer and real work values.

mdlOutputs function

Lines 242-270

Contain the input and output declarations.

Lines 242-247 and 253-258

Define the pointers and values of the position and velocity estimates coming from the open loop filter function call.

Lines 248 and 259

Define the pointer and value coming in from the PVNT update delay subsystem

Lines 249-251 and 260-263

Define the actual PVNT update (x,y,z) from the target model subsystem.

Lines 252 and 263

Designate a port for the clock input.

Lines 264-269

Define output ports for the position and velocity initial conditions to the open-loop filter function call.

Line 270

Defines the integrator reset signal, which is also fed into the open-loop filter function call.

Lines 272-280

Contain declarations for the work values and the buffers which match the declarations found in the *mdlStart* function.

Lines 286-290

Define and initialize the non-persistent variables that are used only in the *Euler_integration* and *mdlOutputs* function.

Lines 302-318

Retrieves the values that were stored in the pointer work vectors

Lines 321-328

Retrieves the values that were stored in the integer and real work vectors.

Lines 338-388

Contained in an *if* loop that executes only if the index counter is less than or equal to the preset *MAX_INDEX* value. This ensures that no data is written to the buffers beyond their maximum preset number of storage positions, reducing the risk of buffer overflow.

Line 345

Sets the *integrator_reset* output to the integer work value *integrator_flag*.

Lines 354-359

Take in the estimated position and velocity values from the first six inputs (arriving from the outputs of the open-loop filter function call).

Lines 362-367

Set the respective buffer values to the inputted position and velocity estimates. These values are then also stored in the real work vectors designating position and velocity initial conditions.

Line 378

Resets the integrator flag integer work value to zero (if it was set to one following the Euler integration loop, see **line 473**).

Lines 380-382

Take in the PVNT position update (x,y,z) from the true target model subsystem in the Simulink diagram

Lines 385-387.

Assign the values from the PVNT position update to their respective buffers.

Lines 392-475

Contained in an *if* loop that is only triggered if the input from the PVNT delay subsystem is set high, indicating that a PVNT update is available.

Lines 395-400

Adjust the pointers to each position and velocity buffer so that they now refer to time τ , the time to which the PVNT update refers. This is controlled by the *index* integer work vector which is incremented after each iteration of the *mdlOutputs* function (see **line 479**).

Lines 402-404

Perform the same operation for the buffers that contain the PVNT position update data.

Lines 408-410

Calculate the difference between the estimated position data at time τ and the PVNT position update at time τ for x , y , and z .

Lines 414-419

Set up the values for the first position of the buffers that are used in the *Euler_integration* function and to pass on the updated position and velocity data to the open-loop filter function call.

Line 425

Begins the asynchronous portion of the S-function. The *for* loop runs enough times to move the new estimated position and velocity values from time τ to time t (current system time), which is controlled by the *index* integer work vector value.

Lines 430-432

Set the *delta* variable values originally set in **lines 408-410** to zero after the first iteration of the *for* loop, allowing for normal, dead-reckoning style integration.

Line 435

Passes the required variables to the *Euler_integration* function in **lines 58-73**. Additionally, the “&” in front of the *temp* buffers indicate that their changed values from the *Euler_integration* function will be saved after the function executes.

Lines 439-444

Increment the pointer values for the buffers that will contain the updated position and velocity estimates.

Lines 448-453

Actually set the buffers equal to the updates.

Lines 463-468

After the *for* loop runs the appropriate number of times to arrive at time t , the final value from each of the buffers containing the updated position and velocity estimates are passed to the initial condition real work vectors in these lines.

Additionally, two integrator reset values are set. The first is the *reset_index* variable on **line 470** set equal to one and used inside the S-function program on **line 477**.

The second is the *integrator_flag* integer work value on **line 473** that is outputted to the open-loop function call outside the S-function block.

The remainder of the buffer pointer incrementation/resets take place in the *if/else* loop in **lines 477-501**. The *if* loop portion checks to see if the current *index*

variable value is less than the preset *MAX_INDEX* value and if the *reset_index* variable value is equal to zero (indicating that a PVNT update did not arrive during the current *mdlOutputs* function iteration. If so, the *index* integer work value is incremented along with the pointers to the position and velocity data buffers.

If the criterion for the *if* loop are not met, meaning that a PVNT update has occurred, the buffer pointers are all reset back to their first position and the *index* integer work value is set to zero. This ensures that the buffers are simply overwritten with the new data until the next PVNT update and buffer overflow does not occur. Finally, the pointer work values are updated to now designate the new pointer values for the position and velocity data buffers.

mdlUpdate function

This would be the function in which states would be incremented if they were used in the program. Since the filter design does not use these states, however, the *mdlUpdate* function is only left in the program as a formality.

mdlTerminate function

In this case, all of the data from the buffers must be cleared to avoid errors when re-running the simulation multiple times.

Lines 537-553

Designate each of the buffers that were originally defined in the *mdlStart* function.

Lines 560-574

Actually release the data stored in the buffers.

2. Code

```
1  /* File : s_filter_general.c
2  * Abstract:
3  *
4  * This S-function is a combination of an open-loop filter using a
5  * function call subsystem and an asynchronous filter contained in the
6  * C code of the S-function. The model is used for a target tracking
7  * system, utilizing a delayed position update at different time
8  * intervals. When the position update (labeled PVNT) is not
9  * available, the S-function calls the open-loop filter and stores the
10 * results. When the delayed position update arrives, the loop
11 * containing the asynchronous filter is run to update the previous
12 * data from time tau (corresponding to the PVNT update) to time t
13 * (corresponding to the current time) using buffers to store all
14 * data. The model takes in parameters from the S-function block in
15 * the Simulink model for the maximum amount of delay (seconds) and
16 * the desired time step (seconds). The user can easily manipulate
17 * these parameters without having to change C code in the S-function
18 *
19 * For more details about S-functions, see
20 * matlabroot/simulink/src/sfunmpl_doc.c
21 *
22 * Copyright 1990-2006 The MathWorks, Inc.
23 * $Revision: 1.15.4.3 $
24 */
25
26 #define S_FUNCTION_NAME s_filter_general
27 #define S_FUNCTION_LEVEL 2
28
29 #include "simstruc.h"
30
31 #include <stdlib.h>
32 #include <stdio.h>
33 #include <string.h>
34 #include <math.h>
35
36
37 /* Input Arguments */
38 /*takes in parameters that define a max value for the PVNT update delay and
39 *the desired time step*/
40 #define MAX_DELAY ssGetSFcnParam(S,0)
41 #define TIME_STEP ssGetSFcnParam(S,1)
42
43 /*converts the above parameters from structs to allow them to be used in
44 *computations*/
45 #define dT ((real_T) mxGetPr(TIME_STEP)[0])
46 #define DELAY_MAX ((real_T) mxGetPr(MAX_DELAY)[0])
47
48 /*defines and global constant that is used to prevent buffer overflow*/
49 #define MAX_INDEX (DELAY_MAX/dT)
50
51
```

```

52  /* Function: Euler_integration =====
53  * Abstract:
54  * Performs asynchronous forward Euler integration once the PVNT update is
55  * received in order to rewrite over the previous data from time tau to
56  * time t.
57  */
58  void Euler_integration(double k1, double k2, float delta_x_tou, float delta_y_tou, float
delta_z_tou, float time_step, real_T *new_Px_est_tou, real_T *new_Vx_est_tou, real_T
*new_Py_est_tou, real_T *new_Vy_est_tou, real_T *new_Pz_est_tou, real_T
*new_Vz_est_tou, real_T *temp_new_Px_est_tou, real_T *temp_new_Vx_est_tou,
real_T *temp_new_Py_est_tou, real_T *temp_new_Vy_est_tou, real_T
*temp_new_Pz_est_tou, real_T *temp_new_Vz_est_tou)
59  {
60      /*performs asynchronous double integration with a time step
61      *equal to dT seconds and stores the results in a temp variable
62      *to be transferred to the buffers after they have been
63      *incremented*/
64
65      *temp_new_Px_est_tou = *new_Px_est_tou+ (*new_Vx_est_tou +
k1*delta_x_tou)*time_step;
66      *temp_new_Vx_est_tou = *new_Vx_est_tou+ (k2*delta_x_tou)*time_step;
67
68      *temp_new_Py_est_tou = *new_Py_est_tou+ (*new_Vy_est_tou +
k1*delta_y_tou)*time_step;
69      *temp_new_Vy_est_tou = *new_Vy_est_tou+ (k2*delta_y_tou)*time_step;
70
71      *temp_new_Pz_est_tou = *new_Pz_est_tou+ (*new_Vz_est_tou +
k1*delta_z_tou)*time_step;
72      *temp_new_Vz_est_tou = *new_Vz_est_tou+ (k2*delta_z_tou)*time_step;
73  }
74
75
76  /* Function: mdlInitializeSizes =====
77  * Abstract:
78  * Setup sizes of the various vectors.
79  */
80  static void mdlInitializeSizes(SimStruct *S)
81  {
82      ssSetNumSFcnParams(S, 2); /* Number of expected parameters */
83      if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
84      {
85          return; /* Parameter mismatch will be reported by Simulink */
86      }
87
88      ssSetNumContStates(S, 0);          /*defines 0 continuous states*/
89      ssSetNumDiscStates(S, 11);        /*defines 11 discrete states*/
90
91      if (!ssSetNumInputPorts(S, 11)) return; /*defines 11 input ports*/
92      ssSetInputPortWidth(S, 0, 1);     /*sets input 1 port size to 1*/
93      ssSetInputPortWidth(S, 1, 1);     /*sets input 2 port size to 1*/
94      ssSetInputPortWidth(S, 2, 1);     /*sets input 3 port size to 1*/
95      ssSetInputPortWidth(S, 3, 1);     /*sets input 4 port size to 1*/
96      ssSetInputPortWidth(S, 4, 1);     /*sets input 5 port size to 1*/
97      ssSetInputPortWidth(S, 5, 1);     /*sets input 6 port size to 1*/
98      ssSetInputPortWidth(S, 6, 1);     /*sets input 7 port size to 1*/

```

```

99     ssSetInputPortWidth(S, 7, 1);      /*sets input 8 port size to 1*/
100    ssSetInputPortWidth(S, 8, 1);      /*sets input 9 port size to 1*/
101    ssSetInputPortWidth(S, 9, 1);      /*sets input 10 port size to 1*/
102    ssSetInputPortWidth(S, 10, 1);     /*sets input 11 port size to 1*/
103    ssSetInputPortDirectFeedThrough(S, 0, 1);
104    ssSetInputPortDirectFeedThrough(S, 1, 1);
105    ssSetInputPortDirectFeedThrough(S, 2, 1);
106    ssSetInputPortDirectFeedThrough(S, 3, 1);
107    ssSetInputPortDirectFeedThrough(S, 4, 1);
108    ssSetInputPortDirectFeedThrough(S, 5, 1);
109    ssSetInputPortDirectFeedThrough(S, 6, 1);
110    ssSetInputPortDirectFeedThrough(S, 7, 1);
111    ssSetInputPortDirectFeedThrough(S, 8, 1);
112    ssSetInputPortDirectFeedThrough(S, 9, 1);
113    ssSetInputPortDirectFeedThrough(S, 10, 1);
114
115    if (!ssSetNumOutputPorts(S,8)) return;
116    ssSetOutputPortWidth(S, 0, 1);     /*sets output port 1 width to 1*/
117    ssSetOutputPortWidth(S, 1, 1);     /*sets output port 2 width to 1*/
118    ssSetOutputPortWidth(S, 2, 1);     /*sets output port 3 width to 1*/
119    ssSetOutputPortWidth(S, 3, 1);     /*sets output port 4 width to 1*/
120    ssSetOutputPortWidth(S, 4, 1);     /*sets output port 5 width to 1*/
121    ssSetOutputPortWidth(S, 5, 1);     /*sets output port 6 width to 1*/
122    ssSetOutputPortWidth(S, 6, 1);     /*sets output port 7 width to 1*/
123    ssSetOutputPortWidth(S, 7, 1);     /*sets output port 8 width to 1*/
124
125    ssSetNumSampleTimes(    S, 1);
126    ssSetNumRWork(          S, 6);     /*real vector*/
127    ssSetNumIWork(          S, 2);     /*integer vector*/
128    ssSetNumPWork(          S, 15);    /*pointer vector*/
129    ssSetNumModes(          S, 0);     /*mode vector*/
130    ssSetNumNonsampledZCs(  S, 0);     /*number of zero crossings*/
131
132    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
133    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
134 }
135
136
137 /* Function: mdlInitializeSampleTimes =====
138  * Abstract:
139  *   Discrete sample time of dT seconds and specify that we are doing
140  *   function-call's on the 1st element of the 1st output port.
141  */
142 static void mdlInitializeSampleTimes(SimStruct *S)
143 {
144     ssSetSampleTime(S, 0, dT); /*sets sample time to dT seconds*/
145     ssSetOffsetTime(S, 0, 0.0); /*indicates 0 offset time*/
146
147     ssSetCallSystemOutput(S,0); /* call on first element */
148     ssSetModelReferenceSampleTimeDefaultInheritance(S);
149 }
150
151
152 /*Function: mdlStart =====
153  *Abstract:

```

```

154 * This function sets up the variables passed between the function and
155 * the s-function.
156 */
157 #define MDL_START
158
159 static void mdlStart(SimStruct *S)
160 {
161     int_T index, integrator_flag;
162     real_T initial_x_position, initial_y_position, initial_z_position;
163     real_T initial_x_velocity, initial_y_velocity, initial_z_velocity;
164
165     /*The four real_T variables below denote the buffers used to store
166     *velocity and position data over multiple iterations of the
167     *s-function*/
168     real_T *velocity_x_data, *position_x_data, *velocity_y_data, *position_y_data,
169     *velocity_z_data, *position_z_data;
170     real_T *new_Vx_est_tou, *new_Px_est_tou, *new_Vy_est_tou, *new_Py_est_tou,
171     *new_Vz_est_tou, *new_Pz_est_tou;
172     real_T *x_PVNT_data, *y_PVNT_data, *z_PVNT_data;
173
174     /*The buffers are allocated enough memory to store 'MAX_INDEX' data
175     *with each data space being 'real_T' size. The 'calloc' command also
176     *initializes the buffers*/
177     velocity_x_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
178     position_x_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
179     velocity_y_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
180     position_y_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
181     velocity_z_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
182     position_z_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
183     new_Vx_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
184     new_Px_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
185     new_Vy_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
186     new_Py_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
187     new_Vz_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
188     new_Pz_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
189     x_PVNT_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
190     y_PVNT_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
191     z_PVNT_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
192
193     index = 0; /*initializes index to 0*/
194     integrator_flag = 0; //sets integration reset flag to 0
195
196     initial_x_velocity = 0.0; //initializes position and velocity
197     initial_y_velocity = 0.0; //IC's to 0
198     initial_z_velocity = 0.0;
199     initial_x_position = 0.0;
200     initial_y_position = 0.0;
201     initial_z_position = 0.0;
202
203     /*Sets the pointer work variables for the buffers*/
204     ssSetPWorkValue(S, 0, (real_T *)velocity_x_data);
205     ssSetPWorkValue(S, 1, (real_T *)position_x_data);
206     ssSetPWorkValue(S, 2, (real_T *)velocity_y_data);
207     ssSetPWorkValue(S, 3, (real_T *)position_y_data);

```

```

207     ssSetPWorkValue(S, 4, (real_T *)velocity_z_data);
208     ssSetPWorkValue(S, 5, (real_T *)position_z_data);
209
210     ssSetPWorkValue(S, 6, (real_T *)new_Vx_est_tou);
211     ssSetPWorkValue(S, 7, (real_T *)new_Px_est_tou);
212     ssSetPWorkValue(S, 8, (real_T *)new_Vy_est_tou);
213     ssSetPWorkValue(S, 9, (real_T *)new_Py_est_tou);
214     ssSetPWorkValue(S, 10, (real_T *)new_Vz_est_tou);
215     ssSetPWorkValue(S, 11, (real_T *)new_Pz_est_tou);
216
217     ssSetPWorkValue(S, 12, (real_T *)x_PVNT_data);
218     ssSetPWorkValue(S, 13, (real_T *)y_PVNT_data);
219     ssSetPWorkValue(S, 14, (real_T *)z_PVNT_data);
220
221     ssSetIWorkValue(S, 0, index);           /*sets the first integer work
222                                             *value to the index variable*/
223     ssSetIWorkValue(S, 1, integrator_flag); /*sets the second integer work
224                                             *value to the integrator flag*/
225     ssSetRWorkValue(S, 0, initial_x_velocity); /*sets the real work values*/
226     ssSetRWorkValue(S, 1, initial_x_position);
227     ssSetRWorkValue(S, 2, initial_y_velocity);
228     ssSetRWorkValue(S, 3, initial_y_position);
229     ssSetRWorkValue(S, 4, initial_z_velocity);
230     ssSetRWorkValue(S, 5, initial_z_position);
231 }
232
233
234 /* Function: mdlOutputs =====
235  * Abstract:
236  *   Issue ssCallSystemWithTid on 1st output element of 1st output port
237  *   and then update 2nd output port with the state.
238  */
239 static void mdlOutputs(SimStruct *S, int_T tid)
240 {
241     /*S-function input and output declarations*/
242     real_T      *Vx_est      = ssGetRealDiscStates(S,0);
243     real_T      *Px_est      = ssGetRealDiscStates(S,1);
244     real_T      *Vy_est      = ssGetRealDiscStates(S,2);
245     real_T      *Vz_est      = ssGetRealDiscStates(S,4);
246     real_T      *Pz_est      = ssGetRealDiscStates(S,5);
247     real_T      *PVNT        = ssGetRealDiscStates(S,6);
248     real_T      *x_ro        = ssGetRealDiscStates(S,7);
249     real_T      *y_ro        = ssGetRealDiscStates(S,8);
250     real_T      *z_ro        = ssGetRealDiscStates(S,9);
251     real_T      *clock        = ssGetRealDiscStates(S,10);
252     InputRealPtrsType Vx_est_Ptrs = ssGetInputPortRealSignalPtrs(S,0);
253     InputRealPtrsType Px_est_Ptrs = ssGetInputPortRealSignalPtrs(S,1);
254     InputRealPtrsType Vy_est_Ptrs = ssGetInputPortRealSignalPtrs(S,2);
255     InputRealPtrsType Py_est_Ptrs = ssGetInputPortRealSignalPtrs(S,3);
256     InputRealPtrsType Vz_est_Ptrs = ssGetInputPortRealSignalPtrs(S,4);
257     InputRealPtrsType Pz_est_Ptrs = ssGetInputPortRealSignalPtrs(S,5);
258     InputRealPtrsType PVNT_Ptrs  = ssGetInputPortRealSignalPtrs(S,6);
259     InputRealPtrsType x_ro_Ptrs  = ssGetInputPortRealSignalPtrs(S,7);
260     InputRealPtrsType y_ro_Ptrs  = ssGetInputPortRealSignalPtrs(S,8);
261     InputRealPtrsType z_ro_Ptrs  = ssGetInputPortRealSignalPtrs(S,9);

```

```

263 InputRealPtrsType clock_Ptrs = ssGetInputPortRealSignalPtrs(S,10);
264 real_T *TgtVx_IC = ssGetOutputPortRealSignal(S,1);
265 real_T *TgtPx_IC = ssGetOutputPortRealSignal(S,2);
266 real_T *TgtVy_IC = ssGetOutputPortRealSignal(S,3);
267 real_T *TgtPy_IC = ssGetOutputPortRealSignal(S,4);
268 real_T *TgtVz_IC = ssGetOutputPortRealSignal(S,5);
269 real_T *TgtPz_IC = ssGetOutputPortRealSignal(S,6);
270 real_T *integrator_reset = ssGetOutputPortRealSignal(S,7);
271
272 int_T index, integrator_flag;
273 real_T initial_x_position, initial_y_position, initial_z_position;
274 real_T initial_x_velocity, initial_y_velocity, initial_z_velocity;
275 real_T temp_new_Vx_est_tou, temp_new_Px_est_tou, temp_new_Vy_est_tou,
temp_new_Py_est_tou, temp_new_Vz_est_tou, temp_new_Pz_est_tou;
276
277 /*buffer declarations for mdlOutputs*/
278 real_T *velocity_x_data, *position_x_data, *velocity_y_data, *position_y_data,
*velocity_z_data, *position_z_data;
279 real_T *new_Vx_est_tou, *new_Px_est_tou, *new_Vy_est_tou, *new_Py_est_tou,
*new_Vz_est_tou, *new_Pz_est_tou;
280 real_T *x_PVNT_data, *y_PVNT_data, *z_PVNT_data;
281
282 /*defines pointer to output file for forward Euler integration results*/
283 // FILE *Euler_output_data;
284
285 /*defines intermediate position and velocity matrices*/
286 float delta_x_tou = 0.0, delta_y_tou = 0.0, delta_z_tou = 0.0;
287 float time_index = 0.0, delay = 0.0, time_step = dT;
288 int i = 0; /*counter*/
289 int reset_index = 0; /*flag indicating and index reset to 0*/
290 double k1=0.5, k2=0.5; /*sets integrator gains*/
291
292 /*
293 * ssCallSystemWithTid is used to execute a function-call subsystem. The
294 * 2nd argument is the element of the 1st output port index which
295 * connected to the function-call subsystem. Function-call subsystems
296 * can be driven by the first output port of s-function blocks.
297 */
298
299 UNUSED_ARG(tid); /* not used in single tasking mode */
300
301 /*Retrieves the pointer work values for the buffers*/
302 velocity_x_data = (real_T *)ssGetPWorkValue(S, 0);
303 position_x_data = (real_T *)ssGetPWorkValue(S, 1);
304 velocity_y_data = (real_T *)ssGetPWorkValue(S, 2);
305 position_y_data = (real_T *)ssGetPWorkValue(S, 3);
306 velocity_z_data = (real_T *)ssGetPWorkValue(S, 4);
307 position_z_data = (real_T *)ssGetPWorkValue(S, 5);
308
309 new_Vx_est_tou = (real_T *)ssGetPWorkValue(S, 6);
310 new_Px_est_tou = (real_T *)ssGetPWorkValue(S, 7);
311 new_Vy_est_tou = (real_T *)ssGetPWorkValue(S, 8);
312 new_Py_est_tou = (real_T *)ssGetPWorkValue(S, 9);
313 new_Vz_est_tou = (real_T *)ssGetPWorkValue(S, 10);
314 new_Pz_est_tou = (real_T *)ssGetPWorkValue(S, 11);

```

```

315
316 x_PVNT_data      = (real_T *)ssGetPWorkValue(S, 12);
317 y_PVNT_data      = (real_T *)ssGetPWorkValue(S, 13);
318 z_PVNT_data      = (real_T *)ssGetPWorkValue(S, 14);
319
320 /*Retrieves integer and real work values*/
321 index             = ssGetIWorkValue(S,0);
322 integrator_flag    = ssGetIWorkValue(S,1);
323 initial_x_velocity = ssGetRWorkValue(S,0);
324 initial_x_position = ssGetRWorkValue(S,1);
325 initial_y_velocity = ssGetRWorkValue(S,2);
326 initial_y_position = ssGetRWorkValue(S,3);
327 initial_z_velocity = ssGetRWorkValue(S,4);
328 initial_z_position = ssGetRWorkValue(S,5);
329
330 /*creates .txt file for output results*/
331 // Euler_output_data = fopen("Euler_data_general.txt", "w");
332
333 /*Entire sequence is in an 'if' loop to ensure that there is no
334 *overflow for the position and velocity arrays (defined with a maximum
335 *of MAX_INDEX data points.)*/
336 if(index <= (int)MAX_INDEX)
337 {
338     TgtPx_IC[0] = initial_x_position;          /*sets outputs to initial V and P*/
339     TgtVx_IC[0] = initial_x_velocity;
340     TgtPy_IC[0] = initial_y_position;
341     TgtVy_IC[0] = initial_y_velocity;
342     TgtPz_IC[0] = initial_z_position;
343     TgtVz_IC[0] = initial_z_velocity;
344
345     integrator_reset[0] = integrator_flag;      /*sets output 3 to integration
346                                                  *reset flag*/
347
348     if(!ssCallSystemWithTid(S,0,tid))          /*calls system with task ID 1*/
349     {
350         /* Error occurred which will be reported by Simulink */
351         return;
352     }
353
354     Vx_est_Ptrs = ssGetInputPortRealSignalPtrs(S,0); /*Gets inputs*/
355     Px_est_Ptrs = ssGetInputPortRealSignalPtrs(S,1);
356     Vy_est_Ptrs = ssGetInputPortRealSignalPtrs(S,2);
357     Py_est_Ptrs = ssGetInputPortRealSignalPtrs(S,3);
358     Vz_est_Ptrs = ssGetInputPortRealSignalPtrs(S,4);
359     Pz_est_Ptrs = ssGetInputPortRealSignalPtrs(S,5);
360
361     /*assigns the position and velocity data to the buffers*/
362     *position_x_data = (real_T)*Px_est_Ptrs[0];
363     *velocity_x_data = (real_T)*Vx_est_Ptrs[0];
364     *position_y_data = (real_T)*Py_est_Ptrs[0];
365     *velocity_y_data = (real_T)*Vy_est_Ptrs[0];
366     *position_z_data = (real_T)*Pz_est_Ptrs[0];
367     *velocity_z_data = (real_T)*Vz_est_Ptrs[0];
368
369     /*resets the initial velocity and position values*/

```

```

370     initial_x_velocity = ssSetRWorkValue(S, 0, (real_T)*Vx_est_Ptrs[0]);
371     initial_x_position = ssSetRWorkValue(S, 1, (real_T)*Px_est_Ptrs[0]);
372     initial_y_velocity = ssSetRWorkValue(S, 2, (real_T)*Vy_est_Ptrs[0]);
373     initial_y_position = ssSetRWorkValue(S, 3, (real_T)*Py_est_Ptrs[0]);
374     initial_z_velocity = ssSetRWorkValue(S, 4, (real_T)*Vz_est_Ptrs[0]);
375     initial_z_position = ssSetRWorkValue(S, 5, (real_T)*Pz_est_Ptrs[0]);
376
377     /*resets the integrator reset to 0*/
378     integrator_flag = ssSetIWorkValue(S, 1, 0);
379
380     x_ro_Ptrs = ssGetInputPortRealSignalPtrs(S,7); /*takes in ro_star value*/
381     y_ro_Ptrs = ssGetInputPortRealSignalPtrs(S,8);
382     z_ro_Ptrs = ssGetInputPortRealSignalPtrs(S,9);
383
384     /*assigns coordinates to buffers*/
385     *x_PVNT_data = (real_T)*x_ro_Ptrs[0];
386     *y_PVNT_data = (real_T)*y_ro_Ptrs[0];
387     *z_PVNT_data = (real_T)*z_ro_Ptrs[0];
388 }
389
390 if ((real_T)*PVNT_Ptrs[0] >= 0.99)
391 /*indicates pulse is high (PVNT update present)*/
392 {
393     /*calls the estimated position and velocity values at time tou from
394     *the buffers*/
395     position_x_data = position_x_data - index;
396     velocity_x_data = velocity_x_data - index;
397     position_y_data = position_y_data - index;
398     velocity_y_data = velocity_y_data - index;
399     position_z_data = position_z_data - index;
400     velocity_z_data = velocity_z_data - index;
401
402     x_PVNT_data = x_PVNT_data - index;
403     y_PVNT_data = y_PVNT_data - index;
404     z_PVNT_data = z_PVNT_data - index;
405
406     /*calculates the difference between the ro_star update value and the
407     *estimated ro value at time tou*/
408     delta_x_tou = *x_PVNT_data - *position_x_data;
409     delta_y_tou = *y_PVNT_data - *position_y_data;
410     delta_z_tou = *z_PVNT_data - *position_z_data;
411
412     /*sets up the initial conditions based on the x,y,z input from the
413     *PVNT update*/
414     *new_Px_est_tou = *x_PVNT_data;
415     *new_Vx_est_tou = *velocity_x_data;
416     *new_Py_est_tou = *y_PVNT_data;
417     *new_Vy_est_tou = *velocity_y_data;
418     *new_Pz_est_tou = *z_PVNT_data;
419     *new_Vz_est_tou = *velocity_z_data;
420
421     /*sets up time output for Euler_data file*/
422     delay = index;
423     time_index = *clock_Ptrs[0] - (delay * dT);
424

```

```

425     for (i=0; i<index; i++)          /*increments counter from 0 to the
426                                     *maximum value of the index*/
427     {
428         if (i != 0)                  /*allows normal integration after first iteration*/
429         {
430             delta_x_tou = 0.0;
431             delta_y_tou = 0.0;
432             delta_z_tou = 0.0;
433         }
434
435         Euler_integration(k1, k2, delta_x_tou, delta_y_tou, delta_z_tou, time_step,
new_Px_est_tou, new_Vx_est_tou, new_Py_est_tou, new_Vy_est_tou,
new_Pz_est_tou, new_Vz_est_tou, &temp_new_Px_est_tou,
&temp_new_Vx_est_tou, &temp_new_Py_est_tou, &temp_new_Vy_est_tou,
&temp_new_Pz_est_tou, &temp_new_Vz_est_tou);

436
437         /*increments the new_ro_est_tou and new_V_sca_est_tou buffer
438         *pointers*/
439         new_Px_est_tou++;
440         new_Vx_est_tou++;
441         new_Py_est_tou++;
442         new_Vy_est_tou++;
443         new_Pz_est_tou++;
444         new_Vz_est_tou++;
445
446         /*sets the now incremented buffers equal to the results from
447         *the forward Euler integration*/
448         *new_Px_est_tou = temp_new_Px_est_tou;
449         *new_Vx_est_tou = temp_new_Vx_est_tou;
450         *new_Py_est_tou = temp_new_Py_est_tou;
451         *new_Vy_est_tou = temp_new_Vy_est_tou;
452         *new_Pz_est_tou = temp_new_Pz_est_tou;
453         *new_Vz_est_tou = temp_new_Vz_est_tou;
454
455         /*prints Euler integration data to the output file for later
456         *comparison to actual target data*/
457         //      fprintf(Euler_output_data, "%f %f %f %f %f %f %f %f\n", time_index,
(float)*new_Px_est_tou, (float)*new_Py_est_tou, (float)*new_Pz_est_tou,
(float)*new_Vx_est_tou, (float)*new_Vy_est_tou, (float)*new_Vz_est_tou);
time_index = time_index + dT;

458     }
459
460
461     /*resets the initial velocity and position values that will go to
462     *the open loop filter during the next function iteration.*/
463     initial_x_velocity = ssSetRWorkValue(S, 0, *new_Vx_est_tou);
464     initial_x_position = ssSetRWorkValue(S, 1, *new_Px_est_tou);
465     initial_y_velocity = ssSetRWorkValue(S, 2, *new_Vy_est_tou);
466     initial_y_position = ssSetRWorkValue(S, 3, *new_Py_est_tou);
467     initial_z_velocity = ssSetRWorkValue(S, 4, *new_Vz_est_tou);
468     initial_z_position = ssSetRWorkValue(S, 5, *new_Pz_est_tou);
469
470     reset_index = 1; /*triggers flag to indicate that an index
471                     *reset is needed*/
472
473     integrator_flag = ssSetIWorkValue(S, 1, 1); /*triggers open loop

```

```

474                                     *integrator reset*/
475     }
476
477     if((index <= (int)MAX_INDEX) && (reset_index==0))
478     {
479         index = ssSetIWorkValue(S, 0, index+1); /*increments index value*/
480         velocity_x_data++; /*increments buffer pointers*/
481         position_x_data++;
482         velocity_y_data++;
483         position_y_data++;
484         velocity_z_data++;
485         position_z_data++;
486         x_PVNT_data++;
487         y_PVNT_data++;
488         z_PVNT_data++;
489     }
490
491     else
492     {
493         new_Px_est_tou=new_Px_est_tou-index;
494         new_Vx_est_tou=new_Vx_est_tou-index;
495         new_Py_est_tou=new_Py_est_tou-index;
496         new_Vy_est_tou=new_Vy_est_tou-index;
497         new_Pz_est_tou=new_Pz_est_tou-index;
498         new_Vz_est_tou=new_Vz_est_tou-index;
499
500         index = ssSetIWorkValue(S, 0, 0); /*resets index value to 0*/
501     }
502
503     /*resets the pointer work values for the velocity_data and
504     *position_data buffers*/
505     ssSetPWorkValue(S, 0, (real_T *)velocity_x_data);
506     ssSetPWorkValue(S, 1, (real_T *)position_x_data);
507     ssSetPWorkValue(S, 2, (real_T *)velocity_y_data);
508     ssSetPWorkValue(S, 3, (real_T *)position_y_data);
509     ssSetPWorkValue(S, 4, (real_T *)velocity_z_data);
510     ssSetPWorkValue(S, 5, (real_T *)position_z_data);
511
512     ssSetPWorkValue(S, 12, (real_T *)x_PVNT_data);
513     ssSetPWorkValue(S, 13, (real_T *)y_PVNT_data);
514     ssSetPWorkValue(S, 14, (real_T *)z_PVNT_data);
515 }
516
517
518 /* Function: mdlUpdate =====
519 * Abstract:
520 *   Increment the state for next time around (i.e. a counter).
521 */
522 #define MDL_UPDATE
523 static void mdlUpdate(SimStruct *S, int_T tid)
524 {
525
526     UNUSED_ARG(tid); /* not used in single tasking mode */
527
528 }

```

```

529
530
531 /* Function: mdlTerminate =====
532  * Abstract:
533  *   Required to have this routine.
534  */
535 static void mdlTerminate(SimStruct *S)
536 {
537     real_T *velocity_x_data      = ssGetPWorkValue(S, 0);
538     real_T *position_x_data      = ssGetPWorkValue(S, 1);
539     real_T *velocity_y_data      = ssGetPWorkValue(S, 2);
540     real_T *position_y_data      = ssGetPWorkValue(S, 3);
541     real_T *velocity_z_data      = ssGetPWorkValue(S, 4);
542     real_T *position_z_data      = ssGetPWorkValue(S, 5);
543
544     real_T *new_Vx_est_tou       = ssGetPWorkValue(S, 6);
545     real_T *new_Px_est_tou       = ssGetPWorkValue(S, 7);
546     real_T *new_Vy_est_tou       = ssGetPWorkValue(S, 8);
547     real_T *new_Py_est_tou       = ssGetPWorkValue(S, 9);
548     real_T *new_Vz_est_tou       = ssGetPWorkValue(S, 10);
549     real_T *new_Pz_est_tou       = ssGetPWorkValue(S, 11);
550
551     real_T *x_PVNT_data          = ssGetPWorkValue(S, 12);
552     real_T *y_PVNT_data          = ssGetPWorkValue(S, 13);
553     real_T *z_PVNT_data          = ssGetPWorkValue(S, 14);
554
555     // FILE *Euler_output_data;
556
557     UNUSED_ARG(S); /* unused input argument */
558
559     /*releases data stored in buffers*/
560     free(velocity_x_data);
561     free(position_x_data);
562     free(velocity_y_data);
563     free(position_y_data);
564     free(velocity_z_data);
565     free(position_z_data);
566     free(new_Vx_est_tou);
567     free(new_Px_est_tou);
568     free(new_Vy_est_tou);
569     free(new_Py_est_tou);
570     free(new_Vz_est_tou);
571     free(new_Pz_est_tou);
572     free(x_PVNT_data);
573     free(y_PVNT_data);
574     free(z_PVNT_data);
575
576     /*closes Euler integration data output file*/
577     // fclose(Euler_output_data);
578 }
579
580 #ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
581 #include "simulink.c" /* MEX-file interface mechanism */

```

```
582  #else
583  #include "cg_sfuns.h"      /* Code generation registration function */
584  #endif
```

B. ROAD FOLLOWING FILTER S-FUNCTION

The road following filter is similar in method to the general filter but contains two major differences. First, the S-function receives the PVNT update input just like the general filter S-function, but it utilizes an optimization routine before passing the position update on to the remainder of the program. Since the filter can use the road equations in its calculations, it is able to calculate the best position update in terms of the road parameter, ρ . The optimization function uses a dichotomy method to quickly and accurately find the best ρ value pertaining to the PVNT x, y, z input. The dichotomy method divides the area of the road it is to search in half and uses a step size to define two points on either side of the halfway mark. The function then calculates and compares the distance from these points to the inputted PVNT update. Using the results, the function will reset either the lower or upper boundary and repeat the calculations until a pre-determined tolerance is met. This results in a routine that is much faster than and just as accurate as calculating and comparing distances from each point within a given range along the road to the inputted PVNT update.

The second main difference between the filters is that all of the integration calculations are done using ρ and velocity instead of the x, y , and z coordinates and magnitudes. Once again, this is only possible because the road equations are known before the system is simulated.

1. Manual

File: *s_filter_road_following.c*

Lines 26-57

Complete basic program initializations, library calls, and global variable input.

Lines 26 and 27

Designate the file name and indicate that the file is in C code, to be converted into MEX format and run in MATLAB.

Lines 29-34

Make all the necessary library calls that are required in the program.

Lines 40-42

Take in the three S-function parameters: *MAX_DELAY*, *TIME_STEP*, and *ROAD_EQUATION* from the S-function block in the Simulink model. *MAX_DELAY* is the maximum amount of expected delay in between PVNT updates while *TIME_STEP* is the time step to be used by the C code. NOTE: The time step parameter value must match the discrete time step value found on the simulation parameters menu in Simulink. The *ROAD_EQUATION* parameter is used to define which road model equations are used in the optimization function (there were two different road models used during initial simulation and testing).

Lines 46-48

Convert the parameters into “real_T” format for use in numerical calculations later.

Line 52

Defines the global variable *MAX_INDEX*, used to ensure that buffer overflow does not occur.

Lines 53-56

Define the coefficients for the third order road model and the radius of the circle (meters) for the circular road model.

Line 57

Defines pi as a constant for used in the phase shift of the circular road model equations found in the optimization function (see **lines 103** and **107**)

PVNT_optimization function

Line 69

Lists the inputs to the function along with buffers marked by an asterisk in front of their names.

Lines 71-76

Define and initialize the variables that are only used inside the function such as the upper and lower bounds and the desired tolerance of the final result.

Lines 79-80

Initialize the upper and lower bounds before the dichotomy loop. The *ro_optimize_start* real work value is taken in from the *mdlOutputs* function (see **line 455**).

Lines 85-126

Contain the dichotomy loop.

Line 85

Defines limits the amount of loop iterations to 100 and dictates that the loop should continue until the required tolerance is met.

Lines 87-88

Define the upper and lower ρ limits by dividing the search area of the road in half and adding and subtracting the step size.

Lines 90-99

Compute the position points along the road based on the upper and lower ρ limits if the third order road model is being used based on the inputted parameter from **line 42**.

Lines 101-110

Compute the position points along the road based on the upper and lower ρ limits if the circular road model is being used based on the inputted parameter from **line 42**.

Lines 112-113

Use the distance formula to compute the distance between the PVNT position input and the calculated position points.

The *if/else* loops in **lines 115-122** compare the distance values and, based on the results, reset the right or left boundary to one of the ρ limits.

Lines 124 and 125

Compute the current tolerance and increments the counter pertaining to the *while* loop.

Line 127

Sends out the new ρ update value following successful completion of the dichotomy loop.

Euler_integration function:

Line 137

Lists the inputs to the function along with buffers marked by an asterisk in front of their names.

Lines 144-145

Perform forward Euler integration for the ρ variable, assigning the new position and velocity values to the buffers beginning with “*temp.*”

mdlInitializeSizes function

Line 155

Means that there will be three parameters inputted into the S-function block in Simulink

Lines 156-159

Return an error to MATLAB if the incorrect number of parameters is found.

Line 161

Defines zero continuous states since the model is running with a preset, fixed step time.

Line 162

Defines seven discrete states which must match the number of input ports found in **line 164**.

Lines 165-171

Set the size of each input port.

Lines 172-178

Denote each input port as a direct feed through port.

Line 180

Defines four output ports from the S-function.

Lines 181-184

Define the width of each port.

Line 186

Defines one sample time to be used

Lines 187-190

Define the number of real, integer, pointer, and mode work vectors to be used in the program. The work vectors can be thought of as a value of a certain type (real, integer, pointer, etc.) that is stored in persistent memory. This means that the value will be stored even while the program is called multiple times.

Line 191

Defines the number of zero crossings to be zero as it is not used in the filter program.

mdlInitializeSampleTimes function**Line 205**

Defines the program's sample time to be set to dT , which come from the second parameter input to the S-function block in **line 46**.

Line 206

Indicates a 0.0 second offset time and **line 208** indicates that a function call is made on the first element of the first output port.

mdlStart function

Defines all of the variables that need to be initialized only once, i.e. the very first time the program is run in the simulation.

Line 222

Predefines the integer work vectors for the index counter and the integrator flag that indicates when the discrete integrator blocks in the open-loop filter subsystem need to be reset.

Line 223

Predefines the real work vectors for the initial x, y, and z positions and velocities.

Lines 228-230

Predefine the buffers that are used in the code for data storage

Lines 235-241

Initialize the buffers to a number of positions equal to *MAX_INDEX* (from **line 52**) with each position having enough memory to store a piece of data with the size *real_T*. The *calloc* command also initializes every position in the buffers to zero.

Lines 243-247

Define the first value for the index counter, integrator reset flag, position and velocity initial conditions, and the starting ρ variable for the optimization function to be zero.

Lines 251-257

Set the pointer work vectors to point to the first position of each of the buffers

Lines 259-267

Set and store the initial integer and real work values.

mdlOutputs function

Lines 278-294

Contain the input and output declarations.

Lines 278-279 and 285-286

Define the pointers and values of the position and velocity estimates coming from the open loop filter function call.

Lines 280 and 287

Define the pointer and value coming in from the PVNT update delay subsystem

Lines 281-283 and 288-290

Define the actual PVNT update (*x,y,z*) from the target model subsystem.

Lines 284 and 291

Designate a port for the clock input.

Lines 292-293

Define output ports for the position and velocity initial conditions to the open-loop filter function call.

Line 294

Defines the integrator reset signal, which is also fed into the open-loop filter function call.

Lines 296-303

Contain declarations for the work values and the buffers which match the declarations found in the *mdlStart* function.

Lines 309-319

Define and initialize the non-persistent variables that are used only in the *Euler_integration*, *PVNT_optimization*, and *mdlOutputs* function.

Lines 331-337

Retrieves the values that were stored in the pointer work vectors.

Lines 340-344

Retrieves the values that were stored in the integer and real work vectors.

Lines 352-389

Contained in an *if* loop that executes only if the index counter is less than or equal to the preset *MAX_INDEX* value. This ensures that no data is written to the buffers beyond their maximum preset number of storage positions, reducing the risk of buffer overflow.

Lines 354-355

Set the second and third output ports to the position and velocity initial conditions.

Line 357

Sets the *integrator_reset* output to the integer work value *integrator_flag*.

Lines 360-364

Call the open loop filter function call block in the Simulink diagram through the first output port.

Lines 366-367

Take in the estimated position and velocity values from the first six inputs (arriving from the outputs of the open-loop filter function call).

Lines 370-371

Set the respective buffer values to the inputted position and velocity estimates. These values are then also stored in the real work vectors designating position and velocity initial conditions.

Line 378

Resets the integrator flag integer work value to zero (if it was set to one following the Euler integration loop, see **line 460**).

Lines 381-383

Take in the PVNT position update (x,y,z) from the true target model subsystem in the Simulink diagram.

Lines 386-388

Assign PVNT position update to buffers.

Lines 391-447

Contained in an *if* loop that is only triggered if the input from the PVNT delay subsystem is set high, indicating that a PVNT update is available.

Lines 397-399

Adjust the pointers to each PVNT buffer so that they now refer to time τ . This is controlled by the *index* integer work vector which is incremented after each iteration of the *mdlOutputs* function (see **line 471**).

Line 402

Calls the *PVNT_optimization* function and receives the new ρ value.

Lines 407-408

Adjust the pointers to each position and velocity buffer so that they now refer to time τ , the time to which the PVNT update refers.

Line 412

Calculates the difference between the estimated position data at time τ and the PVNT position update at time τ for ρ .

Lines 416-417

Set up the values for the first position of the buffers that are used in the *Euler_integration* function and to pass on the updated position and velocity data to the open-loop filter function call.

Line 423

Begins the asynchronous portion of the S-function. The *for* loop runs enough times to move the new estimated position and velocity values from time τ to time t (current system time), which is controlled by the *index* integer work vector value.

Lines 426-429

Set the *delta* variable value originally set in **lines 412** to zero after the first iteration of the *for* loop, allowing for normal, dead-reckoning style integration.

Line 431

Passes the required variables to the *Euler_integration* function in **lines 131-146**. Additionally, the “&” in front of the *temp* buffers indicate that their changed values from the *Euler_integration* function will be saved after the function executes.

Lines 435-436

Increment the pointer values for the buffers that will contain the updated position and velocity estimates.

Lines 440-441

Actually set the buffers equal to the updates.

After the *for* loop runs the appropriate number of times to arrive at time t , the final value from each of the buffers containing the updated position and velocity estimates are passed to the initial condition real work vectors in **lines 451-452**.

Line 455

The *ro_optimize_start* real work value (used in the optimization function) is set. Additionally, two integrator reset values are set. The first is the *reset_index* variable on **line 457** set equal to one and used inside the S-function program on **line 464**.

The second is the *integrator_flag* integer work value on **line 460** that is outputted to the open-loop function call outside the S-function block.

The remainder of the buffer pointer incrementation/resets take place in the *if/else* loop in **lines 464-478**. The *if* loop portion checks to see if the current *index* variable value is less than the preset *MAX_INDEX* value and if the *reset_index* variable value is equal to zero (indicating that a PVNT update did not arrive during the current *mdlOutputs* function iteration. If so, the *index* integer work value is incremented along with the pointers to the position and velocity data buffers.

If the criterion for the *if* loop are not met, meaning that a PVNT update has occurred, the buffer pointers are all reset back to their first position and the *index* integer work value is set to zero. This ensures that the buffers are simply overwritten with the new data until the next PVNT update and buffer overflow does not occur. Finally, the pointer work values are updated to now designate the new pointer values for the position and velocity data buffers.

Lines 482-486

Reset the pointer work values for the PVNT, position, and velocity buffers.

mdlUpdate function

This would be the function in which states would be incremented if they were used in the program. Since the filter design does not use these states, however, the *mdlUpdate* function is only left in the program as a formality.

mdlTerminate function

In this case, all of the data from the buffers must be cleared to avoid errors when re-running the simulation multiple times.

Lines 510-516

Designate each of the buffers that were originally defined in the *mdlStart* function

Lines 523-529

Actually release the data stored in the buffers.

2. Code

```
1  /* File   : s_filter_road_following.c
2  * Abstract:
3  *
4  * This S-function is a combination of an open-loop filter using a
5  * function call subsystem and an asynchronous filter contained in the
6  * C code of the S-function. The model is used for a target tracking
7  * system, utilizing a delayed position update at different time
8  * intervals. When the position update (labeled PVNT) is not
9  * available, the S-function calls the open-loop filter and stores the
10 * results. When the delayed position update arrives, the loop
11 * containing the asynchronous filter is run to update the previous
12 * data from time tau (corresponding to the PVNT update) to time t
13 * (corresponding to the current time) using buffers to store all
14 * data. The model takes in parameters from the S-function block in
15 * the Simulink model for the maximum amount of delay (seconds) and
16 * the desired time step (seconds). The user can easily manipulate
17 * these parameters without having to change C code in the S-function
18 *
19 * For more details about S-functions, see
20 * matlabroot/simulink/src/sfuntmpl_doc.c
21 *
22 * Copyright 1990-2006 The MathWorks, Inc.
23 * $Revision: 1.15.4.3 $
24 */
25
26 #define S_FUNCTION_NAME s_filter_road_following
27 #define S_FUNCTION_LEVEL 2
28
29 #include "simstruc.h"
30
31 #include <stdlib.h>
32 #include <stdio.h>
33 #include <string.h>
34 #include <math.h>
35
36
37 /* Input Arguments */
38 /*takes in parameters that define a max value for the PVNT update delay and
39 *the desired time step*/
40 #define MAX_DELAY          ssGetSFcnParam(S,0)
41 #define TIME_STEP          ssGetSFcnParam(S,1)
42 #define ROAD_EQUATION      ssGetSFcnParam(S,2)
43
44 /*converts the above parameters from structs to allow them to be used in
45 *computations*/
46 #define dT                  ((real_T) mxGetPr(TIME_STEP)[0])
47 #define DELAY_MAX           ((real_T) mxGetPr(MAX_DELAY)[0])
48 #define road_equation_selection ((real_T) mxGetPr(ROAD_EQUATION)[0])
49
50 /*defines global constant that is used to prevent buffer overflow and
```

```

51     coefficients for road equation*/
52     #define MAX_INDEX          (DELAY_MAX/dT)
53     #define coeff_3            0.0000192
54     #define coeff_2            -0.025
55     #define coeff_1            9.74
56     #define radius             2865.0
57     #define pi                  3.14159
58
59
60     /* Function: PVNT_optimization
61     * Abstract:
62     * Performs distance measurment between the original PVNT update
63     * coordinates and coordinates defined by the road equation. It then finds
64     * the closest point on the road to the PVNT coordinates and sets that
65     * point as the actual PVNT position update. The third parameter in the
66     * S-function block determines which optimization equation is called
67     * based on which road equation is to be used.
68     */
69     void PVNT_optimization (real_T *x_PVNT_data, real_T *y_PVNT_data, real_T
70     *z_PVNT_data, real_T ro_optimize_start, real_T *ro_star, int road_eq_selector)
71     {
72         float lower_ro_limit=0.0, upper_ro_limit=0.0;
73         float x_left=0.0, y_left=0.0, z_left=0.0, x_right=0.0, y_right=0.0, z_right=0.0;
74         float distance_1=0.0, distance_2=0.0, step_size=0.5;
75         float left_boundary = 0.0, right_boundary = 0.0, tolerance = 0.000001;
76         float L = 2*tolerance; /*sets L so it is initially higher than tolerance*/
77         int j = 0;
78
79         /*initializes upper and lower bounds for optimization loop*/
80         left_boundary = (float)ro_optimize_start - 50.0;
81         right_boundary = (float)ro_optimize_start + 50.0;
82
83         /*optimization routine for PVNT update: utilizes dichotomy technique to
84         *compare distance from points along the road model to PVNT update
85         *point. final value is outputted as the ro_star update*/
86         while (L>=tolerance && j<=100)
87         {
88             lower_ro_limit = (right_boundary+left_boundary-step_size)/2.0;
89             upper_ro_limit = (right_boundary+left_boundary+step_size)/2.0;
90
91             if (road_eq_selector == 0)
92             {
93                 x_left = lower_ro_limit;
94                 y_left = coeff_3*pow(lower_ro_limit,3) + coeff_2*pow(lower_ro_limit,2) +
95                 coeff_1*lower_ro_limit;
96                 z_left = 0.0;
97
98                 x_right = upper_ro_limit;
99                 y_right = coeff_3*pow(upper_ro_limit,3) + coeff_2*pow(upper_ro_limit,2) +
100                 coeff_1*upper_ro_limit;
101                 z_right = 0.0;
102             }

```

```

103     x_left = radius + radius * sin(lower_ro_limit/radius + 3*pi/2);
104     y_left = radius*sin(lower_ro_limit/radius);
105     z_left = 0.0;
106
107     x_right = radius + radius * sin(upper_ro_limit/radius + 3*pi/2);
108     y_right = radius*sin(upper_ro_limit/radius);
109     z_right = 0.0;
110 }
111
112     distance_1 = sqrt(pow(x_left-x_PVNT_data,2) + pow(y_left-y_PVNT_data,2) +
113     pow(z_left-z_PVNT_data,2));
114     distance_2 = sqrt(pow(x_right-x_PVNT_data,2) + pow(y_right-y_PVNT_data,2) +
115     pow(z_right-z_PVNT_data,2));
116
117     if(distance_1 <= distance_2)
118     {
119         right_boundary = upper_ro_limit;
120     }
121     else
122     {
123         left_boundary = lower_ro_limit;
124     }
125
126     L = fabs(distance_1 - distance_2);          /*computes current error*/
127     j++;                                         /*increments counter*/
128 }
129
130
131 /* Function: Euler_integration =====
132 * Abstract:
133 * Performs asynchronous forward Euler integration once the PVNT update is
134 * received in order to rewrite over the previous data from time tau to
135 * time t.
136 */
137 void Euler_integration(double k1, double k2, float delta_ro_tou, float time_step, real_T
138 *new_ro_est_tou, real_T *new_V_sca_est_tou, real_T *temp_new_V_sca_est_tou,
139 real_T *temp_new_ro_est_tou)
140 {
141     /*performs asynchronous double integration with a time step
142     *equal to dT seconds and stores the results in a temp variable
143     *to be transferred to the buffers after they have been
144     *incremented*/
145     *temp_new_ro_est_tou = *new_ro_est_tou+ (*new_V_sca_est_tou +
146     k1*delta_ro_tou)*time_step;
147     *temp_new_V_sca_est_tou=*new_V_sca_est_tou+ (k2*delta_ro_tou)*time_step;
148 }
149
150 /* Function: mdlInitializeSizes =====
151 * Abstract:
152 * Setup sizes of the various vectors.
153 */

```

```

153 static void mdlInitializeSizes(SimStruct *S)
154 {
155     ssSetNumSFcnParams(S, 3); /* Number of expected parameters */
156     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
157     {
158         return; /* Parameter mismatch will be reported by Simulink */
159     }
160
161     ssSetNumContStates(S, 0); /*defines 0 continuous states*/
162     ssSetNumDiscStates(S, 7); /*defines 7 discrete state*/
163
164     if (!ssSetNumInputPorts(S, 7)) return; /*defines 7 input ports*/
165     ssSetInputPortWidth(S, 0, 1); /*sets input 1 port size to 1*/
166     ssSetInputPortWidth(S, 1, 1); /*sets input 2 port size to 1*/
167     ssSetInputPortWidth(S, 2, 1); /*sets input 3 port size to 1*/
168     ssSetInputPortWidth(S, 3, 1); /*sets input 4 port size to 1*/
169     ssSetInputPortWidth(S, 4, 1); /*sets input 5 port size to 1*/
170     ssSetInputPortWidth(S, 5, 1); /*sets input 6 port size to 1*/
171     ssSetInputPortWidth(S, 6, 1); /*sets input 7 port size to 1*/
172     ssSetInputPortDirectFeedThrough(S, 0, 1);
173     ssSetInputPortDirectFeedThrough(S, 1, 1);
174     ssSetInputPortDirectFeedThrough(S, 2, 1);
175     ssSetInputPortDirectFeedThrough(S, 3, 1);
176     ssSetInputPortDirectFeedThrough(S, 4, 1);
177     ssSetInputPortDirectFeedThrough(S, 5, 1);
178     ssSetInputPortDirectFeedThrough(S, 6, 1);
179
180     if (!ssSetNumOutputPorts(S, 4)) return;
181     ssSetOutputPortWidth(S, 0, 1); /*sets output port 1 width to 1*/
182     ssSetOutputPortWidth(S, 1, 1); /*sets output port 2 width to 1*/
183     ssSetOutputPortWidth(S, 2, 1); /*sets output port 3 width to 1*/
184     ssSetOutputPortWidth(S, 3, 1); /*sets output port 4 width to 1*/
185
186     ssSetNumSampleTimes(S, 1);
187     ssSetNumRWork(S, 3); /*real vector*/
188     ssSetNumIWork(S, 2); /*integer vector*/
189     ssSetNumPWork(S, 7); /*pointer vector*/
190     ssSetNumModes(S, 0); /*mode vector*/
191     ssSetNumNonsampledZCs(S, 0); /*number of zero crossings*/
192
193     /* Take care when specifying exception free code - see sfuntmpl_doc.c */
194     ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
195 }
196
197
198 /* Function: mdlInitializeSampleTimes =====
199 * Abstract:
200 * Discrete sample time of dT seconds and specify that we are doing
201 * function-calls on the 1st element of the 1st output port.
202 */
203 static void mdlInitializeSampleTimes(SimStruct *S)
204 {
205     ssSetSampleTime(S, 0, dT); /*sets sample time to dT seconds*/
206     ssSetOffsetTime(S, 0, 0.0); /*indicates 0 offset time*/
207

```

```

208     ssSetCallSystemOutput(S,0);                /* call on first element */
209     ssSetModelReferenceSampleTimeDefaultInheritance(S);
210 }
211
212
213 /*Function: mdlStart =====
214
215 *Abstract:
216 * This function sets up the variables passed between the function and
216 * the s-function.
217 */
218 #define MDL_START
219
220 static void mdlStart(SimStruct *S)
221 {
222     int_T index, integrator_flag;
223     real_T initial_position, initial_velocity, ro_optimize_start;
224
225     /*The real_T variables below denote the buffers used to store
226     *velocity and position data over multiple iterations of the
227     *s-function*/
228     real_T *velocity_data, *position_data;
229     real_T *new_V_sca_est_tou, *new_ro_est_tou;
230     real_T *x_PVNT_data, *y_PVNT_data, *z_PVNT_data;
231
232     /*The buffers are allocated enough memory to store 'MAX_INDEX' data
233     *with each data space being 'real_T' size. The 'calloc' command also
234     *initializes the buffers*/
235     velocity_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
236     position_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
237     new_V_sca_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
238     new_ro_est_tou = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
239     x_PVNT_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
240     y_PVNT_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
241     z_PVNT_data = (real_T *) calloc(MAX_INDEX, sizeof(real_T));
242
243     index = 0;                /*initializes index to 0*/
244     integrator_flag = 0;      //sets integration reset flag to 0
245     initial_velocity = 0.0;    //initializes position and velocity
246     initial_position = 0.0;    //IC's to 0
247     ro_optimize_start = 0.0;
248
249
250     /*Sets the pointer work variables for the buffers*/
251     ssSetPWorkValue(S, 0, (real_T *)velocity_data);
252     ssSetPWorkValue(S, 1, (real_T *)position_data);
253     ssSetPWorkValue(S, 2, (real_T *)new_V_sca_est_tou);
254     ssSetPWorkValue(S, 3, (real_T *)new_ro_est_tou);
255     ssSetPWorkValue(S, 4, (real_T *)x_PVNT_data);
256     ssSetPWorkValue(S, 5, (real_T *)y_PVNT_data);
257     ssSetPWorkValue(S, 6, (real_T *)z_PVNT_data);
258
259     ssSetIWorkValue(S, 0, index);                /*sets the first integer work
260                                                    *value to the index variable*/
261

```

```

262     ssSetIWorkValue(S, 1, integrator_flag);      /*sets the second integer work
263                                                    *value to the integrator flag*/
264
265     ssSetRWorkValue(S, 0, initial_velocity);      /*sets the real work values*/
266     ssSetRWorkValue(S, 1, initial_position);
267     ssSetRWorkValue(S, 2, ro_optimize_start);
268 }
269
270
271 /* Function: mdlOutputs =====
272  * Abstract:
273  *   Issue ssCallSystemWithTid on 1st output element of 1st output port.
274  */
275 static void mdlOutputs(SimStruct *S, int_T tid)
276 {
277     /*S-function input and output declarations*/
278     real_T      *ro_est      = ssGetRealDiscStates(S,0);
279     real_T      *V_sca_est   = ssGetRealDiscStates(S,1);
280     real_T      *PVNT        = ssGetRealDiscStates(S,2);
281     real_T      *x_PVNT      = ssGetRealDiscStates(S,3);
282     real_T      *y_PVNT      = ssGetRealDiscStates(S,4);
283     real_T      *z_PVNT      = ssGetRealDiscStates(S,5);
284     real_T      *clock       = ssGetRealDiscStates(S,6);
285     InputRealPtrsType ro_est_Ptrs = ssGetInputPortRealSignalPtrs(S,0);
286     InputRealPtrsType V_sca_est_Ptrs = ssGetInputPortRealSignalPtrs(S,1);
287     InputRealPtrsType PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,2);
288     InputRealPtrsType x_PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,3);
289     InputRealPtrsType y_PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,4);
290     InputRealPtrsType z_PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,5);
291     InputRealPtrsType clock_Ptrs = ssGetInputPortRealSignalPtrs(S,6);
292     real_T      *TgtP_IC     = ssGetOutputPortRealSignal(S,1);
293     real_T      *TgtV_IC     = ssGetOutputPortRealSignal(S,2);
294     real_T      *integrator_reset = ssGetOutputPortRealSignal(S,3);
295
296     int_T index, integrator_flag;
297     real_T initial_velocity, initial_position;
298     real_T temp_new_V_sca_est_tou, temp_new_ro_est_tou;
299
300     /*buffer declarations for mdlOutputs*/
301     real_T *velocity_data, *position_data;
302     real_T *new_V_sca_est_tou, *new_ro_est_tou;
303     real_T *x_PVNT_data, *y_PVNT_data, *z_PVNT_data;
304
305     /*defines pointer to output file for forward Euler integration results*/
306     //FILE *Euler_output_data;
307
308     /*defines intermediate position and velocity matrices*/
309     float delta_ro_tou = 0.0, time_index = 0.0;
310     int i = 0; /*counter*/
311     int road_eq_selector = road_equation_selection; /*picks which road
312                                                    *equation is to be
313                                                    *used*/
314     int reset_index = 0; /*flag indicating and index reset to 0*/
315     double k1=0.5, k2=0.5; /*sets integrator gains*/
316     float delay = 0.0, time_step = dT;

```

```

317
318 /*optimization variables*/
319 real_T ro_star, ro_optimize_start;
320
321 /*
322  * ssCallSystemWithTid is used to execute a function-call subsystem. The
323  * 2nd argument is the element of the 1st output port index which
324  * connected to the function-call subsystem. Function-call subsystems
325  * can be driven by the first output port of s-function blocks.
326  */
327
328 UNUSED_ARG(tid); /* not used in single tasking mode */
329
330 /*Retrieves the pointer work values for the buffers*/
331 velocity_data = (real_T *)ssGetPWorkValue(S, 0);
332 position_data = (real_T *)ssGetPWorkValue(S, 1);
333 new_V_sca_est_tou = (real_T *)ssGetPWorkValue(S, 2);
334 new_ro_est_tou = (real_T *)ssGetPWorkValue(S, 3);
335 x_PVNT_data = (real_T *)ssGetPWorkValue(S, 4);
336 y_PVNT_data = (real_T *)ssGetPWorkValue(S, 5);
337 z_PVNT_data = (real_T *)ssGetPWorkValue(S, 6);
338
339 /*Retrieves integer and real work values*/
340 index = ssGetIWorkValue(S,0);
341 integrator_flag = ssGetIWorkValue(S,1);
342 initial_velocity = ssGetRWorkValue(S,0);
343 initial_position = ssGetRWorkValue(S,1);
344 ro_optimize_start = ssGetRWorkValue(S,2);
345
346 /*creates .txt file for output results*/
347 //Euler_output_data = fopen("Euler_data_rf.txt", "w");
348
349 /*Entire sequence is in an 'if' loop to ensure that there is no
350 *overflow for the position and velocity arrays (defined with a maximum
351 *of MAX_INDEX data points.)*/
352 if(index <= (int)MAX_INDEX)
353 {
354     TgtP_IC[0] = initial_position; /*sets outputs to initial V and P*/
355     TgtV_IC[0] = initial_velocity;
356
357     integrator_reset[0] = integrator_flag; /*sets output 3 to integration
358                                             *reset flag*/
359
360     if(!ssCallSystemWithTid(S,0,tid)) /*calls system with task ID 1*/
361     {
362         /* Error occurred which will be reported by Simulink */
363         return;
364     }
365
366     ro_est_Ptrs = ssGetInputPortRealSignalPtrs(S,0);/*Gets inputs*/
367     V_sca_est_Ptrs = ssGetInputPortRealSignalPtrs(S,1);
368
369     /*assigns the position and velocity data to the buffers*/
370     *position_data = (real_T)*ro_est_Ptrs[0];
371     *velocity_data = (real_T)*V_sca_est_Ptrs[0];

```

```

372
373      /*resets the initial velocity and position values*/
374      initial_velocity = ssSetRWorkValue(S, 0, (real_T)*V_sca_est_Ptrs[0]);
375      initial_position = ssSetRWorkValue(S, 1, (real_T)*ro_est_Ptrs[0]);
376
377      /*resets the integrator reset to 0*/
378      integrator_flag = ssSetIWorkValue(S, 1, 0);
379
380      /*takes in x, y, and z coordinates from PVNT update*/
381      x_PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,3);
382      y_PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,4);
383      z_PVNT_Ptrs = ssGetInputPortRealSignalPtrs(S,5);
384
385      /*assigns coordinates to buffers*/
386      *x_PVNT_data = (real_T)*x_PVNT_Ptrs[0];
387      *y_PVNT_data = (real_T)*y_PVNT_Ptrs[0];
388      *z_PVNT_data = (real_T)*z_PVNT_Ptrs[0];
389  }
390
391  if ((real_T)*PVNT_Ptrs[0] >= 0.99)
392  /*indicates pulse is high (PVNT update present)*/
393  {
394
395      /*goes back delay/dT spaces in the PVNT position buffers to get the
396      *actual PVNT position update (NOTE: index = delay/dT)*/
397      x_PVNT_data = x_PVNT_data - index;
398      y_PVNT_data = y_PVNT_data - index;
399      z_PVNT_data = z_PVNT_data - index;
400
401
402      PVNT_optimization (x_PVNT_data, y_PVNT_data, z_PVNT_data,
403      ro_optimize_start, &ro_star, road_eq_selector);
404
405      /*calls the estimated ro and velocity values at time tou from the
406      *buffers*/
407      position_data = position_data - index;
408      velocity_data = velocity_data - index;
409
410      /*calculates the difference between the ro_star update value and the
411      *estimated ro value at time tou*/
412      delta_ro_tou = ro_star - *position_data;
413
414      /*sets up the initial conditions based on the ro input from the
415      *PVNT update*/
416      *new_ro_est_tou = ro_star;
417      *new_V_sca_est_tou = *velocity_data;
418
419      /*sets up time output for Euler_data file*/
420      delay = index;
421      time_index = *clock_Ptrs[0] - (delay / dT);
422
423      for (i=0; i<index; i++)      /*increments counter from 0 to the
424                                  *maximum value of the index*/
425  {

```

```

426         if (i != 0)           /*allows normal integration after first iteration*/
427         {
428             delta_ro_tou = 0.0;
429         }
430
431         Euler_integration(k1, k2, delta_ro_tou, time_step, new_ro_est_tou,
432             new_V_sca_est_tou, &temp_new_V_sca_est_tou, &temp_new_ro_est_tou);
433
434         /*increments the new_ro_est_tou and new_V_sca_est_tou buffer
435         *pointers*/
436         new_ro_est_tou++;
437         new_V_sca_est_tou++;
438
439         /*sets the now incremented buffers equal to the results from
440         *the forward Euler integration*/
441         *new_ro_est_tou = temp_new_ro_est_tou;
442         *new_V_sca_est_tou = temp_new_V_sca_est_tou;
443
444         /*prints Euler integration data to the output file for later
445         *comparison to actual target data*/
446         //fprintf(Euler_output_data, "%f %f %f \n", time_index, (float)*new_ro_est_tou,
447             (float)*new_V_sca_est_tou);
448         time_index = time_index + dT;
449     }
450
451     /*resets the initial velocity and position values that will go to
452     *the open loop filter during the next function iteration.*/
453     initial_velocity = ssSetRWorkValue(S, 0, *new_V_sca_est_tou);
454     initial_position = ssSetRWorkValue(S, 1, *new_ro_est_tou);
455
456     /*resets the initial ro value for use in the optimization loop*/
457     ro_optimize_start = ssSetRWorkValue(S, 2, initial_position);
458
459     reset_index = 1; /*triggers flag to indicate that an index
460                     *reset is needed*/
461
462     integrator_flag = ssSetIWorkValue(S, 1, 1); /*triggers open loop
463                                                 *integrator reset*/
464 }
465
466 if((index <= (int)MAX_INDEX) && (reset_index==0)) /*checks to see if flag is set*/
467 {
468     velocity_data++; /*increments buffer pointers*/
469     position_data++;
470     x_PVNT_data++;
471     y_PVNT_data++;
472     z_PVNT_data++;
473     index = ssSetIWorkValue(S, 0, index+1); /*increments index value*/
474 }
475 else
476 {
477     new_ro_est_tou=new_ro_est_tou-index;
478     new_V_sca_est_tou=new_V_sca_est_tou-index;
479     index = ssSetIWorkValue(S, 0, 0); /*resets index value to 0*/
480 }

```

```

478
480     /*resets the pointer work values for the velocity_data and
481     *position_data buffers*/
482     ssSetPWorkValue(S, 0, (real_T *)velocity_data);
483     ssSetPWorkValue(S, 1, (real_T *)position_data);
484     ssSetPWorkValue(S, 4, (real_T *)x_PVNT_data);
485     ssSetPWorkValue(S, 5, (real_T *)y_PVNT_data);
486     ssSetPWorkValue(S, 6, (real_T *)z_PVNT_data);
487 }
488
489
490 /* Function: mdlUpdate =====
491  * Abstract:
492  *   Increment the state for next time around (i.e. a counter).
493  */
494
495 #define MDL_UPDATE
496 static void mdlUpdate(SimStruct *S, int_T tid)
497 {
498
499     UNUSED_ARG(tid); /* not used in single tasking mode */
500
501 }
502
503
504 /* Function: mdlTerminate =====
505  * Abstract:
506  *   Required to have this routine.
507
508 static void mdlTerminate(SimStruct *S)
509 {
510     real_T *velocity_data      = ssGetPWorkValue(S, 0);
511     real_T *position_data      = ssGetPWorkValue(S, 1);
512     real_T *new_V_sca_est_tou  = ssGetPWorkValue(S, 2);
513     real_T *new_ro_est_tou     = ssGetPWorkValue(S, 3);
514     real_T *x_PVNT_data        = ssGetPWorkValue(S, 4);
515     real_T *y_PVNT_data        = ssGetPWorkValue(S, 5);
516     real_T *z_PVNT_data        = ssGetPWorkValue(S, 6);
517
518     //FILE *Euler_output_data;
519
520     UNUSED_ARG(S); /* unused input argument */
521
522     /*releases data stored in buffers*/
523     free(velocity_data);
524     free(position_data);
525     free(new_V_sca_est_tou);
526     free(new_ro_est_tou);
527     free(x_PVNT_data);
528     free(y_PVNT_data);
529     free(z_PVNT_data);
530
531     /*closes Euler integration data output file*/
532     //fclose(Euler_output_data);
533 }

```

```
534
535 #ifdef MATLAB_MEX_FILE      /* Is this file being compiled as a MEX-file? */
536 #include "simulink.c"        /* MEX-file interface mechanism */
537 #else
538 #include "cg_sfun.h"         /* Code generation registration function */
539 #endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [1] Wolfgang Baer. "Generating One-Meter Terrain Data for Tactical Simulations." In Military Intelligence Professional Bulletin (Oct-Dec, 2002) [electronic bulletin board]. 19 May 2007.
http://findarticles.com/p/articles/mi_m0IBS/is_4_28/ai_94538586.
- [2] Benjamin C. Kuo. *Digital Control Systems*. San Francisco: Holt, Rinehart and Winston, Inc., 1980.
- [3] "Perspective View Nascent Technologies." 19 May 2007.
<http://www.trac.nps.navy.mil/pvnt/>.
- [4] Robert A. Prince. "Autonomous Visual Tracking of Stationary Targets Using Unmanned Aerial Vehicles." Master's Thesis, Naval Postgraduate School, June 2004.
- [5] Chin Khoon Quek. "Vision Based Control and Target Range Estimation for Small Unmanned Aerial Vehicle." Master's Thesis, Naval Postgraduate School, December 2005.
- [6] Kwee Chye Yap. "Incorporating Target Mensuration System for Target Motion Estimation Along a Road Using Asynchronous Filter." Master's Thesis, Naval Postgraduate School, December 2006.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Prof Anthony Healey
Chairman, Department of Mechanical and Astronautical Engineering
Naval Postgraduate School
Monterey, California
4. Prof Isaac Kaminer
Naval Postgraduate School
Monterey, California
5. Dr. Vladimir Dobrokhodov
Naval Postgraduate School
Monterey, California
6. Dr. Eng. Ioannis Kitsios
Naval Postgraduate School
Monterey, California